Web Development

JavaScript

January 7, 2022

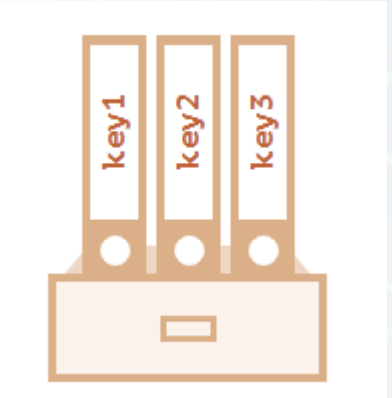
Data Types

There are 8 basic data types in JavaScript.

- number for numbers of any kind: integer or floating-point, integers are limited by ±(2^53-1).
- bigint is for integer numbers of arbitrary length.
- string for strings. A string may have zero or more characters, there's no separate single-character type.
- boolean for true/false.
- null for unknown values – a standalone type that has a single value null.
- undefined for unassigned values – a standalone type that has a single value undefined.
- object for more complex data structures.
- symbol for unique identifiers.

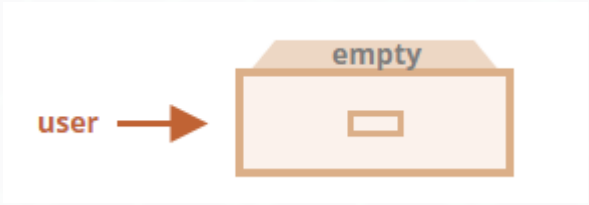
Objects

- Objects are used to store keyed collections of various data and more complex entities.

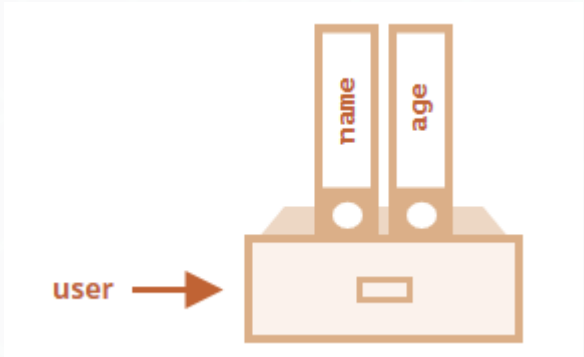


- An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax
```




```
let user = {     // an object
  name: "John",  // by key "name" store value "John"
  age: 30        // by key "age" store value 30
};
```




```
let user = {};

// set
user["likes birds"] = true;

// get
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];
```

Arrays

- There are two syntaxes for creating an empty array:
  - let arr = new Array();
  - let arr = [];

We can get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum


// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );
}
```

Class

- The basic syntax is:

```
class MyClass {
 // class methods
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

```
class User {

  constructor(name) {
    this.name = name;
  }

  sayHi() {
    alert(this.name);
  }

}

// Usage:
let user = new User("John");
user.sayHi();
```

Callbacks (1 of 2)

- A callback is a function passed as an argument to another function
- This technique allows a function to call another function
- A callback function can run after another function has finished
- When to Use a Callback?

Where callbacks really shine are in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

```
function myDisplayer(some) {

document.getElementById("demo").i
nnerHTML = some;
}

function myCalculator(num1, num2,
myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

Callbacks (2 of 2)

- Many functions are provided by JavaScript host environments that allow you to schedule asynchronous actions. In other words, actions that we initiate now, but they finish later.

  For instance, one such function is the **setTimeout** function.

- There are other real-world examples of asynchronous actions, e.g. loading scripts and modules (we'll cover them in later chapters).

- Take a look at the function loadScript(src), that loads a script with the given src:

```
function loadScript(src) {
  // creates a <script> tag and append it to the page
  // this causes the script with given src to start loading and run when complete
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Asynchronous JavaScript

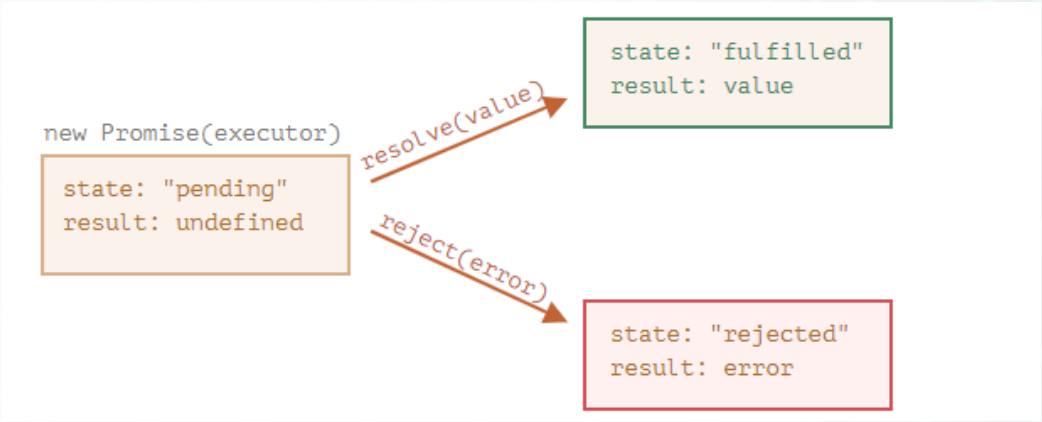
- "I will finish later!"
- Functions running in parallel with other functions are called asynchronous
- A good example is JavaScript setTimeout()
- setInterval()
- If you create a function to load an external resource (like a script or a file), you cannot use the content before it is fully loaded.
- This is the perfect time to use a callback.

```
setTimeout(myFunction, 3000);

function myFunction() {

        document.getElementById("demo").inne
        rHTML = “Good!!";
}
```

Promise (1 of 2)

- The constructor syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```



Its arguments resolve and reject are callbacks provided by JavaScript itself. Our code is only inside the executor.

Consumers: then, catch

```
promise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Promise (2 of 2)

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1
second
);
```

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1
second
```

Async/Await (1 of 3)

- There's a special syntax to work with promises in a more comfortable fashion, called "async/await". It's surprisingly easy to understand and use.
- async makes a function return a Promise
- await makes a function wait for a Promise

- Async functions

Let's start with the async keyword.

It can be placed before a function, like this:

```
async function f() {
  return 1;
}
```

```
// works only inside async functions
let value = await promise;
```

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

Async/Await (2 of 3)

- Async Syntax

```
async function myFunction() {
  return "Hello";
}
```

Is the same as:

```
function myFunction() {
  return Promise.resolve("Hello");
}
```

- Here is how to use the Promise:

```
myFunction().then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Async/Await (3 of 3)

- The keyword await before a function makes the function wait for a promise:
- `let value = await promise;`
- The await keyword can only be used inside an async function.

```
async function myDisplay() {
  let myPromise = new Promise(function(resolve) {
    setTimeout(function() {resolve("Done!!");}, 3000);
  });
  document.getElementById("demo").innerHTML = await myPromise;
}

myDisplay();
```

Modules

- As our application grows bigger, we want to split it into multiple files, so called “modules”. A module may contain a class or a library of functions for a specific purpose.
- For a long time, JavaScript existed without a language-level module syntax. That wasn’t a problem, because initially scripts were small and simple, so there was no need.
- But eventually scripts became more and more complex, so the community invented a variety of ways to organize code into modules, special libraries to load modules on demand.
- export keyword labels variables and functions that should be accessible from outside the current module.
- import allows the import of functionality from other modules.

...Then another file may import and use it:

```
// 🗀 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

```
// 🗀 sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

Links

- https://javascript.info/
- https://caolan.github.io/async/v3/