eBook Gratuit

APPRENEZ dart

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

#dart

Table des matières

À propos

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

It is an unofficial and free dart 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 dart.

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 Dart

Remarques

Dart est un langage de programmation open-source, basé sur des classes et éventuellement typé, destiné à la création d'applications Web - à la fois sur le client et le serveur - créé par Google. Les objectifs de conception de Dart sont les suivants:

- Créez un langage structuré mais flexible pour la programmation Web.
- Rendez Dart familier et naturel aux programmeurs et donc facile à apprendre.
- Assurez-vous que Dart offre de hautes performances sur tous les navigateurs et environnements Web modernes, allant des petits ordinateurs de poche à l'exécution côté serveur.

Dart cible un large éventail de scénarios de développement, allant d'un projet individuel sans trop de structure à un projet à grande échelle nécessitant des types formels dans le code pour indiquer l'intention du programmeur.

Pour prendre en charge cette large gamme de projets, Dart fournit les fonctionnalités et outils suivants:

- **Types facultatifs:** cela signifie que vous pouvez commencer à coder sans types et les ajouter ultérieurement si nécessaire.
- **Isole:** programmation simultanée sur serveur et client
- **Accès facile au DOM:** utilisation de sélecteurs CSS (de la même manière que jQuery le fait)
- **Outils Dart IDE: les** plug-ins Dart existent pour de nombreux IDE couramment utilisés, Ex: WebStorm .
- **Dartium:** une version du navigateur Web Chromium avec une machine virtuelle Dart intégrée

Liens

- La page d'accueil Dart
- Actualités et mises à jour officielles
- The Dartosphere - Une collection de blogs récents sur Dart
- Communauté Dartisans Dartisans sur Google+
- Développement Web Dart - Page Groupes Google
- Dart Language Misc - Page Groupes Google
- DartLang sub-Reddit

Documentation

- Tour du langage Dart
- Visite des bibliothèques Dart
- Échantillons de code Dart
- Référence de l'API Dart

FAQ

- Questions fréquemment posées

Versions

Version	Date de sortie
1.22.1	2017-02-22
1.22.0	2017-02-14
1.21.1	2016-01-13
1,21.0	2016-12-07
1.20.1	2016-10-13
1.20.0	2016-10-11
1.19.1	2016-09-07
1.19.0	2016-08-26
1.18.1	2016-08-02
1,18.0	2016-07-27
1.17.1	2016-06-10
1,17,0	2016-06-06
1.16.1	2016-05-23
1,16,0	2016-04-26
1,15.0	2016-03-09
1.14.2	2016-02-09
1.14.1	2016-02-03
1.14.0	2016-01-28

Version	Date de sortie
1.13.2	2016-01-05
1.13.1	2015-12-17
1.13.0	2015-11-18
1.12.2	2015-10-21
1.12.1	2015-09-08
1.12.0	2015-08-31
1.11.3	2015-08-03
1.11.1	2015-07-02
1.11.0	2015-06-24
1.10.1	2015-05-11
1.10.0	2015-04-24
1.9.3	2015-04-13
1.9.1	2015-03-25
1.8.5	2015-01-13
1.8.3	2014-12-01
1.8.0	2014-11-27
1.7.2	2014-10-14
1.6.0	2014-08-27
1.5.8	2014-07-29
1.5.3	2014-07-03
1.5.2	2014-07-02
1.5.1	2014-06-24
1.4.3	2014-06-16
1.4.2	2014-05-27
1.4.0	2014-05-20

Version	Date de sortie
1,3.6	2014-04-30
1.3.3	2014-04-16
1.3.0	2014-04-08
1.2.0	2014-02-25
1.1.3	2014-02-06
1.1.1	2014-01-15
1.0.0.10_r30798	2013-12-02
1.0.0.3_r30188	2013-11-12
0.8.10.10_r30107	2013-11-08
0.8.10.6_r30036	2013-11-07
0.8.10.3_r29803	2013-11-04

Examples

Installation ou configuration

Le Dart SDK comprend tout ce dont vous avez besoin pour écrire et exécuter le code Dart: VM, bibliothèques, analyseur, gestionnaire de paquets, générateur de documents, formateur, débogueur, etc. Si vous faites du développement Web, vous aurez également besoin de Dartium.

Installation automatisée et mises à jour

- Installation de Dart sur Windows
- Installation de Dart sur Mac
- Installation de Dart sur Linux

Installation manuelle

Vous pouvez également installer manuellement n'importe quelle version du SDK .

Bonjour le monde!

Créez un nouveau fichier nommé `hello_world.dart` avec le contenu suivant:

```
void main() {
```

```
  print('Hello, World!');
}
```

Dans le terminal, accédez au répertoire contenant le fichier `hello_world.dart` et tapez ce qui suit:

```
dart hello_world.dart
```

Appuyez sur `Entrée` pour afficher `Hello, World!` dans la fenêtre du terminal.

Http Request

Html

```
<img id="cats"></img>
```

Dard

```
import 'dart:html';

/// Stores the image in [blob] in the [ImageElement] of the given [selector].
void setImage(selector, blob) {
  FileReader reader = new FileReader();
  reader.onLoad.listen((fe) {
    ImageElement image = document.querySelector(selector);
    image.src = reader.result;
  });
  reader.readAsDataUrl(blob);
}

main() async {
  var url = "https://upload.wikimedia.org/wikipedia/commons/2/28/Tortoiseshell_she-cat.JPG";

  // Initiates a request and asynchronously waits for the result.
  var request = await HttpRequest.request(url, responseType: 'blob');
  var blob = request.response;
  setImage("#cats", blob);
}
```

Exemple

voir Exemple sur https://dartpad.dartlang.org/a0e092983f63a40b0b716989cac6969a

Getters et Setters

```
void main() {
  var cat = new Cat();

  print("Is cat hungry? ${cat.isHungry}");  // Is cat hungry? true
  print("Is cat cuddly? ${cat.isCuddly}");  // Is cat cuddly? false
  print("Feed cat.");
```

```
  cat.isHungry = false;
  print("Is cat hungry? ${cat.isHungry}");  // Is cat hungry? false
  print("Is cat cuddly? ${cat.isCuddly}");  // Is cat cuddly? true
}

class Cat {
  bool _isHungry = true;

  bool get isCuddly => !_isHungry;

  bool get isHungry => _isHungry;
  bool set isHungry(bool hungry) => this._isHungry = hungry;
}
```

Les getters et setters de classe Dart permettent aux API d'encapsuler les changements d'état des objets.

Voir l'exemple de dartpad ici: https://dartpad.dartlang.org/c25af60ca18a192b84af6990f3313233

Lire Démarrer avec Dart en ligne: https://riptutorial.com/fr/dart/topic/843/demarrer-avec-dart

Chapitre 2: Bibliothèques

Remarques

Les directives d' `import` et de `library` peuvent vous aider à créer une base de code modulable et partageable. Chaque application Dart est une `library` , même si elle n'utilise pas de directive de bibliothèque. Les bibliothèques peuvent être distribuées à l'aide de packages. Pour plus d'informations sur pub, un gestionnaire de packages inclus dans le SDK, voir Package Package et Asset Manager .

Examples

Utiliser les librairies

Utilisez `import` pour spécifier comment un espace de noms d'une bibliothèque est utilisé dans le cadre d'une autre bibliothèque.

```
import 'dart:html';
```

Le seul argument requis pour `import` est un URI spécifiant la bibliothèque. Pour les bibliothèques intégrées, l'URI a le schéma spécial `dart:` :. Pour les autres bibliothèques, vous pouvez utiliser un chemin de système de fichiers ou le `package:` scheme. Le `package:` scheme spécifie les bibliothèques fournies par un gestionnaire de packages tel que l'outil pub. Par exemple:

```
import 'dart:io';
import 'package:mylib/mylib.dart';
import 'package:utils/utils.dart';
```

Bibliothèques et visibilité

Contrairement à Java, Dart n'a pas les mots clés `public` , `protected` et private. Si un identifiant commence par un trait de soulignement _ , il est privé à sa bibliothèque.

Si, par exemple, vous avez la classe A dans un fichier de bibliothèque distinct (par exemple, `other.dart`), par exemple:

```
library other;

class A {
  int _private = 0;

  testA() {
    print('int value: $_private');  // 0
    _private = 5;
    print('int value: $_private'); // 5
  }
}
```

puis importez-le dans votre application principale, par exemple:

```
import 'other.dart';

void main() {
  var b = new B();
  b.testB();
}

class B extends A {
  String _private;

  testB() {
    _private = 'Hello';
    print('String value: $_private'); // Hello
    testA();
    print('String value: $_private'); // Hello
  }
}
```

Vous obtenez le résultat attendu:

```
String value: Hello
int value: 0
int value: 5
String value: Hello
```

Spécifier un préfixe de bibliothèque

Si vous importez deux bibliothèques ayant des identificateurs en conflit, vous pouvez spécifier un préfixe pour une ou les deux bibliothèques. Par exemple, si library1 et library2 ont tous deux une classe Element, vous pourriez avoir un code comme celui-ci:

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;
// ...
var element1 = new Element(); // Uses Element from lib1.
var element2 =
    new lib2.Element();       // Uses Element from lib2.
```

Importer seulement une partie d'une bibliothèque

Si vous souhaitez utiliser uniquement une partie d'une bibliothèque, vous pouvez importer la bibliothèque de manière sélective. Par exemple:

```
// Import only foo and bar.
import 'package:lib1/lib1.dart' show foo, bar;

// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

Charger une bibliothèque

Le chargement différé (également appelé chargement différé) permet à une application de charger une bibliothèque à la demande, si et quand cela est nécessaire. Pour charger une bibliothèque paresseusement, vous devez d'abord l'importer en différé comme.

```
import 'package:deferred/hello.dart' deferred as hello;
```

Lorsque vous avez besoin de la bibliothèque, appelez loadLibrary () en utilisant l'identifiant de la bibliothèque.

```
greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

Dans le code précédent, l' `await` mot - clé interrompt l' exécution jusqu'à ce que la bibliothèque est chargée. Pour plus d'informations sur l' `async` et l' `await` , consultez d'autres exemples ici: prise en charge asynchrone ou visitez la partie support asynchrone de la visite linguistique.

Lire Bibliothèques en ligne: https://riptutorial.com/fr/dart/topic/3332/bibliotheques

Chapitre 3: Collections

Examples

Créer une nouvelle liste

Les listes peuvent être créées de plusieurs manières.

La méthode recommandée consiste à utiliser un littéral de `List` :

```
var vegetables = ['broccoli', 'cabbage'];
```

Le constructeur de `List` peut également être utilisé:

```
var fruits = new List();
```

Si vous préférez un typage plus fort, vous pouvez également fournir un paramètre de type de l'une des manières suivantes:

```
var fruits = <String>['apples', 'oranges'];
var fruits = new List<String>();
```

Pour créer une petite liste pouvant être développée, vide ou contenant des valeurs initiales connues, la forme littérale est préférable. Il existe des constructeurs spécialisés pour d'autres types de listes:

```
var fixedLengthList1 = new List(8);
var fixedLengthList2 = new List.filled(8, "initial text");
var computedValues = new List.generate(8, (n) => "x" * n);
var fromIterable = new List<String>.from(computedValues.getRange(2, 5));
```

Voir aussi le guide de style Dart efficace sur les collections .

Créer un nouvel ensemble

Les ensembles peuvent être créés via le constructeur:

```
var ingredients = new Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);
```

Créer une nouvelle carte

Les cartes peuvent être créées de plusieurs manières.

En utilisant le constructeur, vous pouvez créer une nouvelle carte comme suit:

```
var searchTerms = new Map();
```

Les types pour la clé et la valeur peuvent également être définis en utilisant des génériques:

```
var nobleGases = new Map<int, String>();
var nobleGases = <int, String>{};
```

Les cartes peuvent être créées en utilisant le littéral de la carte:

```
var map = {
    "key1": "value1",
    "key2": "value2"
};
```

Mappez chaque élément de la collection.

Tous les objets de collection contiennent une méthode de `map` qui prend une `Function` comme argument, qui doit prendre un seul argument. Cela retourne une `Iterable` soutenue par la collection. Lorsque l' `Iterable` est itéré, chaque étape appelle la fonction avec un nouvel élément de la collection, et le résultat de l'appel devient l'élément suivant de l'itération.

Vous pouvez `Iterable` une `Iterable` en une collection en utilisant les méthodes `Iterable.toSet()` ou `Iterable.toList()` ou en utilisant un constructeur de collection qui utilise une `Queue.from` comme `Queue.from` ou `List.from` .

Exemple:

```
main() {
  var cats = [
    'Abyssinian',
    'Scottish Fold',
    'Domestic Shorthair'
  ];

  print(cats); // [Abyssinian, Scottish Fold, Domestic Shorthair]

  var catsInReverse =
  cats.map((String cat) {
    return new String.fromCharCodes(cat.codeUnits.reversed);
  })
  .toList(); // [nainissybA, dloF hsittocS, riahtrohS citsemoD]

  print(catsInReverse);
}
```

Voir l'exemple de dartpad ici: https://dartpad.dartlang.org/a18367ff767f172b34ff03c7008a6fa1

Filtrer une liste

Dart permet de filtrer facilement une liste en utilisant `where` .

```
var fruits = ['apples', 'oranges', 'bananas'];
```

```
fruits.where((f) => f.startsWith('a')).toList(); //apples
```

Bien sûr, vous pouvez utiliser des opérateurs AND ou OR dans votre clause where.

Lire Collections en ligne: https://riptutorial.com/fr/dart/topic/859/collections

Chapitre 4: commentaires

Syntaxe

- // Commentaire sur une seule ligne
- / * Multi-line / In-line comment * /
- /// Commentaire Dartdoc

Remarques

Il est conseillé d'ajouter des commentaires à votre code pour expliquer pourquoi quelque chose est fait ou pour expliquer ce que fait quelque chose. Cela aide les futurs lecteurs de votre code à mieux comprendre votre code.

Rubrique (s) associée (s) non sur StackOverflow:

- Dart effectif: Documentation

Examples

Commentaire de fin de ligne

Tout à droite de `//` dans la même ligne est commenté.

```
int i = 0; // Commented out text
```

Commentaire multi-lignes

Tout entre `/*` et `*/` est commenté.

```
void main() {
  for (int i = 0; i < 5; i++) {
    /* This is commented, and
    will not affect code */
    print('hello ${i + 1}');
  }
}
```

Documentation utilisant Dartdoc

L'utilisation d'un commentaire doc au lieu d'un commentaire régulier permet à dartdoc de le trouver et de générer la documentation correspondante.

```
/// The number of characters in this chunk when unsplit.
int get length => ...
```

Vous êtes autorisé à utiliser la plupart des formats de démarques dans vos commentaires doc et dartdoc les traitera en conséquence en utilisant le package markdown .

```
/// This is a paragraph of regular text.
///
/// This sentence has *two* _emphasized_ words (i.e. italics) and **two**
/// __strong__ ones (bold).
///
/// A blank line creates another separate paragraph. It has some `inline code`
/// delimited using backticks.
///
/// * Unordered lists.
/// * Look like ASCII bullet lists.
/// * You can also use `-` or `+`.
///
/// Links can be:
///
/// * http://www.just-a-bare-url.com
/// * [with the URL inline](http://google.com)
/// * [or separated out][ref link]
///
/// [ref link]: http://google.com
///
/// # A Header
///
/// ## A subheader
```

Lire commentaires en ligne: https://riptutorial.com/fr/dart/topic/2436/commentaires

Chapitre 5: Conversion de données

Examples

JSON

```
import 'dart:convert';

void main() {
    var jsonString = """
      {
        "cats": {
                "abysinnian": {
                    "origin": "Burma",
                    "behavior": "playful"
                }
            }
    }
  """;

  var obj = JSON.decode(jsonString);

  print(obj['cats']['abysinnian']['behavior']); // playful
}
```

Voir exemple sur dartpad: https://dartpad.dartlang.org/7d5958cf10e611b36326f27b062108fe

Lire Conversion de données en ligne: https://riptutorial.com/fr/dart/topic/2778/conversion-de-donnees

Chapitre 6: Cordes

Examples

Concaténation et interpolation

Vous pouvez utiliser l'opérateur plus (`+`) pour concaténer des chaînes:

```
'Dart ' + 'is ' + 'fun!'; // 'Dart is fun!'
```

Vous pouvez également utiliser des littéraux de chaîne adjacents pour la concaténation:

```
'Dart ' 'is ' 'fun!';    // 'Dart is fun!'
```

Vous pouvez utiliser `${}` pour interpoler la valeur des expressions Dart dans les chaînes. Les accolades peuvent être omises lors de l'évaluation des identificateurs:

```
var text = 'dartlang';
'$text has ${text.length} letters'; // 'dartlang has 8 letters'
```

Chaînes valides

Une chaîne peut être simple ou multiligne. Les chaînes simples sont écrites à l'aide de guillemets simples ou doubles, et les chaînes multilignes sont écrites à l'aide de guillemets. Toutes les chaînes Dart sont les suivantes:

```
'Single quotes';
"Double quotes";
'Double quotes in "single" quotes';
"Single quotes in 'double' quotes";

'''A
multiline
string''';

"""
Another
multiline
string""";
```

Bâtiment à partir de pièces

Générer une chaîne par programme est mieux réalisé avec un StringBuffer . Un StringBuffer ne génère pas un nouvel objet String tant que `toString()` n'est pas appelé.

```
var sb = new StringBuffer();

sb.write("Use a StringBuffer");
```

```
sb.writeAll(["for ", "efficient ", "string ", "creation "]);
sb.write("if you are ")
sb.write("building lots of strings");

// or you can use method cascades:

sb
  ..write("Use a StringBuffer")
  ..writeAll(["for ", "efficient ", "string ", "creation "])
  ..write("if you are ")
  ..write("building lots of strings");

var fullString = sb.toString();

print(fullString);
// Use a StringBufferfor efficient string creation if you are building lots of strings

sb.clear(); // all gone!
```

Lire Cordes en ligne: https://riptutorial.com/fr/dart/topic/5003/cordes

Chapitre 7: Date et l'heure

Examples

Utilisation de base de DateTime

```
DateTime now = new DateTime.now();
DateTime berlinWallFell = new DateTime(1989, 11, 9);
DateTime moonLanding = DateTime.parse("1969-07-20 20:18:00");  // 8:18pm
```

Vous pouvez trouver plus d'informations détaillées ici .

Lire Date et l'heure en ligne: https://riptutorial.com/fr/dart/topic/3322/date-et-l-heure

Chapitre 8: Des classes

Examples

Créer une classe

Les classes peuvent être créées comme suit:

```
class InputField {
  int maxLength;
  String name;
}
```

La classe peut être instanciée à l'aide du `new` mot clé, après quoi les valeurs du champ seront nulles par défaut.

```
var field = new InputField();
```

Les valeurs de champ sont alors accessibles:

```
// this will trigger the setter
field.name = "fieldname";

// this will trigger the getter
print(field.name);
```

Membres

Une classe peut avoir des membres.

Les variables d'instance peuvent être déclarées avec / sans annotation de type et éventuellement initialisées. Les membres non initialisés ont la valeur `null` , à moins d'être définis sur une autre valeur par le constructeur.

```
class Foo {
  var member1;
  int member2;
  String member3 = "Hello world!";
}
```

Les variables de classe sont déclarées à l'aide du mot-clé `static` .

```
class Bar {
  static var member4;
  static String member5;
  static int member6 = 42;
}
```

Si une méthode ne prend aucun argument, est rapide, renvoie une valeur et n'a pas d'effets secondaires visibles, une méthode de lecture peut être utilisée:

```
class Foo {
  String get bar {
    var result;
    // ...
    return result;
  }
}
```

Les getters ne prennent jamais d'arguments, donc les parenthèses pour la liste de paramètres (vide) sont omises à la fois pour déclarer les getters, comme ci-dessus, et pour les appeler, comme ceci:

```
main() {
  var foo = new Foo();
  print(foo.bar); // prints "bar"
}
```

Il existe également des méthodes de réglage, qui doivent prendre exactement un argument:

```
class Foo {
  String _bar;

  String get bar => _bar;

  void set bar(String value) {
    _bar = value;
  }
}
```

La syntaxe pour appeler un setter est identique à celle de la variable:

```
main() {
  var foo = new Foo();
  foo.bar = "this is calling a setter method";
}
```

Constructeurs

Un constructeur de classe doit avoir le même nom que sa classe.

Créons un constructeur pour une classe Person:

```
class Person {
  String name;
  String gender;
  int age;

  Person(this.name, this.gender, this.age);
}
```

L’exemple ci-dessus est une manière plus simple et plus efficace de définir le constructeur de la manière suivante, qui est également possible:

```
class Person {
  String name;
  String gender;
  int age;

  Person(String name, String gender, int age) {
    this.name = name;
    this.gender = gender;
    this.age = age;
  }
}
```

Maintenant, vous pouvez créer une instance de Person comme ceci:

```
var alice = new Person('Alice', 'female', 21);
```

Lire Des classes en ligne: https://riptutorial.com/fr/dart/topic/1511/des-classes

Chapitre 9: Des exceptions

Remarques

Le code Dart peut lancer et intercepter des exceptions. Les exceptions sont des erreurs indiquant que quelque chose d'inattendu s'est produit. Si l'exception n'est pas interceptée, l'isolat qui a déclenché l'exception est suspendu et l'isolat et son programme sont généralement terminés.

Contrairement à Java, toutes les exceptions de Dart sont des exceptions non vérifiées. Les méthodes ne déclarent pas les exceptions qu'elles peuvent émettre et vous n'êtes pas obligé de détecter des exceptions.

Dart fournit des types d' exception et d' erreur , ainsi que de nombreux sous-types prédéfinis. Vous pouvez bien entendu définir vos propres exceptions. Toutefois, les programmes Dart peuvent lancer tout objet non nul, pas seulement les objets Exception et Error, comme exception.

Examples

Exception personnalisée

```
class CustomException implements Exception {
  String cause;
  CustomException(this.cause);
}

void main() {
  try {
    throwException();
  } on CustomException {
    print("custom exception is been obtained");
  }
}

throwException() {
  throw new CustomException('This is my first custom exception');
}
```

Lire Des exceptions en ligne: https://riptutorial.com/fr/dart/topic/3334/des-exceptions

Chapitre 10: Enums

Examples

Utilisation de base

```
enum Fruit {
  apple, banana
}

main() {
  var a = Fruit.apple;
  switch (a) {
    case Fruit.apple:
      print('it is an apple');
      break;
  }

  // get all the values of the enums
  for (List<Fruit> value in Fruit.values) {
    print(value);
  }

  // get the second value
  print(Fruit.values[1]);
}
```

Lire Enums en ligne: https://riptutorial.com/fr/dart/topic/5107/enums

Chapitre 11: Expressions régulières

Syntaxe

- var regExp = RegExp (r '^ (. *) $', multiLine: true, caseSensitive: false);

Paramètres

Paramètre	Détails
`String source`	L'expression régulière en tant que `String`
`{bool multiline}`	S'il s'agit d'une expression régulière multiligne. (correspond à `^` et `$` au début et à la fin de chaque ligne individuellement, pas la chaîne entière)
`{bool caseSensitive}`	Si l'expression est sensible à la casse

Remarques

Les expressions régulières Dart ont la même syntaxe et la même sémantique que les expressions régulières JavaScript. Voir http://ecma-international.org/ecma-262/5.1/#sec-15.10 pour la spécification des expressions régulières JavaScript.

Cela signifie que toute ressource JavaScript que vous trouvez à propos des expressions régulières en ligne s'applique à dart.

Examples

Créer et utiliser une expression régulière

```
var regExp = new RegExp(r"(\w+)");
var str = "Parse my string";
Iterable<Match> matches = regExp.allMatches(str);
```

C'est une bonne idée d'utiliser des "chaînes brutes" (préfixe avec `r`) lors de l'écriture des expressions régulières afin que vous puissiez utiliser des barres obliques inversées dans votre expression.

Lire Expressions régulières en ligne: https://riptutorial.com/fr/dart/topic/3624/expressions-regulieres

Chapitre 12: Filtres de liste

Introduction

Dart filtre les listes via les méthodes `List.where` et `List.retainWhere` . La fonction `where` prend un argument: une fonction booléenne qui est appliquée à chaque élément de la liste. Si la fonction est évaluée à `true` l'élément de liste est conservé. Si la fonction est évaluée à `false` , l'élément est supprimé.

L'appel de `theList.retainWhere(foo)` équivaut pratiquement à la définition de `theList = theList.where(foo)` .

Examples

Filtrage d'une liste d'entiers

```
[-1, 0, 2, 4, 7, 9].where((x) => x > 2) --> [4, 7, 9]
```

Lire Filtres de liste en ligne: https://riptutorial.com/fr/dart/topic/10948/filtres-de-liste

Chapitre 13: Flux de contrôle

Examples

Sinon

Dart a Si Else:

```
if (year >= 2001) {
  print('21st century');
} else if (year >= 1901) {
  print('20th century');
} else {
  print('We Must Go Back!');
}
```

Dart a également un opérateur ternaire `if` :

```
var foo = true;
print(foo ? 'Foo' : 'Bar'); // Displays "Foo".
```

En boucle

Les boucles while et do while sont autorisées dans Dart:

```
while(peopleAreClapping()) {
  playSongs();
}
```

et:

```
do {
  processRequest();
} while(stillRunning());
```

Les boucles peuvent être terminées en utilisant une pause:

```
while (true) {
  if (shutDownRequested()) break;
  processIncomingRequests();
}
```

Vous pouvez ignorer les itérations dans une boucle en utilisant continue:

```
for (var i = 0; i < bigNumber; i++) {
  if (i.isEven){
    continue;
  }
  doSomething();
```

```
}
```

Pour boucle

Deux types de boucles for sont autorisés:

```
for (int month = 1; month <= 12; month++) {
  print(month);
}
```

et:

```
for (var object in flybyObjects) {
  print(object);
}
```

La boucle `for-in` est pratique lorsque vous parcourez simplement une collection `Iterable` . Il existe également une méthode `forEach` laquelle vous pouvez appeler des objets `Iterable` qui se comportent comme des `for-in` :

```
flybyObjects.forEach((object) => print(object));
```

ou plus concis:

```
flybyObjects.forEach(print);
```

Boîtier de commutation

Dart dispose d'un commutateur qui peut être utilisé à la place des instructions if-else longues:

```
var command = 'OPEN';

switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'OPEN':
    executeOpen();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'UNSURE':
    // missing break statement means this case will fall through
    // to the next statement, in this case the default case
  default:
    executeUnknown();
}
```

Vous ne pouvez comparer que des constantes de type entier, chaîne ou à la compilation. Les objets comparés doivent être des instances de la même classe (et non de l'un de ses sous-types),

et la classe ne doit pas remplacer ==.

Un aspect surprenant de la commutation dans Dart est que les clauses non vides doivent se terminer par une rupture, ou moins fréquemment, continuer, lancer ou retourner. C'est-à-dire que les clauses non vides ne peuvent pas tomber. Vous devez explicitement terminer une clause non vide, généralement avec une interruption. Vous obtiendrez un avertissement statique si vous omettez de rompre, de continuer, de lancer ou de renvoyer, et le code sera erroné à cet emplacement lors de l'exécution.

```
var command = 'OPEN';
switch (command) {
  case 'OPEN':
    executeOpen();
    // ERROR: Missing break causes an exception to be thrown!!

  case 'CLOSED': // Empty case falls through
  case 'LOCKED':
    executeClosed();
    break;
}
```

Si vous voulez fall-through dans un non vide `case` , vous pouvez utiliser `continue` et une étiquette:

```
      var command = 'OPEN';
      switch (command) {
        case 'OPEN':
          executeOpen();
          continue locked;
locked: case 'LOCKED':
          executeClosed();
          break;
      }
```

Lire Flux de contrôle en ligne: https://riptutorial.com/fr/dart/topic/923/flux-de-controle

Chapitre 14: Interopérabilité Dart-JavaScript

Introduction

L'interopérabilité Dart-JavaScript nous permet d'exécuter du code JavaScript à partir de nos programmes Dart.

L'interopérabilité est réalisée en utilisant la bibliothèque `js` pour créer des stubs Dart. Ces stubs décrivent l'interface que nous aimerions avoir avec le code JavaScript sous-jacent. Au moment de l'exécution, l'appel du stub Dart invoquera le code JavaScript.

Examples

Appeler une fonction globale

Supposons que nous souhaitons appeler la fonction JavaScript `JSON.stringify` qui reçoit un objet, la code dans une chaîne JSON et la renvoie.

Il suffit d'écrire la signature de la fonction, de la marquer comme externe et de l'annoter avec l'annotation `@JS` :

```
 @JS("JSON.stringify")
external String stringify(obj);
```

L'annotation `@JS` sera utilisée à partir de `@JS` pour marquer les classes Dart que nous aimerions utiliser également dans JavaScript.

Emballage de classes / espaces de noms JavaScript

Supposons que nous souhaitons `google.maps` API JavaScript Google Maps `google.maps` :

```
@JS('google.maps')
library maps;

import "package:js/js.dart";

@JS()
class Map {
  external Map(Location location);
  external Location getLocation();
}
```

Nous avons maintenant la classe Map Dart qui correspond à la classe JavaScript `google.maps.Map` .

L'exécution d'une `new Map(someLocation)` dans Dart lancera un `new google.maps.Map(location)` dans JavaScript.

Notez que vous n'avez pas à nommer votre classe Dart comme la classe JavaScript:

```
@JS("LatLng")
class Location {
  external Location(num lat, num lng);
}
```

La classe `Location` Dart correspond à la classe `google.maps.LatLng` .

L'utilisation de noms incohérents est déconseillée car elle peut créer de la confusion.

Littéraux d'objet en passant

Il est courant en JavaScript de transmettre des littéraux d'objet à des fonctions:

```
// JavaScript
printOptions({responsive: true});
Unfortunately we cannot pass Dart Map objects to JavaScript in these cases.
```

Ce que nous devons faire, c'est créer un objet Dart qui représente le littéral d'objet et contient tous ses champs:

```
// Dart
@JS()
@anonymous
class Options {
  external bool get responsive;

  external factory Options({bool responsive});
}
```

Notez que la classe `Options` Dart ne correspond à aucune classe JavaScript. En tant que tel, nous devons le marquer avec l'annotation `@anonymous` .

Maintenant, nous pouvons créer un stub pour la fonction printOptions originale et l'appeler avec un nouvel objet Options:

```
// Dart
@JS()
external printOptions(Options options);

printOptions(new Options(responsive: true));
```

Lire Interopérabilité Dart-JavaScript en ligne:
https://riptutorial.com/fr/dart/topic/9240/interoperabilite-dart-javascript

Chapitre 15: Les fonctions

Remarques

Dart est un véritable langage orienté objet, donc même les fonctions sont des objets et ont un type, Fonction. Cela signifie que les fonctions peuvent être affectées à des variables ou transmises comme arguments à d'autres fonctions. Vous pouvez également appeler une instance d'une classe Dart comme s'il s'agissait d'une fonction.

Examples

Fonctions avec paramètres nommés

Lors de la définition d'une fonction, utilisez {param1, param2,…} pour spécifier les paramètres nommés:

```
void enableFlags({bool bold, bool hidden}) {
  // ...
}
```

Lors de l'appel d'une fonction, vous pouvez spécifier des paramètres nommés à l'aide de paramName: value

```
enableFlags(bold: true, hidden: false);
```

Fonction scoping

Les fonctions Dart peuvent également être déclarées anonymement ou imbriquées. Par exemple, pour créer une fonction imbriquée, ouvrez simplement un nouveau bloc fonction dans un bloc fonction existant.

```
void outerFunction() {

    bool innerFunction() {
        /// Does stuff
    }
}
```

La fonction `innerFunction` peut maintenant être utilisée à l'intérieur, et uniquement à l'intérieur, `outerFunction` . Aucune autre fonction n'y a accès.

Les fonctions dans Dart peuvent également être déclarées de manière anonyme, ce qui est couramment utilisé comme argument de fonction. Un exemple courant est la méthode de `sort` de l'objet `List` . Cette méthode prend un argument facultatif avec la signature suivante:

```
int compare(E a, E b)
```

La documentation indique que la fonction doit retourner `0` si `a` et `b` sont égaux. Il renvoie `-1` si `a < b` et `1` si `a > b` .

Sachant cela, nous pouvons trier une liste d'entiers en utilisant une fonction anonyme.

```
List<int> numbers = [4,1,3,5,7];

numbers.sort((int a, int b) {
   if(a == b) {
      return 0;
   } else if (a < b) {
      return -1;
   } else {
      return 1;
   }
});
```

La fonction anonyme peut également être liée à des identificateurs tels que:

```
Function intSorter = (int a, int b) {
   if(a == b) {
      return 0;
   } else if (a < b) {
      return -1;
   } else {
      return 1;
   }
}
```

et utilisé comme une variable ordinaire.

```
numbers.sort(intSorter);
```

Lire Les fonctions en ligne: https://riptutorial.com/fr/dart/topic/2965/les-fonctions

Chapitre 16: Programmation asynchrone

Examples

Retourner un avenir en utilisant un Completer

```
Future<Results> costlyQuery() {
  var completer = new Completer();

  database.query("SELECT * FROM giant_table", (results) {
    // when complete
    completer.complete(results);
  }, (error) {
    completer.completeException(error);
  });

  // this returns essentially immediately,
  // before query is finished
  return completer.future;
}
```

Async et attend

```
import 'dart:async';

Future main() async {
  var value = await _waitForValue();
  print("Here is the value: $value");
  //since _waitForValue() returns immediately if you un it without await you won't get the
result
  var errorValue = "not finished yet";
  _waitForValue();
  print("Here is the error value: $value");// not finished yet
}

Future<int> _waitForValue() => new Future((){

  var n = 100000000;

  // Do some long process
  for (var i = 1; i <= n; i++) {
    // Print out progress:
    if ([n / 2, n / 4, n / 10, n / 20].contains(i)) {
      print("Not done yet...");
    }

    // Return value when done.
    if (i == n) {
      print("Done.");
      return i;
    }
  }
});
```

Voir exemple sur Dartpad: https://dartpad.dartlang.org/11d189b51e0f2680793ab3e16e53613c

Conversion de rappels en contrats à terme

Dart possède une bibliothèque asynchrone robuste, avec Future , Stream , etc. Cependant, parfois , vous croiserez peut - être une API asynchrone qui utilise *callbacks* au lieu de *contrats* à *terme.* Pour combler le fossé entre les rappels et les contrats à terme, Dart propose la classe *Completer* . Vous pouvez utiliser un Completer pour convertir un rappel en Future.

Completers sont parfaits pour relier une API basée sur le rappel avec une API basée sur l'avenir. Par exemple, supposons que votre pilote de base de données n'utilise pas Futures, mais vous devez renvoyer un futur. Essayez ce code:

```
// A good use of a Completer.

Future doStuff() {
  Completer completer = new Completer();
  runDatabaseQuery(sql, (results) {
    completer.complete(results);
  });
  return completer.future;
}
```

Si vous utilisez une API qui renvoie déjà un futur, vous n'avez pas besoin d'utiliser un Completer.

Lire Programmation asynchrone en ligne: https://riptutorial.com/fr/dart/topic/2520/programmation-asynchrone

Chapitre 17: Pub

Remarques

Lorsque vous installez le Dart SDK, l'un des outils que vous obtenez est pub. L'outil pub fournit des commandes à des fins diverses. Une commande installe des packages, un autre démarre un serveur HTTP pour le test, un autre prépare votre application pour le déploiement et un autre publie votre package sur pub.dartlang.org . Vous pouvez accéder aux commandes pub via un IDE, tel que WebStorm, ou sur la ligne de commande.

Pour un aperçu de ces commandes, voir Commandes de publication .

Examples

construction de pub

Utilisez la génération de pub lorsque vous êtes prêt à déployer votre application Web. Lorsque vous exécutez une génération de pub, elle génère les actifs du package actuel et de toutes ses dépendances, en les plaçant dans le nouveau répertoire nommé build.

Pour utiliser la `pub build` , exécutez-la simplement dans le répertoire racine de votre package. Par exemple:

```
$ cd ~/dart/helloworld
$ pub build
Building helloworld......
Built 5 files!
```

pub servir

Cette commande démarre un serveur de développement ou un serveur de développement pour votre application Web Dart. Le serveur dev est un serveur HTTP sur localhost qui sert les ressources de votre application Web.

Démarrez le serveur de développement à partir du répertoire contenant le fichier `pubspec.yaml` de votre application Web:

```
$ cd ~/dart/helloworld
$ pub serve
Serving helloworld on http://localhost:8080
```

Lire Pub en ligne: https://riptutorial.com/fr/dart/topic/3335/pub

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Dart	4444, Challe, Community, Damon, Florian Loitsch, Gomiero, Kleak, losnake, martin, Raph, Timothy C. Quinn
2	Bibliothèques	Challe, Ganymede
3	Collections	Alexi Coard, Damon, Jan Vladimir Mostert, Kleak, lrn, Pacane, Raph
4	commentaires	Challe
5	Conversion de données	Damon
6	Cordes	Challe
7	Date et l'heure	Challe
8	Des classes	Ganymede, Hoylen, Jan Vladimir Mostert, Raph
9	Des exceptions	Challe
10	Enums	Challe
11	Expressions régulières	enyo
12	Filtres de liste	jxmorris12
13	Flux de contrôle	Ganymede, Jan Vladimir Mostert, Pacane, Raph
14	Interopérabilité Dart-JavaScript	Meshulam Silk
15	Les fonctions	Jan Vladimir Mostert, Kim Rostgaard Christensen
16	Programmation asynchrone	Challe, Damon, Ray Hulha, Zied Hamdi
17	Pub	Challe