JavaScript Checklist

1. Stack & Heap

JavaScript engines have two places to store data: Stack & Heap.

Stack	Heap
Primitive data types and references	Objects and functions
Size is known at compile time	Size is known at run time
Fixed memory allocated	No limit for object memory

Memory management in JavaScript

Memory Life Cycle: Irrespective of programming language, the memory life cycle follows the following stages:

1. **Allocates the memory we need:** JavaScript allocates memory to the object created.
2. **Use the allocated memory.**
3. **Release the memory when not in use:** Once the allocated memory is released, it is used for other purposes. It is handled by a JavaScript engine.

Note: The second stage is the same for all the languages. However, the first and last stage is implicit in high-level languages like JavaScript.

JavaScript engines have two places to store data:

- **Stack:** It is a data structure used to store static data. Static data refers to data whose size is known by the engine during compile time. In JavaScript, static data includes primitive values like strings, numbers, boolean, null, and undefined. References that point to objects and functions are also included. A fixed amount of memory is allocated for static data. This process is known as **static memory allocation.**
- **Heap:** It is used to store objects and functions in JavaScript. The engine doesn't allocate a fixed amount of memory. Instead, it allocates more space as required.

Overview:

Stack	Heap
Primitive data types and references	Objects and functions
Size is known at compile time	Size is known at run time
Fixed memory allocated	No limit for object memory

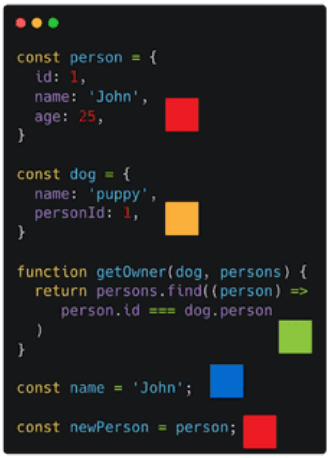
Garbage Collection: Garbage collectors are used in releasing memory. Once the engine recognizes that a variable, object, or function is not needed anymore, it releases the memory it occupied. The main issue here is that it is very difficult to predict accurately whether a particular variable, object, or function is needed anymore or not. Some algorithms help to find the moment when they become obsolete with great precision.

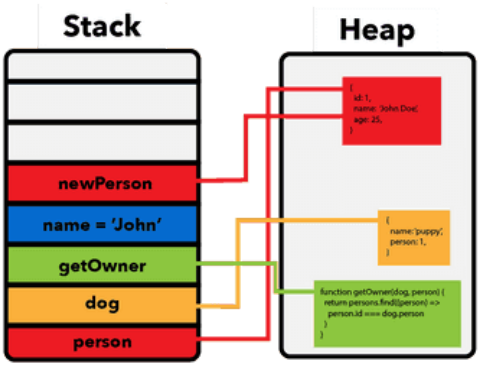
Reference-counting garbage collection: It frees the memory allocated to the objects that have no references pointing to them. However, the problem with this algorithm is that it doesn't understand cyclic references.

```
1. Stack
/*
    Memory Life Cycle: Allocate => Use => Release
    - Stack: + Static memory allocation - Cấp phát bộ nhớ tĩnh
    - stored primitive values (strings, numbers, booleans, undefined, and null)
*/

const male = true
const name = 'John Doe'
const age = 24
const adult = true

// All the values get stored in the stack since they all contain primitive values.
/*
    Stack:
    male = true
    name = 'John Doe'
    age = 24
    adult = true
*/

/*
    - Static data is data where the engine knows the size at compile time.
    - This includes primitive values (strings, numbers, booleans, undefined, and null)
    and references, which point to objects and functions stored in heap.
    - Since the engine knows that the size won't change, it will allocate a fixed amount of memory for each value.
    - The process of allocating memory right before execution is known as static memory allocation.
    - Because the engine allocates a fixed amount of memory for these values, there is a limit to how large primitive values can be.
    - The limits of these values and the entire stack vary depending on the browser.
*/
```

```
2. Heap
/*
    Memory Life Cycle: Allocate => Use => Release
    - Heap: + Dynamic memory allocation - Cấp phát bộ nhớ động
    - stored value of objects and functions in heap and a refrence point to value in stack.
*/

const person = {
    name: 'John',
    age: 24,
};

const hobbies = ['hiking', 'reading'];

function foo() {
    const a = 1;
    console.log('Function stored in heap!')
}

/*
    Stack:
    - person: reference to "person" object
    - hobbies: reference to "hobbies" array
    - foo: reference to "foo" function

    Heap:
    - {name: 'John', age: 24}
    - ['hiking', 'reading']
    - foo() {
        console.log('Function stored in heap!')
    }
*/

// stack
const a = 1;
const b = 1;

// heap
const c = [1];
const d = [1];

console.log(a === b) // true --- stack
console.log(c === d) // false --- heap
```

Explanation: In the below example object '**person**' is created and object `**newPerson**` **copied from** `**person**` stored in the heap and a reference to 'person - newPerson' in stack.





2. Garbage Collection

> **Definition:** Garbage collectors are used in releasing memory. Once the engine recognizes that a variable, object, or function is not needed anymore, it releases the memory it occupied. The main issue here is that it is very difficult to predict accurately whether a particular variable, object, or function is needed anymore or not. Some algorithms help to find the moment when they become obsolete with great precision.

Memory management

Low-level languages like C, have manual memory management primitives such as `malloc()` and `free()`. In contrast, JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (*garbage collection*). This automaticity is a potential source of confusion: it can give developers the false impression that they don't need to worry about memory management.

Garbage collection

Memory management in JavaScript is performed automatically and invisibly to us. We create primitives, objects, functions… All that takes memory.

What happens when something is not needed any more? How does the JavaScript engine discover it and clean it up?

Garbage collectors are used in releasing memory. Once the engine recognizes that a variable, object, or function is not needed anymore, it releases the memory it occupied. The main issue here is that it is very difficult to predict accurately whether a particular variable, object, or function is needed anymore or not. Some algorithms help to find the moment when they become obsolete with great precision.

Memory life cycle

Memory life cycle: Allocate => Use => Release

Regardless of the programming language, the memory life cycle is pretty much always the same:

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

Note: The second part is explicit in all languages. The first and last parts are explicit in low-level languages but are mostly implicit in high-level languages like JavaScript.

- Garbage collection: Releasing memory: Delete these values do not use
- scavenge and mark-sweep/mark-compact algorithm

Once the JavaScript engine recognizes that a given variable or function is not needed anymore,
it releases the memory it occupied.

1. Bước đánh dấu: thuật toán sẽ duyệt qua tất cả các giá trị có trong khu vực bộ nhớ mà nó quản lý,
bước duyệt này đơn giản chỉ là depth-first search, tìm gặp và đánh dấu.

2. Bước xử lý: sau quá trình duyệt, tất cả những giá trị chưa được đánh dấu, sẽ bị coi là đã "chết",
và sẽ bị xóa bỏ, trả lại bộ nhớ trống (sweep), hoặc gom góp lại để lấy lại các khoảng trống trong bộ nhớ không sử dụng được (compact).

Using values

Using values basically means reading and writing in allocated memory. This can be done by reading or writing the value of a variable or an object property or even passing an argument to a function.

Release when the memory is not needed anymore

The majority of memory management issues occur at this phase. The most difficult aspect of this stage is determining when the allocated memory is no longer needed.

Low-level languages require the developer to manually determine at which point in the program the allocated memory is no longer needed and to release it.

Some high-level languages, such as JavaScript, utilize a form of automatic memory management known as garbage collection (GC). The purpose of a garbage collector is to monitor memory allocation and determine when a block of allocated memory is no longer needed and reclaim it. This automatic process is an approximation since the general problem of determining whether or not a specific piece of memory is still needed is undecidable.

As stated above, the general problem of automatically finding whether some memory "is not needed anymore" is undecidable. As a consequence, garbage collectors implement a restriction of a solution to the general problem. This section will explain the concepts that are necessary for understanding the main garbage collection algorithms and their respective limitations.

References

The main concept that garbage collection algorithms rely on is the concept of *reference*. Within the context of memory management, an object is said to reference another object if the former has access to the latter (either implicitly or explicitly). For instance, a JavaScript object has a reference to its prototype (implicit reference) and to its properties values (explicit reference).

In this context, the notion of an "object" is extended to something broader than regular JavaScript objects and also contain function scopes (or the global lexical scope).

Reference-counting garbage collection

This is the most native garbage collection algorithm. This algorithm reduces the problem from determining whether or not an object is still needed to determining if an object still has any other objects referencing it. An object is said to be "garbage", or collectible if there are zero references pointing to it.

Mark-and-sweep algorithm

This algorithm reduces the definition of "an object is no longer needed" to "an object is unreachable".

This algorithm assumes the knowledge of a set of objects called *roots.* In JavaScript, the root is the global object. Periodically, the garbage collector will start from these roots, find all objects that are referenced from these roots, then all objects referenced from these, etc. Starting from the roots, the garbage collector will thus find all *reachable* objects and collect all non-reachable objects.

This algorithm is an improvement over the previous one since an object having zero references is effectively unreachable. The opposite does not hold true as we have seen with circular references.

Currently, all modern engines ship a mark-and-sweep garbage collector. All improvements made in the field of JavaScript garbage collection (generational/incremental/concurrent/parallel garbage collection) over the last few years are implementation improvements of this algorithm, but not improvements over the garbage collection algorithm itself nor its reduction of the definition of when "an object is no longer needed".

The immediate benefit of this approach is that cycles is no longer a problem. In the first example above, after the function call returns, the two objects are no longer referenced by any resource that is reachable from the global object. Consequently, they will be found unreachable by the garbage collector and have their allocated memory reclaimed.

However, the inability to manually control garbage collection remains. There are times when it would be convenient to manually decide when and what memory is released. In order to release the memory of an object, it needs to be made explicitly unreachable. It is also not possible to programmatically trigger garbage collection in JavaScript — and will likely never be within the core language, although engines may expose APIs behind opt-in flags.

3. Var - Let - Const

💡 **Differences between var, let, and const:**

var	let	const
The scope of a *var* variable is functional scope.	The scope of a *let* variable is block scope.	The scope of a *const* variable is block scope.
It can be updated and re-declared into the scope.	It can be updated but cannot be re-declared into the scope.	It cannot be updated or re-declared into the scope.
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.
It can be accessed without initialization as its default value is “undefined”.	It cannot be accessed without initialization otherwise it will give ‘referenceError’.	It cannot be accessed without initialization, as it cannot be declared without initialization.
hoisting done, with initializing as ‘default’ value	Hoisting is done , but not initialized (this is the reason for the error when we access the let variable before declaration/initialization	Hoisting is done, but not initialized (this is the reason for error when we access the const variable before declaration/initialization

Note: Sometimes, users face problems while working with the *var* variable as they change its value of it in a particular block. So, users should use the *let* and *const* key words the to declare a variable in JavaScript.

Variables Lifecycle:

```
/*
    Variables Lifecycle:

    1. Declaration Phase => 2. Initialization Phase => 3. Assignment Phase
*/

/*
    var variables lifecycle

    1. Declaration Phase, Initialization Phase => variable === undefined | Initialized state
```

```
    2. Assignment Phase: variable = 'value' => variable === 'value' | Asssigned state
*/

/*
    Function declaration lifecycle

    function fullName() {

    }

    1. Declaration Phase, Initialization Phase, Assignment Phase => fullName is invoked | Assgined state
*/

/*
    let, const variables lifecycle

    1. Declaration Phase: RefenrenceError => Temporal Dead Zone | Declared sate

    2. Initialization Phase: let variable; => variable === undefined | Initialized state

    3. Assignment Phase: variable = 'value => variable === 'value' | Asssigned state
*/
```

Var

```
// var

// 1. Scope: global & function scope
// 2. Re-assignable - Có thể gán lại: Yes
// 3. Re-declarable - Có thể khai báo lại: Yes
// 4. Hoisting: Yes
```

1. Scope (global & function scope)

```
var language = "JavaScript"; // global scope

function foo() {
    var language = "Python";  // function scope
    framework = "React"; // global scope (Err in strict mode)
    console.log(language); // Python
}

console.log(language); // "JavaScript"
foo(); // "Python"

// console.log(window.language); // "JavaScript" -- log on browser
// console.log(window.framework); // "React" -- log on browser
```

2. Re-assignable (Yes)

```
var name = "Quang";
name = "Tuan"; // Re-assignable: Gán lại giá trị cho biến
console.log(name); // "Tuan"
```

3. Re-declarable (Yes)

```
var name = "Quang";
var name = "John"; // Re-declarable - Khai báo lại biến
console.log(name); // "John"
```

4. Hoisting (Yes)

```
console.log(myVar); // undefined
var myVar = "isHoistingSupport";
```

Let

```
// let

// 1. Scope: block scope {}
// 2. Re-assignable - Có thể gán lại: Yes
// 3. Re-declarable - Có thể khai báo lại: No
// 4. Hoisting: Yes => But stored in "Temporal Dead Zone"
```

1. Scope (block scope {})

```
{
    let language = "React"; // Block Scope = Phạm vi khối
    console.log(language); // "React"
}

console.log(window.language); // undefined - log on browser
// console.log(language); // ReferenceError: language is not defined

function foo() {
    let person = "Quang";
    console.log(person);
}

foo(); // Quang
// console.log(person); // ReferenceError: person is not defined
```

2. Re-assignable (Yes)

```
let age = 22;
age = 25;
console.log(age); // 25

let temp = "FPT Software"
temp = {
    isHoistingSupport: "No"
}
console.log(temp); // {isHoistingSupport: 'No'}
```

3. Re-declarable (No)

```
let company = "FPT Software";
let company = "Viettel"; // SyntaxError: Identifier 'company' has already been declared
```

4. Hoisting (No)

```
console.log(language); // ReferenceError: language is not defined
let language;
```

Const

```
// const -- hằng số

// 1. Scope: block scope {}
// 2. Re-assignable - Có thể gán lại: No
// 3. Re-declarable - Có thể khai báo lại: No
// 4. Hoisting: Yes => But stored in "Temporal Dead Zone"
```

1. Scope (block scope {})

```
const language = "JavaScript";

{
    const name = "Quang";
    const language = "React"; // Phạm vi khối
    console.log(name); // "Quang"
    console.log(language); // "React"
}

function foo() {
```

```
    const language = "Python";
    console.log(language); // Python
}

foo(); // "Python"
console.log(name); // ReferenceError: name is not defined
console.log(language); // "JavaScript"
// console.log(window.language); // undefined
```

2. Re-assignable (No)

```
const laptop = "Asus";
laptop = "Dell"; // TypeError: Assignment to constant variable

// Re-assign for values of object (Yes)
const user = {
    name: "Quang",
    age: 24,
}

user.name = "John";

console.log(user); // { name: 'John', age: 24 }
```

3. Re-declarable (No)

```
const laptop = "Asus";
const laptop = "Dell"; // SyntaxError: Identifier 'laptop' has already been declared
```

4. Hoisting (No)

```
console.log(language); // ReferenceError: language is not defined
const language;
```

Difference between var, let and const keywords in JavaScript

In **JavaScript**, users can declare a variable using 3 keywords that are **var, let, and const**. In this article, we will see the differences between the var, let, and const keywords. We will discuss the scope and other required concepts about each keyword.

JavaScript var keyword: The *var* is the oldest keyword to declare a variable in JavaScript.

Scope: Global scoped or function scoped. The scope of the *var* keyword is the global or function scope. It means variables defined outside the function can be accessed globally, and variables defined inside a particular function can be accessed within the function.

Example 1: Variable ‘a’ is declared globally. So, the scope of the variable ‘a’ is global, and it can be accessible everywhere in the program. The output shown is in the console.

```
var a = 10
function f(){
    console.log(a)
}
f();
console.log(a);
```

Output:

```
10
10
```

Example 2: The variable ‘a’ is declared inside the function. If the user tries to access it outside the function, it will display the error. Users can declare the 2 variables with the same name using the *var* keyword. Also, the user can reassign the value into

the *var* variable. The output shown in the console.

```
function f() {
    // It can be accessible any
    // where within this function
    var a = 10;
    console.log(a)
}
f();

// A cannot be accessible
// outside of function
console.log(a);
```

Output:

```
10
ReferenceError: a is not defined
```

Example 3: The user can re-declare the variable using *var* and the user can update *var* variable. The output is shown in the console.

```
var a = 10

// User can re-declare
// variable using var
var a = 8

// User can update var variable
a = 7
```

Output:

```
7
```

Example 4: If users use the var variable before the declaration, it initializes with the *undefined* value. The output is shown in the console.

```
console.log(a);
var a = 10;
```

Output:

```
undefined
```

JavaScript let keyword: The *let* keyword is an improved version of the *var* keyword.

Scope: block scoped: The scope of a *let* variable is only block scoped. It can't be accessible outside the particular block ({block}). Let's see the below example.

Example 1: The output is shown in the console.

```
let a = 10;
function f() {
    let b = 9
    console.log(b);
    console.log(a);
}
f();
```

Output:

```
9
10
```

Example 2: The code returns an error because we are accessing the *let* variable outside the function block. The output is shown in the console.

```
let a = 10;
function f() {
    if (true) {
        let b = 9

        // It prints 9
        console.log(b);
    }

    // It gives error as it
    // defined in if block
    console.log(b);
}
f()

// It prints 10
console.log(a)
```

Output:

```
9
ReferenceError: b is not defined
```

Example 3: Users cannot re-declare the variable defined with the *let* keyword but can update it.

```
let a = 10

// It is not allowed
let a = 10

// It is allowed
a = 10
```

Output:

```
Uncaught SyntaxError: Identifier 'a' has already been declared
```

Example 4: Users can declare the variable with the same name in different blocks using the *let* keyword.

```
let a = 10;
if (true) {
  let a = 9;
  console.log(a); // It prints 9
}
console.log(a); // It prints 10
```

Output:

```
9
10
```

Example 5: If users use the *let* variable before the declaration, it does not initialize with *undefined* just like a *var* variable, and returns an error.

```
console.log(a);
let a = 10;
```

Output:

```
Uncaught ReferenceError: Cannot access 'a' before initialization
```

const keyword in JavaScript: The *const* keyword has all the properties that are the same as the *let* keyword, except the user cannot update it.

Scope: block scoped: When users declare a *const* variable, they need to initialize it, otherwise, it returns an error. The user cannot update the *const* variable once it is declared.

Example 1: We are changing the value of the const variable so that it returns an error. The output is shown in the console.

```
const a = 10;
function f() {
    a = 9
    console.log(a)
}
f();
```

Output:

```
TypeError:Assignment to constant variable.
```

Example 2: Users cannot change the properties of the *const* object, but they can change the value of the properties of the *const* object.

```
const a = {
    prop1: 10,
    prop2: 9
}

// It is allowed
a.prop1 = 3

// It is not allowed
a = {
    b: 10,
    prop2: 9
}
```

Output:

```
Uncaught SyntaxError:Unexpected identifier
```

Differences between var, let, and const

var	let	const
The scope of a *var* variable is functional scope.	The scope of a *let* variable is block scope.	The scope of a *const* variable is block scope.
It can be updated and re-declared into the scope.	It can be updated but cannot be re-declared into the scope.	It cannot be updated or re-declared into the scope.
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.
It can be accessed without initialization as its default value is “undefined”.	It cannot be accessed without initialization otherwise it will give ‘referenceError’.	It cannot be accessed without initialization, as it cannot be declared without initialization.
hoisting done, with initializing as ‘default’ value	Hoisting is done , but not initialized (this is the reason for the error when we access the let variable before declaration/initialization	Hoisting is done, but not initialized (this is the reason for error when we access the const variable before declaration/initialization

Note: Sometimes, users face problems while working with the *var* variable as they change its value of it in a particular block. So, users should use the *let* and *const* keywordsthe to declare a variable in JavaScript.

4. Hoisting

💡 **Definition:** Hoisting is the process of virtually moving the variable or function definition to the beginning of the scope, usually for variable statement `var` and function declaration `function fun() {...}`.

When `let` (and also `const` and `class`, which have similar declaration behavior as `let`) declarations were introduced by ES2015, many developers including myself were using the *hoisting* definition to describe how variables are accessed. But after more search on the question, surprisingly for me *hoisting* is not the correct term to describe the initialization and availability of the `let` variables.

ES2015 provides a different and improved mechanism for `let`. It demands stricter variable declaration practices (you can't use before definition) and as result better code quality. Let's dive into more details about this process.

1. Error prone var hoisting

Sometimes I see a weird practice of variables `var varname` and functions `function funName() {...}` declaration in any place in the scope:

```
// var hoisting
num;     // => undefined
var num;
num = 10;
num;     // => 10
// function hoisting
getPi;   // => function getPi() {...}
getPi(); // => 3.14
function getPi() {
  return 3.14;
}
```

The variable `num` is accessed before declaration `var num`, so it is evaluated to `undefined`. The function `function getPi() {...}` is defined at the end of file. However the function can be called before declaration `getPi()`, as it is hoisted to the top of the scope.

This is the classical *hoisting*.

As it turns out, the possibility to first use and then declare a variable or function creates confusion. Suppose you scroll a big file and suddenly see an undeclared variable... how the hell it does appear here and where is it defined? Of course a practiced JavaScript developer won't code this way. But in the thousands of JavaScript GitHub repos is quite possible to deal with such code.

Even looking at the code sample presented above, it is difficult to understand the declaration flow in the code.

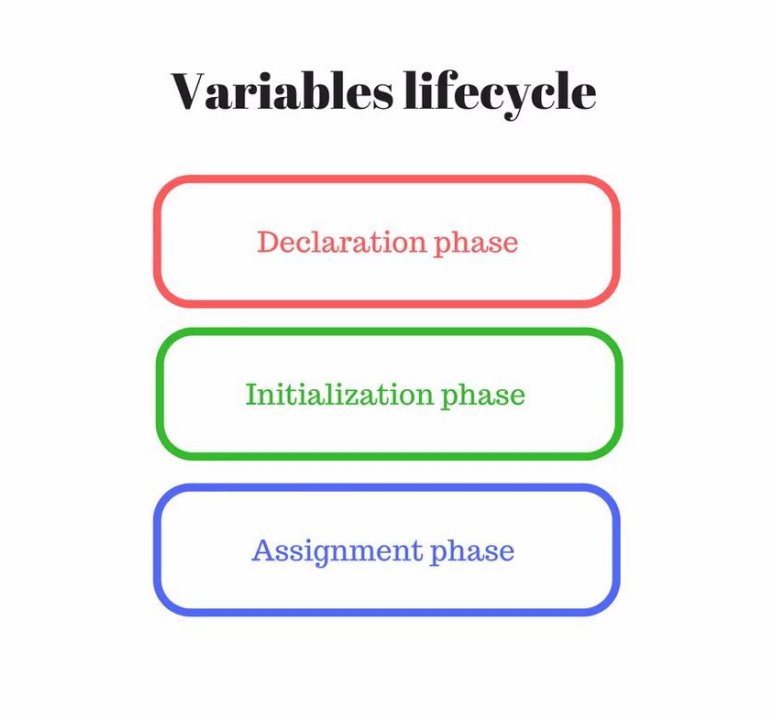
Naturally first you declare or describe an unknown term. And only later make phrases with it. `let` encourages you to follow this approach with variables.

2. Under the hood: variables lifecycle

When the engine works with variables, their lifecycle consists of the following phases:

1. **Declaration phase** is registering a variable in the scope.
2. **Initialization phase** is allocating memory and creating a binding for the variable in the scope. At this step the variable is automatically initialized with `undefined`.
3. **Assignment phase** is assigning a value to the initialized variable.

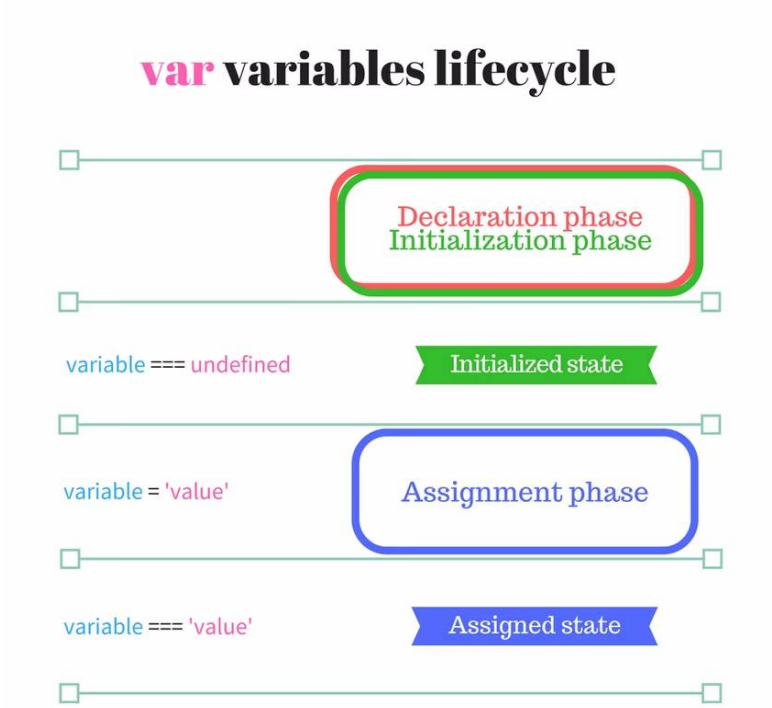
A variable has **unitialized** state when it passed the declaration phase, yet didn't reach the initilization.



Notice that in terms of variables lifecycle, *declaration phase* is a different term than generally speaking *variable declaration*. In simple words, the engine processes the variable declaration in 3 phases: declaration phase, initialization phase and assignment phase.

3. var variables lifecycle

Being familiar with lifecycle phases, let's use them to describe how the engine handles `var` variables.



Suppose a scenario when JavaScript encounters a function scope with `var variable` statement inside. The variable passes the *declaration phase* and right away the *initialization phase* at the beginning of the scope, before any statements are executed (step 1). `var variable` statement position in the function scope does not influence the declaration and initialization phases.

After declaration and initialization, but before assignment phase, the variable has `undefined` value and can be used already.

On *assignment phase* `variable = 'value'` the variable receives its initial value (step 2).

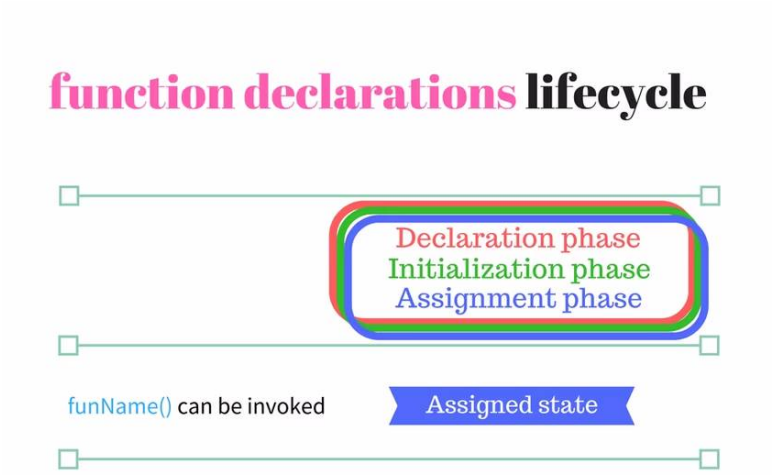
Strictly *hoisting* consists in the idea that a variable is *declared and initialized at the beginning* of the function scope. There is no gap between declaration and initialization phases. Let's study an example. The following code creates a function scope with a `var` statement inside:

When JavaScript starts executing `multipleByTen(4)` and enters the function scope, the variable `ten` passes declaration and initialization steps, before the first statement. So when calling `console.log(ten)` it is logged `undefined`.The statement `ten = 10` assigns an initial value. After assignment, the line `console.log(ten)` logs correctly `10` value.

```
function multiplyByTen(number) {
  console.log(ten); // => undefined
  var ten;
  ten = 10;
  console.log(ten); // => 10
  return number * ten;
}
multiplyByTen(4); // => 40
```

4. Function declaration lifecycle

In case of a *function declaration statement* `function funName() {...}` it's even easier.



The *declaration, initialization and assignment phases* happen at once at the beginning of the enclosing function scope (only one step). `funName()` can be invoked in any place of the scope, not depending on the declaration statement position (it can be even at the end).

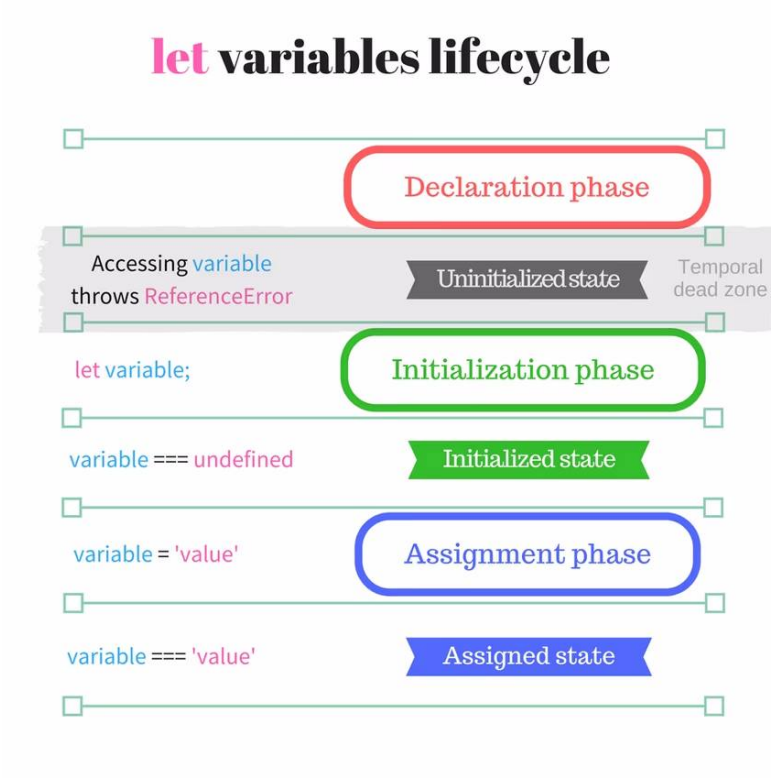
The following code sample demonstrates the function hoisting:

```
function sumArray(array) {
  return array.reduce(sum);
  function sum(a, b) {
    return a + b;
  }
}
sumArray([5, 10, 8]); // => 23
```

When JavaScript executes `sumArray([5, 10, 8])`, it enters `sumArray` function scope. Inside this scope, immediately before any statement execution, `sum` passes all 3 phases: declaration, initialization and assignment. This way `array.reduce(sum)` can use `sum` even before its declaration statement `function sum(a, b) {...}`.

5. let, const variables lifecycle

`let` variables are processed differently than `var`. The main distinction is that declaration and initialization phases are **split**.



Now let's study a scenario when the interpreter enters a block scope that contains a `let variable` statement. Immediately the variable passes the *declaration phase*, registering its name in the scope (step 1).Then interpreter continues parsing the block statements line by line.

If you try to access `variable` at this stage, JavaScript will throw `ReferenceError: variable is not defined`. It happens because the variable state is *uninitialized*.`variable` is in the *temporal dead zone*.

When interpreter reaches the statement `let variable`, the initilization phase is passed (step 2). Now the variable state is *initialized* and accessing it evaluates to `undefined`. The variable exits the *temporal dead zone*.

Later when an assignment statement appears `variable = 'value'`, the assignment phase is passed (step 3).

If JavaScript encounters `let variable = 'value'`, then initialization and assignment happen in a single statement.

Let's follow an example. `let` variable `number` is created in a block scope:

```
let condition = true;
if (condition) {
  // console.log(number); // => Throws ReferenceError
  let number;
  console.log(number); // => undefined
  number = 5;
  console.log(number); // => 5
}
```

When JavaScript enters `if (condition) {...}` block scope, `number` instantly passes the declaration phase. Because `number` has unitialized state and is in a temporal dead zone, an attempt to access the variable throws `ReferenceError: number is not defined`. Later the statement `let number` makes the initialization. Now the variable can be accessed, but its value is `undefined`.The assignment statement `number = 5` of course makes the assignment phase.

`const` and `class` types have the same lifecycle as `let`, other than the assignment can happen only once.

5.1 Why hoisting is not valid in let lifecycle

As mentioned above, *hoisting* is variable's *coupled* declaration and initialization at the top of the scope. `let` lifecycle however *decouples* declaration and initialization phases. Decoupling vanishes the *hoisting* term for `let`.The gap between the two phases creates the temporal dead zone, where the variable cannot be accessed.

In a sci-fi style, the collapsed hoisting in `let` lifecycle creates the temporal dead zone.

The freedom to declare variables using `var` is error prone. Based on this lesson, ES2015 introduces `let`. It uses an improved algorithm to declare variables and additionally is block scoped.

Because the declaration and initialization phases are decoupled, hoisting is not valid for a `let` variable (including for `const` and `class`). Before initialization, the variable is in temporal dead zone and is not accessible.

To keep the variables declaration smooth, these tips are recommended:

- Declare, initialize and then use variables. This flow is correct and easy to follow.
- Keep the variables as hidden as possible. The less variables are exposed, the more modular your code becomes.

```
// Hoisting

/**
 * var
 * Hoisting: Yes - Declaration, Initialization Phase
 */

/**
 * let, const
 * Hoisting: Yes - Declaration Phase: RefenrenceError => Temporal Dead Zone
 */

/**
 * function
 * Hoisting: Yes - Declaration, Initialization, Assignment Phase => function is invoked
 */
```

Var (Hoisting)

```
console.log(age); // undefined
console.log(fullName); // ReferenceError: fullName is not defined

var age = 16;

// But not hoisting when declared function by var keyword
exFunc(); // exFunc is not a function

var exFunc = function () {
  console.log('exFunc');
}
```

Function (Hoisting)

```
console.log(sum(6, 9)); // 15

function sum(a, b) {
  return a + b;
}
```

Let & Const (No Hoisting)

```
// Cannot access "myLet" & "myConst" before initialization
console.log(myLet); // ReferenceError: myLet is not defined
console.log(myConst); // ReferenceError: myConst is not defined

// "Temporal Dead Zone" - Vùng nhớ tạm thời không truy cập được
let myLet = "my let";
const myConst = "my const";
```

5. Variable Scope

💡 **JavaScript has 3 types of scope:** .

- Block scope
- Function scope
- Global scope

Block Scope

Before ES6 (2015), JavaScript had only **Global Scope** and **Function Scope**.

ES6 introduced two important new JavaScript keywords: `let` and `const`.

These two keywords provide **Block Scope** in JavaScript.

Variables declared inside a { } block cannot be accessed from outside the block:

Example

```
{
  let x = 2;
}
// x can NOT be used here
```

Variables declared with the `var` keyword can NOT have block scope.

Variables declared inside a { } block can be accessed from outside the block.

Example

```
{
  var x = 2;
}
// x CAN be used here
```

Local Scope

Variables declared within a JavaScript function, become **LOCAL** to the function.

Example

```
// code here can NOT use carName

function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

Local variables have **Function Scope**:

They can only be accessed from within the function.

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

Function Scope

JavaScript has function scope: Each function creates a new scope.

Variables defined inside a function are not accessible (visible) from outside the function.

Variables declared with `var`, `let` and `const` are quite similar when declared inside a function.

They all have **Function Scope**:

```
function myFunction() {
  var carName = "Volvo";   // Function Scope
}
```

```
function myFunction() {
  let carName = "Volvo";   // Function Scope
}
```

```
function myFunction() {
  const carName = "Volvo";   // Function Scope
}
```

Global JavaScript Variables

A variable declared outside a function, becomes **GLOBAL**.

Example

```
let carName = "Volvo";
// code here can use carName

function myFunction() {
// code here can also use carName
}
```

A global variable has **Global Scope**:

All scripts and functions on a web page can access it.

Global Scope

Variables declared **Globally** (outside any function) have **Global Scope**.

Global variables can be accessed from anywhere in a JavaScript program.

Variables declared with `var`, `let` and `const` are quite similar when declared outside a block.

They all have **Global Scope**:

```
var x = 2;       // Global scope
```

```
let x = 2;       // Global scope
```

```
const x = 2;       // Global scope
```

JavaScript Variables

In JavaScript, objects and functions are also variables.

Scope determines the accessibility of variables, objects, and functions from different parts of the code.

Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

This code example will declare a global variable `carName`, even if the value is assigned inside a function.

Example

```
myFunction();

// code here can use carName

function myFunction() {
  carName = "Volvo";
}
```

Strict Mode

All modern browsers support running JavaScript in "Strict Mode".

You will learn more about how to use strict mode in a later chapter of this tutorial.

In "Strict Mode", undeclared variables are not automatically global.

Global Variables in HTML

With JavaScript, the global scope is the JavaScript environment.

In HTML, the global scope is the window object.

Global variables defined with the `var` keyword belong to the window object:

Example

```
var carName = "Volvo";
// code here can use window.carName
```

Global variables defined with the `let` keyword do not belong to the window object:

Example

```
let carName = "Volvo";
// code here can not use window.carName
```

Warning

Do NOT create global variables unless you intend to.

Your global variables (or functions) can overwrite window variables (or functions).Any function, including the window object, can overwrite your global variables and functions.

The Lifetime of JavaScript Variables

The lifetime of a JavaScript variable starts when it is declared.

Function (local) variables are deleted when the function is completed.

In a web browser, global variables are deleted when you close the browser window (or tab).

Function Arguments

Function arguments (parameters) work as local variables inside functions.

Scope chain: Chuỗi phạm vi

```
{
  const myScopeChain = "My scope chain 1";
  console.log(myScopeChain); // "My scope chain 1"
  {
    const myScopeChain = "My scope chain 2";
    console.log(myScopeChain); // "My scope chain 2"
    {
      console.log(myScopeChain); // "My scope chain 2"
    }
  }
}
```

Global Execution Context (GEC)

```
// Global Execution Context (GEC)

/*
    Whenever the JavaScript engine receives a script file, it first creates a default
    Execution Context known as the Global Execution Context (GEC).

    The GEC is the base/default Execution Context where all JavaScript code that is not
    inside of a function gets executed.
*/
```

6. Object

💡 **Definition:** The `Object` type represents one of JavaScript's data types. It is used to store various keyed collections and more complex entities. Objects can be created using the `Object()` constructor or the object initializer / literal syntax.

Description

Nearly all objects in JavaScript are instances of `Object`; a typical object inherits properties (including methods) from `Object.prototype`, although these properties may be shadowed (a.k.a. overridden). The only objects that don't inherit from `Object.prototype` are those with `null` prototype, or descended from other `null` prototype objects.

Changes to the `Object.prototype` object are seen by **all** objects through prototype chaining, unless the properties and methods subject to those changes are overridden further along the prototype chain. This provides a very powerful although potentially dangerous mechanism to override or extend object behavior. To make it more secure, `Object.prototype` is the only object in the core JavaScript language that has immutable prototype — the prototype of `Object.prototype` is always `null` and not changeable.

Object prototype properties

You should avoid calling any `Object.prototype` method, especially those that are not intended to be polymorphic (i.e. only its initial behavior makes sense and no descending object could override it in a meaningful way). All objects descending from `Object.prototype` may define a custom own property that has the same name, but with entirely different semantics from what you expect. Furthermore, these properties are not inherited by `null`-prototype objects. All modern JavaScript utilities for working with objects are static. More specifically:

- `valueOf()`, `toString()`, and `toLocaleString()` exist to be polymorphic and you should expect the object to define its own implementation with sensible behaviors, so you can call them as instance methods. However, `valueOf()` and `toString()` are usually implicitly called through type conversion and you don't need to call them yourself in your code.

- `__defineGetter__()`, `__defineSetter__()`, `__lookupGetter__()`, and `__lookupSetter__()` are deprecated and should not be used. Use the static alternatives `Object.defineProperty()` and `Object.getOwnPropertyDescriptor()` instead.
- The `__proto__` property is deprecated and should not be used.
  The `Object.getPrototypeOf()` and `Object.setPrototypeOf()` alternatives are static methods.
- The `propertyIsEnumerable()` and `hasOwnProperty()` methods can be replaced with the `Object.getOwnPropertyDescriptor()` and `Object.hasOwn()` static methods, respectively.
- The `isPrototypeOf()` method can usually be replaced with `instanceof`, if you are checking the `prototype` property of a constructor.

In case where a semantically equivalent static method doesn't exist, or if you really want to use the `Object.prototype` method, you should directly `call()` the `Object.prototype` method on your target object instead, to prevent the object from having an overriding property that produces unexpected results.

```
const obj = {
  foo: 1,
  // You should not define such a method on your own object,
  // but you may not be able to prevent it from happening if
  // you are receiving the object from external input
  propertyIsEnumerable() {
    return false;
  }
}

obj.propertyIsEnumerable("foo"); // false; unexpected result
Object.prototype.propertyIsEnumerable.call(obj, "foo"); // true; expected result
```

Deleting a property from an object

There isn't any method in an Object itself to delete its own properties (such as `Map.prototype.delete()`). To do so, one must use the delete operator.

null-prototype objects

Almost all objects in JavaScript ultimately inherit from `Object.prototype` (see inheritance and the prototype chain). However, you may create `null`-prototype objects using `Object.create(null)` or the object initializer syntax with `__proto__: null` (note: the `__proto__` key in object literals is different from the deprecated `Object.prototype.__proto__` property). You can also change the prototype of an existing object to `null` by calling `Object.setPrototypeOf(obj, null)`.

```
const obj = Object.create(null);
const obj2 = { __proto__: null };
```

An object with a `null` prototype can behave in unexpected ways, because it doesn't inherit any object methods from `Object.prototype`. This is especially true when debugging, since common object-property converting/detecting utility functions may generate errors, or lose information (especially if using silent error-traps that ignore errors).

For example, the lack of `Object.prototype.toString()` often makes debugging intractable:

```
const normalObj = {}; // create a normal object
const nullProtoObj = Object.create(null); // create an object with "null" prototype

console.log(`normalObj is: ${normalObj}`); // shows "normalObj is: [object Object]"
console.log(`nullProtoObj is: ${nullProtoObj}`); // throws error: Cannot convert object to primitive value

alert(normalObj); // shows [object Object]
alert(nullProtoObj); // throws error: Cannot convert object to primitive value
```

Other methods will fail as well.

```
normalObj.valueOf(); // shows {}
nullProtoObj.valueOf(); // throws error: nullProtoObj.valueOf is not a function

normalObj.hasOwnProperty("p"); // shows "true"
nullProtoObj.hasOwnProperty("p"); // throws error: nullProtoObj.hasOwnProperty is not a function
```

```
normalObj.constructor; // shows "Object() { [native code] }"
nullProtoObj.constructor; // shows "undefined"
```

We can add the `toString` method back to the null-prototype object by assigning it one:

```
nullProtoObj.toString = Object.prototype.toString; // since new object lacks toString, add the original generic one back

console.log(nullProtoObj.toString()); // shows "[object Object]"
console.log(`nullProtoObj is: ${nullProtoObj}`); // shows "nullProtoObj is: [object Object]"
```

Unlike normal objects, in which `toString()` is on the object's prototype, the `toString()` method here is an own property of `nullProtoObj`. This is because `nullProtoObj` has no (`null`) prototype.

In practice, objects with `null` prototype are usually used as a cheap substitute for maps. The presence of `Object.prototype` properties will cause some bugs:

```
const ages = { alice: 18, bob: 27 };

function hasPerson(name) {
  return name in ages;
}

function getAge(name) {
  return ages[name];
}

hasPerson("hasOwnProperty"); // true
getAge("toString"); // [Function: toString]
```

Using a null-prototype object removes this hazard without introducing too much complexity to the `hasPerson` and `getAge` functions:

```
const ages = Object.create(null, {
  alice: { value: 18, enumerable: true },
  bob: { value: 27, enumerable: true },
});

hasPerson("hasOwnProperty"); // false
getAge("toString"); // undefined
```

In such case, the addition of any method should be done cautiously, as they can be confused with the other key-value pairs stored as data.

Making your object not inherit from `Object.prototype` also prevents prototype pollution attacks. If a malicious script adds a property to `Object.prototype`, it will be accessible on every object in your program, except objects that have null prototype.

```
const user = {};

// A malicious script:
Object.prototype.authenticated = true;

// Unexpectedly allowing unauthenticated user to pass through
if (user.authenticated) {
  // access confidential data
}
```

JavaScript also has built-in APIs that produce `null`-prototype objects, especially those that use objects as ad hoc key-value collections. For example:

- The return value of `Array.prototype.group()`
- The `groups` and `indices.groups` properties of the result of `RegExp.prototype.exec()`

- `Array.prototype[@@unscopables]` (all `@@unscopables` objects should have `null` prototype)
- `import.meta`
- Module namespace objects, obtained through `import * as ns from "module";` or `import()`

Object coercion

Many built-in operations that expect objects first coerce their arguments to objects. The operation can be summarized as follows:

- Objects are returned as-is.
- `undefined` and `null` throw a `TypeError`.
- `Number`, `String`, `Boolean`, `Symbol`, `BigInt` primitives are wrapped into their corresponding object wrappers.

The best way to achieve the same effect in JavaScript is through the `Object()` constructor. `Object(x)` converts `x` to an object, and for `undefined` or `null`, it returns a plain object instead of throwing a `TypeError`.

Places that use object coercion include:

- The `object` parameter of `for...in` loops.
- The `this` value of `Array` methods.
- Parameters of `Object` methods such as `Object.keys()`.
- Auto-boxing when a property is accessed on a primitive value, since primitives do not have properties.
- The `this` value when calling a non-strict function. Primitives are boxed while `null` and `undefined` are replaced with the global object.

Unlike conversion to primitives, the object coercion process itself is not observable in any way, since it doesn't invoke custom code like `toString` or `valueOf` methods.

Constructor

`Object()` Turns the input into an object.

Static methods

`Object.assign()` Copies the values of all enumerable own properties from one or more source objects to a target object. `Object.create()` Creates a new object with the specified prototype object and properties. `Object.defineProperty()` Adds the named property described by a given descriptor to an object. `Object.defineProperties()` Adds the named properties described by the given descriptors to an object. `Object.entries()` Returns an array containing all of the `[key, value]` pairs of a given object's **own** enumerable string properties. `Object.freeze()` Freezes an object. Other code cannot delete or change its properties. `Object.fromEntries()` Returns a new object from an iterable of `[key, value]` pairs. (This is the reverse of `Object.entries`). `Object.getOwnPropertyDescriptor()` Returns a property descriptor for a named property on an object. `Object.getOwnPropertyDescriptors()` Returns an object containing all own property descriptors for an object. `Object.getOwnPropertyNames()` Returns an array containing the names of all of the given object's **own** enumerable and non-enumerable properties. `Object.getOwnPropertySymbols()` Returns an array of all symbol properties found directly upon a given object. `Object.getPrototypeOf()` Returns the prototype (internal `[[Prototype]]` property) of the specified object. `Object.is()` Compares if two values are the same value. Equates all `NaN` values (which differs from both `IsLooselyEqual` used by `==` and `IsStrictlyEqual` used by `===`). `Object.isExtensible()` Determines if extending of an object is allowed. `Object.isFrozen()` Determines if an object was frozen. `Object.isSealed()` Determines if an object is sealed. `Object.keys()` Returns an array containing the names of all of the given object's **own** enumerable string properties. `Object.preventExtensions()` Prevents any extensions of an object. `Object.seal()` Prevents other code from deleting properties of an object. `Object.setPrototypeOf()` Sets the object's prototype (its internal `[[Prototype]]` property). `Object.values()` Returns an array containing the values that correspond to all of a given object's **own** enumerable string properties.

Instance properties

`Object.prototype.constructor` Specifies the function that creates an object's prototype. `Object.prototype.__proto__` DeprecatedPoints to the object which was used as prototype when the object was instantiated.

Instance methods

`Object.prototype.__defineGetter__()` Associates a function with a property that, when accessed, executes that function and returns its return value. `Object.prototype.__defineSetter__()` Associates a function with a property that, when set, executes that function which modifies the property. `Object.prototype.__lookupGetter__()` Returns the function bound as a getter to the specified property. `Object.prototype.__lookupSetter__()` Returns the function bound as a setter to the specified property. `Object.prototype.hasOwnProperty()` Returns a boolean indicating whether an object contains the specified property as a direct property of that object and not inherited through the prototype chain. `Object.prototype.isPrototypeOf()` Returns a boolean indicating whether the object this method is called upon is in the prototype chain of the specified object. `Object.prototype.propertyIsEnumerable()` Returns a boolean indicating whether the specified property is the object's enumerable own property. `Object.prototype.toLocaleString()` Calls `toString()`. `Object.prototype.toString()` Returns a string representation of the object. `Object.prototype.valueOf()` Returns the primitive value of the specified object.

Examples

Constructing empty objects

The following examples store an empty `Object` object in `o`:

```
const o1 = new Object();
const o2 = new Object(undefined);
const o3 = new Object(null);
```

Using Object to create Boolean objects

The following examples store `Boolean` objects in `o`:

```
// equivalent to const o = new Boolean(true)
const o = new Object(true);

// equivalent to const o = new Boolean(false)
const o = new Object(Boolean());
```

Object prototypes

When altering the behavior of existing `Object.prototype` methods, consider injecting code by wrapping your extension before or after the existing logic. For example, this (untested) code will pre-conditionally execute custom logic before the built-in logic or someone else's extension is executed.

When modifying prototypes with hooks, pass `this` and the arguments (the call state) to the current behavior by calling `apply()` on the function. This pattern can be used for any prototype, such as `Node.prototype`, `Function.prototype`, etc.

```
const current = Object.prototype.valueOf;

// Since my property "-prop-value" is cross-cutting and isn't always
// on the same prototype chain, I want to modify Object.prototype:
Object.prototype.valueOf = function (...args) {
  if (Object.hasOwn(this, '-prop-value')) {
    return this['-prop-value'];
  } else {
    // It doesn't look like one of my objects, so let's fall back on
    // the default behavior by reproducing the current behavior as best we can.
    // The apply behaves like "super" in some other languages.
    // Even though valueOf() doesn't take arguments, some other hook may.
    return current.apply(this, args);
  }
}
```

Warning: Modifying the `prototype` property of any built-in constructor is considered a bad practice and risks forward compatibility.

---

1. `Object.assign()`
2. `Object.create()`
3. `Object.defineProperties()`

4. `Object.defineProperty()`
5. `Object.entries()`
6. `Object.freeze()`
7. `Object.fromEntries()`
8. `Object.getOwnPropertyDescriptor()`
9. `Object.getOwnPropertyDescriptors()`
10. `Object.getOwnPropertyNames()`
11. `Object.getOwnPropertySymbols()`
12. `Object.getPrototypeOf()`
13. `Object.hasOwn()`
14. `Object.prototype.hasOwnProperty()`
15. `Object.is()`
16. `Object.isExtensible()`
17. `Object.isFrozen()`
18. `Object.prototype.isPrototypeOf()`
19. `Object.isSealed()`
20. `Object.keys()`
21. `Object.preventExtensions()`
22. `Object.prototype.propertyIsEnumerable()`
23. `Object.seal()`
24. `Object.setPrototypeOf()`
25. `Object.prototype.toLocaleString()`
26. `Object.prototype.toString()`
27. `Object.prototype.valueOf()`
28. `Object.values()`

7. Destructuring - Rest & Spread

💡 Định nghĩa**:** Destructuring là một cú pháp cho phép bạn gán các thuộc tính của một Object hoặc một Array. Điều này có thể làm giảm đáng kể các dòng mã cần thiết để thao tác dữ liệu trong các cấu trúc này.
Có hai loại Destructuring: `Destructuring Objects` và `Destructuring Arrays`.

Destructuring Objects

Destructuring Objects cho phép bạn tạo ra một hay nhiều new variables sử dụng những property của một Objects. Xem ví dụ dưới đây:

```
const note = {
    id: 1,
    website: 'anonystick.com',
    date: '17/07/2014',
}
```

Theo cách truyền thống thì chúng ta sẽ lấy ra những giá trị như cú pháp sau:

```
const id =note.idconst website = note.website
const date = note.date

console.log(id)
```

```
console.log(website)
console.log(date)
```

Nhưng với việc sử dụng `object destructuring` chỉ với một dòng code (Thủ thuật viết một dòng code), chúng ta có thể get được những giá trị ấy miễn là trùng tên của thuộc tính trong object là được:

```
// Destructure properties into variablesconst { id, website, date } = note

console.log(id)
console.log(website)
console.log(date)
```

Destructuring Arrays

Array destructuring cho phép bạn tạo ra một new variables bằng cách sử dụng giá trị mỗi index của Array. Xem ví dụ dưới đây cho nó dễ hiểu, chứ nói vậy chả hiểu đâu. Ta có một Array là thông tin về trang "blog javascript" được tạo ngày

```
const date = ['2014', '17', '07']
```

Như ở Object thì ta lần lượt lấy giá trị của mỗi item theo index

```
// Create variables from the Array items
const year = date[0]
const month = date[1]
const day = date[2]
```

Nhưng giờ đây với việc sử dụng Array Destructuring thì công việc sẽ trở nên dế dàng hơn nhiều

```
// Destructure Array values into variables
const [year, month, day] = date

console.log(year) // 2014
console.log(month) // 17
console.log(day) // 07
```

Còn nhiều thủ thuật khác hay hơn nữa nên bạn hay dành thời gian tầm một phút bạn sẽ có được tất cả sự hiểu biết về tính năng destructuring "Destructuring là gì? Vì sao phải dùng đến nó"

Spread operator là gì?

Spread operator là ba dấu chấm (...), có thể chuyển đổi một mảng thành một chuỗi các tham số được phân tách bằng dấu phẩy. Nói cho dễ hiểu, nó giống như một cái xương và một cái xương sườn vậy, chia nhỏ một phần tử lớn thành những phần tử nhỏ riêng lẻ.

Nói về `Spread` tips javascript cũng đã có rất nhiều bài viết liên quan đến tính năng này. Spread syntax hay còn gọi là three dot (...) là một bổ sung hữu ích khác cho JavaScript để làm việc với các Arrays, Objects và các function calls.

Hơn nữa Spread có thể tao ra một cấu trúc dữ liệu shallow copy để tăng tính thao tác dữ liệu. Cũng giống như destructuring thì Spread cũng làm việc nhiều với Arrays và Objects.

 # Spread with Arrays

Ví dụ trường hợp thức tế thì ta có thể merge array sử dụng `concat`.

```
// Create an Arrayconst tools = ['hammer', 'screwdriver']
const otherTools = ['wrench', 'saw']

// Concatenate tools and otherTools togetherconst allTools = tools.concat(otherTools)

console.log(allTools);

//(4) ["hammer",
"screwdriver",
"wrench",
```

```
"saw"
]
```

Nhưng giờ đây đã khác xưa rất nhiều rồi, hãy xem đây khi sử dụng Spread systax.

```
// Unpack the tools Array into the allTools Arrayconst allTools = [...tools, ...otherTools]

console.log(allTools)
```

Còn rất nhiều thứ rất hay đang chờ đón bạn, chút nữa sẽ cung cấp đanh cho những ai đủ kiên nhẫn đọc hết bài viết này.

```
const ocean = ['🐙', '🦀'];

const aquarium = [...ocean, '🐡'];// Add a single valueconst sushi = [...ocean, '🐡', '🍚'];// Add multiple values

aquarium;// ['🐙', '🦀', '🐡']
sushi;// ['🐙', '🦀', '🐡', '🍚']// Original Array Not Affected
ocean;// ['🐙', '🦀']
```

 # Spread with Objects

Khi sử dụng Spread thì chúng ta có thể copy và update một object như những gì mà `Object.assign()` đã làm

```
// Create an Object and a copied Object with Object.assign()const originalObject = { enabled: true, darkMode: false }
const secondObject = Object.assign({}, originalObject)

console.log(secondObject);//{enabled: true, darkMode: false}
```

Sử dụng Spread syntax thì sao? ez game.

```
// Create an object and a copied object with spreadconst originalObject = { enabled: true, darkMode: false }
const secondObject = { ...originalObject }

console.log(secondObject);//{enabled: true, darkMode: false}
```

 # Spread with Function Calls

Giả sử chúng ta có một function như thế này

```
// Create a function to multiply three itemsfunction multiply(a, b, c) {
  return a * b * c
}
```

Nếu bình thường thì sao, thì add từng paramsx zô chứ sao:

```
multiply(1, 2, 3) ;// 6
```

Nhưng khi sử dụng Spread trong function calls thì rất đơn giản

```
const numbers = [1, 2, 3]

multiply(...numbers);//6
```

Rest Parameters là gì?

Tính năng cuối cùng bạn sẽ tìm hiểu trong bài viết này đó là `Rest Parameters`. Cú pháp này giống như Spread Syntax (...) nhưng có tác dụng ngược lại. Ví dụ Ví dụ, trong hàm `restTest`, nếu chúng ta muốn `args` là một mảng bao gồm một số lượng đối số không xác định, chúng ta có thể có:

```
function restTest(...args) {
  console.log(args)
}
```

```
restTest(1, 2, 3, 4, 5, 6);// [1, 2, 3, 4, 5, 6]
```

Và đây là một bài viết về Rest Parameters và Spread Syntax dành cho những bạn kiên nhẫn đọc tới đây. Ở bài viết này thì mọi điều sẽ được sáng tỏ, bạn sẽ phân biệt được giữa Rest Parameters và Spread Syntax. Và khi nào sử dụng chúng. Thực hành về Rest Parameters và Spread Syntax

Tóm tắt

Trên đây là những kiến thức giúp bạn hiểu về những tính năng hỗ trợ thêm cho Arrays và Objecs như Destructure, Spread systax, Rest Parameters.

Tóm lại:

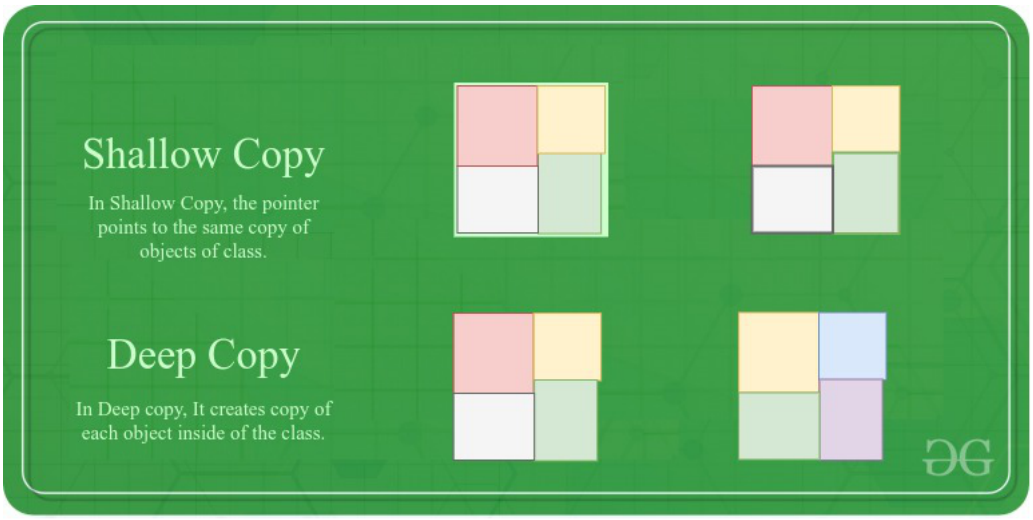
- **Destructure sử dụng để tạo ra một new variables từ array items, hoặc object properties.**
- **Spread syntax sử dụng để unpack terables của một arrays, objects, và function calls.**
- **Rest parameter là một cú pháp tạo ra một array từ một số lượng giá trị không xác định.**

8. Shallow Copy & Deep Copy

💡 **Shallow Copy:** Shallow repetition is quicker. However, it's "lazy" it handles pointers and references. Rather than creating a contemporary copy of the particular knowledge the pointer points to, it simply copies over the pointer price. So, each the first and therefore the copy can have pointers that reference constant underlying knowledge.

Deep Copy: Deep repetition truly clones the underlying data. It is not shared between the first and therefore the copy.

Difference between Shallow and Deep copy



Shallow Copy	Deep Copy
Shallow Copy stores the references of objects to the original memory address.	Deep copy stores copies of the object's value.
Shallow Copy reflects changes made to the new/copied object in the original object.	Deep copy doesn't reflect changes made to the new/copied object in the original object.
Shallow Copy stores the copy of the original object and points the references to the objects.	Deep copy stores the copy of the original object and recursively copies the objects as well.

Shallow copy is faster.	Deep copy is comparatively slower.

Trong lập trình, chúng ta lưu trữ data dưới dạng các biến. Tạo một copy tức là khởi tạo một biến mới có cùng giá trị. Tuy nhiên luôn có cạm bẫy luôn rình rập, đó là deep copy và shallow copy.

Deep copy: tức là tạo mới một biến có cùng giá trị và được cắt đứt quan hệ hoàn toàn với biến được copy.

Shallow copy: có ý nghĩa rằng sau khi copy, biến mới hoặc các thành phần của biến mới vẫn còn quan hệ dây mơ rễ má với biến ban đầu, nguy hiểm quá nhỉ. Để hiểu thêm về copy, chúng ta cùng tìm hiểu về cách Javascript lưu dữ liệu.

Các kiểu dữ liệu nguyên thủy

Đây là các kiểu dữ liệu nguyên thủy:

- Number, ví dụ 1
- String, ví dụ 'Hello'
- Boolean, ví dụ true
- undefined
- null

Với các kiểu dữ liệu nguyên thủy, khi được gán giá trị sẽ được gắn chặt với biến, bạn sẽ không phải lo lắng về việc copy các biến này vì bạn sẽ luôn có được một ban sao thực thụ (biến mới tách biệt hoàn toàn so với biến cũ)

```
const a = 5
let b = a // tạo copy

b = 6

// thay đổi giá trị bản sao b không làm thay đổi giá trị "bản gốc" a (duocday)
console.log(b) // 6
console.log(a) // 5
```

Kiểu dữ liệu hỗn hợp (Object và Array)

Về mặt bản chất, array cũng là object, thế nên chúng cũng xảy ra vấn đề tương tự nhau trong trường hợp này

```
let a = []
typeof a // "object"
```

Có một điều thú vị, khi thực hiện copy object hoặc array vào một biến mới, biến đó sẽ chỉ tham chiếu của giá trị bạn đầu.

```
const a = {
  en: 'Hello',
  vi: 'Xin chào'
}

let b = a
b.vi = 'Chao xìn'
console.log(b.vi) // Chao xìn
console.log(a.vi) // Chao xìn
```

Một điều thú vị nữa, trong ví dụ trên dù a được định nghĩa là const nhưng ta vẫn thay đổi được giá trị, bạn có biết tại sao không? (dx) Quay lại, chúng ta thực sự đã tạo ra 1 shallow copy trong ví dụ trên. Điều này thường sẽ gây ra sai sót nếu chúng ta sử dụng không hợp lý, thay vì thay đổi giá trị của biến mới, chúng ta cũng làm giá trị của biến ban đầu thay đổi. Vậy làm thế nào để copy một cachs an toàn các Array và Object?

Object

Spread operator

Spread operator (Toán tử 3 chấm) là một điều tuyệt vời của ES6, bạn có thể sử dụng nó để deep copy một object như sau:

```
const a = {
  en: 'Hello',
  vi: 'Xin chào'
}

let b = {...a}
b.vi = 'Chao xìn'
console.log(b.vi) // Chao xìn
console.log(a.vi) // Xin chào
```

Bạn cũng có thể lưỡng long nhất thể 2 (hoặc nhiều hơn) object bằng spread operator:

```
const d = {...a, ...b, ...c}
```

Object.assign

Đây là một cách dùng phổ biến trước khi Spread operator được phát minh ra, cũng cho ra kết quả tương tự. Nhưng lưu ý, các thành phần của đối số đầu tiên vẫn có thể thay đổi được, thế nên object cần copy nên ở đối số thứ 2 trở đi, bạn có thể assign object cần copy với một object rỗng `{}`, đây là các thường dùng.

```
const a = {
  en: 'Hello',
  vi: 'Xin chào'
}

let b = Object.assign({}, a)
b.vi = 'Chao xìn'
console.log(b.vi) // Chao xìn
console.log(a.vi) // Xin chào
```

Cạm bẫy: Object lồng nhau

```
const a = {
  languages: {
    vi: 'Xin chào'
  }
}
let b = {...a}
b.languages.vi = 'Chao xìn'
console.log(b.languages.vi) // Chao xìn
console.log(a.languages.vi) // Chao xìn
```

OMG, nó thay đổi cả hai, để giải quyết vấn đề này, ta sử dụng JSON.parse(JSON.stringify(a)):

```
const a = {
  languages: {
    vi: 'Xin chào'
  }
}
let b = JSON.parse(JSON.stringify(a))
b.languages.vi = 'Chao xìn'
console.log(b.languages.vi) // Chao xìn
console.log(a.languages.vi) // Xin chào
```

Arrays

Như đã nói array thực chất cũng là object nên việc copy array cũng tương tự như bên trên:

Spread operator

```
const a = [1,2,3]
let b = [...a]
b[1] = 4
console.log(b[1]) // 4
console.log(a[1]) // 2
```

Các Array function: map, filter, reduce

Những phương thức này sẽ tạo một array mới chứa các giá trị của array cũ, bạn cũng có thể tùy chỉnh để tạo ra array mới có giá trị khác array cũ

```
const a = [1,2,3]
let b = a.map(el => el)
b[1] = 4
console.log(b[1]) // 4
console.log(a[1]) // 2
```

Thay đổi giá trị lúc copy:

```
const a = [1,2,3]
const b = a.map((el, index) => index === 1 ? 4 : el)
console.log(b[1]) // 4
console.log(a[1]) // 2
```

Array.slice

Phuowng thức này thường đực dùng để "cắt" array ra thành một array nhỏ hơn, nhưng bạn cũng có thể sử dụng `array.slice()` hoặc `array.slice(0)` để tạo ra một bản copy:

```
const a = [1,2,3]
let b = a.slice(0)
b[1] = 4
console.log(b[1]) // 4
console.log(a[1]) // 2
```

Array lồng nhau

Tương tự object, sử dụng `JSON.parse(JSON.stringify(array))`

Copy instance của Class

Class cũng là object, nhưng bạn không thể sử dụng stringify và parse để copy như một object bình thường được vì việc này sẽ làm mất đi các phương thức bên trong object của bạn. Để làm việc này, bạn có thể thêm một phương thức để trả về một instance mới với các giá trị ban đầu.

```
class Counter {
  constructor() {
     this.count = 5
  }
  copy() {
    const copy = new Counter()
    copy.count = this.count
    return copy
  }
}
const originalCounter = new Counter()
const copiedCounter = originalCounter.copy()
console.log(originalCounter.count) // 5
console.log(copiedCounter.count) // 5
copiedCounter.count = 7
console.log(originalCounter.count) // 5
console.log(copiedCounter.count) // 7
```

Shallow Copy

```
/*
    Shallow copy
    Original Object, Cloned Object => Referenced Object
*/
```

1. Example

```
const user = {
  id: 1,
  name: "Quang",
};

const user1 = user;

user.name = "John";
user1.age = 24;
user.contact = {
  phone: "0986915765",
};

console.log("user: ", user);
/**
  {
    id: 1,
    name: "John",
    age: 24,
    contact: {
      phone: "0986915765"
    }
  }
*/
console.log("user1: ", user1);
/**
  {
    id: 1,
    name: "John",
    age: 24,
    contact: {
      phone: "0986915765"
    }
  }
*/

console.log(user === user1); // Vì user1 đang copy tham chiếu của user.
```

2. Use spread operator

```
let person = {
  id: 1,
  name: 'Quang',
};

let copiedPerson = { ...person };
copiedPerson.name = 'John';

console.log('person: ', person);
// person:  { id: 1, name: 'Quang' }

console.log('copiedPerson: ', copiedPerson);
// copiedPerson:  { id: 1, name: 'John' }
```

Nested Object

```
let person = {
    id: 1,
    name: "Quang",
    contact: {
        email: "old@gmail.com",
    }
};

let copiedPerson = {...person};

copiedPerson.name = 'John';
copiedPerson.contact.email = "new@gmail.com";
```

```
console.log("person: ", person);
// person:  { id: 1, name: 'Quang', contact: { email: 'new@gmail.com' } }

console.log("copiedPerson: ", copiedPerson);
// copiedPerson:  { id: 1, name: 'John', contact: { email: 'new@gmail.com' } }
```

Use Object["*propertyName*"] - Override old property and **disconnect from the old memory area**

```
let person = {
    id: 1,
    name: "Quang",
    contact: {
        email: "old@gmail.com",
    }
};

let copiedPerson = {...person};

copiedPerson.name = 'John';
copiedPerson['contact'] = {
  email: 'new@gmail.com',
},

console.log("person: ", person);
// person:  { id: 1, name: 'Quang', contact: { email: 'old@gmail.com' } }

console.log("copiedPerson: ", copiedPerson);
// copiedPerson:  { id: 1, name: 'John', contact: { email: 'new@gmail.com' } }
```

3. Use Object.assign(target, source)

Similar to "spread operator"

JavaScript Demo: Object.assign():

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// Expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget === target);
// Expected output: true
```

Deep Copy

```
/*
    Deep Copy
    Original Object => Referenced Object
    Cloned Object => Referenced Clone
    - Khi dùng spread operator và object.assign => Deep copy chỉ có thể disconnect trong object có 1 cấp
*/
```

Primitive value:

```
const a = 5
let b = a

b = 6

console.log(b) // 6
console.log(a) // 5
```

Object:

```
const user = {
  id: 1,
  name: 'Quang',
  contact: {
```

```
    email: 'old@gmail.com',
  },
};

// Using stringify and parse
const userParsed = JSON.parse(JSON.stringify(user));

userParsed.name = 'Nam';
userParsed.contact.email = 'parsed@gmail.com';

console.log('user: ', user);
// user:  { id: 1, name: 'Quang', contact: { email: 'old@gmail.com' } }

console.log('userParsed: ', userParsed);
// userParsed:  { id: 1, name: 'Nam', contact: { email: 'parsed@gmail.com' } }
```

Array:

```
const array = [1, 2, 3, function abc() {}];
const arrayCopied = JSON.parse(JSON.stringify(array));

console.log('array: ', array);
// array:  [ 1, 2, 3, [Function: abc] ]
console.log('arrayCopied: ', arrayCopied);
// arrayCopied:  [ 1, 2, 3, null ]
// can'nt copy a function when using stringify and parse
```

9. Shallow Comparison & Deep Comparison

You most likely heard terms shallow comparison and deep comparison. In this post you will fully understand how it all works.

In this tutorial you will learn:- Problems of standard JavaScript comparison- Implement shallow comparison- Implement deep comparison- Compare all of them

> So just to be on the same page in JavaScript we have default comparison of the variables which can't solve some problems, shallow comparison and deep comparison.

Standard JavaScript comparison

Why it's not enough to use standard comparison of JavaScript? We use === to compare 2 properties

```
const a = 1
const b = 2

console.log(a === b)
```

And it will work fine for primitives like numbers, booleans, strings. But it won't work like you might expect for objects and arrays

```
const a = [1]
const b = [2]

console.log(a === b)
```

As you can see we got false because when we compare primitives JavaScript compares values and when we compare arrays or objects JavaScript compares if they are referenced the same object.

So actually what most people want when they write their code is to compare values inside arrays or objects.

This is the mistake that all beginners do. They want to copy object and change it's value but it doesn't work as intended.

```
const a = {name: 'foo'}
const b = a
b.age = 30
console.log(a, b)
```

As you can see this code modifies body `a` and `b` because they reference the same object in memory.

Shallow comparison

Which brings us to the shallow comparison. It's a custom function which can compare different data types. Most important point to remember that shallow equal is much faster than deep equal.

```
const typeOf = (input) => {
  const rawObject = Object.prototype.toString.call(input).toLowerCase();
  const typeOfRegex = /\[object (.*)]/g;
  const type = typeOfRegex.exec(rawObject)[1];
  return type;
};
```

So here I have a helper `typeOf` function which simply returns the type of data as a string. As you can see it returns correct types.

```
typeof('1'); // string
typeof(1); // number
typeof([1]); // object (array)
typeof({a: 1}); // object
```

Now let's create our shallow equality function.

First of all we want to check if 2 things that we compare are of different type. If it's different than they can't be equal.

```
const shallowCompare = (source, target) => {
  if (typeOf(source) !== typeOf(target)) {
    return false;
  }
};
```

Now we can simply add default Javascript comparison at the end of our function.

```
const shallowCompare = (source, target) => {
  if (typeOf(source) !== typeOf(target)) {
    return false;
  }

  return source === target;
};
```

So it already works for primitives. Now we need to add logic how we will check arrays. And we don't want to do any deep checking. We simply check if every element equals the same element in other array by using `every` function. But again it's plain Javascript we don't do any deep equal and this code won't work correctly with objects or arrays inside each array. But for primitives inside it will work correctly which is completely fine for us as we go for performance here.

```
const shallowCompare = (source, target) => {
  if (typeOf(source) !== typeOf(target)) {
    return false;
  }

  if (typeOf(source) === "array") {
    if (source.length !== target.length) {
      return false;
    }
    return source.every((el, index) => el === target[index]);
  }

  return source === target;
};
```

The next step is to compare objects. Again no deep comparison here, we simply go through keys and compare their values. If our object nested it won't be checked.

```
const shallowCompare = (source, target) => {
  if (typeOf(source) !== typeOf(target)) {
    return false;
  }

  if (typeOf(source) === "array") {
    if (source.length !== target.length) {
      return false;
    }
    return source.every((el, index) => el === target[index]);
  } else if (typeOf(source) === "object") {
    return Object.keys(source).every((key) => source[key] === target[key]);
  } else if (typeOf(source) === "date") {
    return source.getTime() === target.getTime();
  }

  return source === target;
};
```

And the last thing is to check dates.

```
const shallowCompare = (source, target) => {
  if (typeOf(source) !== typeOf(target)) {
    return false;
  }

  if (typeOf(source) === "array") {
    if (source.length !== target.length) {
      return false;
    }
    return source.every((el, index) => el === target[index]);
  } else if (typeOf(source) === "object") {
    return Object.keys(source).every((key) => source[key] === target[key]);
  } else if (typeOf(source) === "date") {
    return source.getTime() === target.getTime();
  }

  return source === target;
};
```

We convert dates to milliseconds and compare them.

And here is the usage of `shallowCompare`.

```
shallowCompare({a: 1}, {a: 1}) // true
```

So if now works how it should work by default in Javascript. But obviously deep comparison will fail here.

```
shallowCompare({a: {b: 1}}, {a: {b: 1}}) // false
```

Deep comparison

But sometimes we really want to compare big difficult nested objects or arrays. And it can't be fast. We should do it recursive and it will be slow. This is why I recommend you to avoid comparing all properties of huge objects when possible. I will copy parse our `shallowCompare` function completely as it is 99% the same code.

```
const deepCompare = (source, target) => {
  if (typeOf(source) !== typeOf(target)) {
    return false;
  }

  if (typeOf(source) === "array") {
    if (source.length !== target.length) {
      return false;
    }

    return source.every((entry, index) => deepCompare(entry, target[index]));
  } else if (typeOf(source) === "object") {
    if (Object.keys(source).length !== Object.keys(target).length) {
      return false;
    }
```

```
    return Object.keys(source).every((key) =>
      deepCompare(source[key], target[key])
    );
  } else if (typeOf(source) === "date") {
    return source.getTime() === target.getTime();
  }

  return source === target;
};
```

In deep comparison we do similar stuff but instead of using plain Javascript comparison we call our deepCompare function recursively. It means for example that if inside object we have a property which is an object we will start this function again and check the types of data inside and if it's an object again we will start deepCompare recursively again.

Exactly the same we do with array. We check recursively every single element.

Here is how our `deepCompare` is working.

```
deepCompare({a: {b: 1}}, {a: {b: 1}}) // true
```

As we compare every single value it is the most correct way of doing things.

Conclusion

So here is a conclusion. For primitives you can simply use plain Javascript. If you need fast comparison of simple arrays and objects use shallow comparison. If it's important for you to check every single key inside and compare them then use deep comparison but you need to be aware that it is the slowest way to compare things.

And you should not build it on your own in popular libraries like Lodash or Ramda or just as npm packages you can find both this implementations.

Manual comparison

```
/*
    Manual comparison: So sánh thủ công
    - Manual comparison is way to compare objects
    by content => is to read the properties and compare them manually.
*/

// Demo: Manual comparison

function isHeroEqual(object1, object2) {
  return object1.name === object2.name && object1.slug === object2.slug;
}

const hero1 = {
  slug: 'DX034',
  name: 'Batman',
};

const hero2 = {
  slug: 'DX034',
  name: 'Batman',
};

const hero3 = {
  slug: 'CI398',
  name: 'Joker',
};

console.log('hero1 and hero2: ', isHeroEqual(hero1, hero2)); // => true
console.log('hero1 and hero3: ', isHeroEqual(hero1, hero3)); // => false
```

Shallow comparison

```
/*
    Shallow comparison (equality): So sánh nông
    - During shallow equality check of objects you get the list of properties
    (using Object.keys()) of both objects, then check the properties' values for equality.
*/

// Demo: Shallow comparison (equality)
```

```
function shallowEqual(object1, object2) {
  const keys1 = Object.keys(object1);
  const keys2 = Object.keys(object2);

  if (keys1.length !== keys2.length) {
    return false;
  }

  for (let key of keys1) {
    if (object1[key] !== object2[key]) {
      return false;
    }
  }

  return true;
}

// Case 1: Object is not nested
const hero1 = {
  name: "Batman",
  realName: "Bruce Wayne",
};

const hero2 = {
  name: "Batman",
  realName: "Bruce Wayne",
};

const hero3 = {
  name: "Joker",
};

console.log("hero1 and hero2: ", shallowEqual(hero1, hero2)); // => true
console.log("hero1 and hero3: ", shallowEqual(hero1, hero3)); // => false

// Case 2: Object is nested
const hero4 = {
  name: "Batman",
  address: {
    city: "Gotham",
  },
};Object.entries

const hero5 = {
  name: "Batman",
  address: {
    city: "Gotham",
  },
};

console.log("hero4 and hero5: ", shallowEqual(hero4, hero5)); // => false
```

Deep comparison

```
/*
    Deep comparison (equality): So sánh sâu
    - The deep equality is similar to the shallow equality, but with one difference.
    During the shallow check, if the compared properties are objects,
    a recursive shallow equality check is performed on these nested objects.
*/

// Demo: Deep comparison (equality)

function deepEqual(object1, object2) {
  const keys1 = Object.keys(object1);
  const keys2 = Object.keys(object2);

  if (keys1.length !== keys2.length) {
    return false;
  }

  for (const key of keys1) {
    const val1 = object1[key];
    const val2 = object2[key];
    const areObjects = isObject(val1) && isObject(val2);

    if (
      (areObjects && !deepEqual(val1, val2)) ||
      (!areObjects && val1 !== val2)
    ) {
      return false;
    }
```

```
    // Explanation for the above "if" code

    // if (areObjects) {
    //     if (!deepEqual(val1, val2)) {
    //         return false;
    //     }
    // } else {
    //     if (val1 !== val2) {
    //         return false;
    //     }
    // }
  }

  return true;
}

function isObject(object) {
  return object != null && typeof object === 'object';
}

const hero1 = {
  name: 'Batman',
  address: {
    id: 1,
    city: 'Gotham',
    childObj: {
      value: 5,
      contact: {
        phone: '0987654321',
      },
    },
  },
  //   getName: function () {
  //     return this.name;
  //   },
};
const hero2 = {
  name: 'Batman',
  address: {
    id: 1,
    city: 'Gotham',
    childObj: {
      value: 5,
      contact: {
        phone: '0987654321',
      },
    },
  },
  //   getName: function () {
  //     return this.name;
  //   },
};

console.log(deepEqual(hero1, hero2)); // => true

// if both of 2 object contain method => false
```

Shallow Compare

> Note:
>
> `shallowCompare` is a legacy add-on. Use `React.memo` or `React.PureComponent` instead.

Importing

```
import shallowCompare from 'react-addons-shallow-compare'; // ES6
var shallowCompare = require('react-addons-shallow-compare'); // ES5 with npm
```

Overview

Before `React.PureComponent` was introduced, `shallowCompare` was commonly used to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

```
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update. `shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

Như chúng ta đã biết, một `component` sẽ *re-render* lại khi chúng ta update state. Khi sử dụng `Class Components` để update state, chúng ta dùng `this.setState()`, React sẽ so sánh state trước và sau, nếu state thay đổi thì lúc này React sẽ re-render component 👍.

Vậy làm thế nào để React biết state đã thay đổi so với trước kia? Câu trả lời là React thực hiện `Shallow Compare` cho state cũ với state mới.

Vậy Shallow Compare trong React hoạt động thế nào? Shallow Compare có thể hiểu là so sánh sử dụng `===`. Khi so sánh các `primitives` như `number`, `string` nó sẽ so sánh các giá trị của chúng. Tuy nhiên, khi so sánh các object, nó sẽ không so sánh các thuộc tính của chúng mà chỉ so sánh *reference* của chúng.

Giả sử ta có state sau:

```
this.state = {
  ten: "Trang",
  lop: "10A"
}

const user = this.state;
user.ten = "Linh";
```

Ta thử thay đổi giá trị `ten` của `user` và so sánh `state` vs `user` để xem kết quả. Bạn đoán xem kết quả sẽ là `true` hay `false`?

`console.log(user === this.state); // true`

Kết quả trả về là `true` 🤷‍♂️ Bởi vì khi chúng ta gán `this.state` cho `user` thì *reference* giống nhau, bạn có thể hiểu là chúng cùng trỏ về một địa chỉ trong bộ nhớ.

Cùng theo dõi ví dụ tiếp theo:

```
const user = {
    name: "Minh",
    born: "2000",
    phone: {
         viettel: 123,
         vinaphone: 423,
    }
}

const updated_user = {
    name: "Minh",
    born: "2000",
    phone: {
         viettel: 123,
         vinaphone: 423,
    }
}
```

Với ví dụ trên, chúng ta có thể hình dung React sẽ so sánh sử dụng `Shallow Compare` 2 object như thế này:

```
if(user.name !== updated_user.name
   || user.born !== updated_user.born
   || user.phone !== updated_user.phone
  ) {
  console.log('state được update giá trị mới nên sẽ re-render');
}
```

Ở ví dụ trên khi so sánh `user.phone !== updated_user.phone` sẽ có kết quả là `true`. Lý do thì như ở trên mình đã nói, đối với so sánh object khi sử dụng `===` nó sẽ so sánh theo *reference*. Vì 2 object này có địa chỉ trong ô nhớ khác nhau nên chúng sẽ khác nhau ^^.

`Shallow Compare` có nghĩa là nó sẽ chỉ so sánh các thuộc tính thuộc `first level` trong object, chứ không so sánh chi tiết các level khác. Ví dụ như thế này: `user.phone.viettel === updated_user.phone.viettel`, với kiểu so sánh thế này thì chúng ta sẽ gọi nó là `Deep Compare` hay so sánh tất tần tật những gì bên trong object 😁.

> Với Class Components extends React.Component, khi chúng ta update state sử dụng this.setState(), React sẽ không thực hiện Shallow Compare mà sẽ thực hiện gọi render() và re-render lại component.

Như lưu ý ở trên thì làm sao để giảm thiểu các re-render không cần thiết?

Thông thường chúng ta thường viết `Class Components` như thế này:

```
class MyComponent extends React.Component {
  //...
}
```

Nếu các bạn muốn React thực hiện `Shallow Compare` cho new/old props hay state thì có thể extends `PureComponent`:

```
class MyComponent extends React.PureComponent {
  //...
}
```

Nếu không thích `PureComponent` bạn có thể tự kiểm tra bằng cách sử dụng `shouldComponentUpdate()`, lifecycle method này sẽ ngăn không cho React re-render lại component khi method này return `false`. Chúng ta có thể kiểm tra các giá trị của state tại đây như sau:

```
shouldComponentUpdate(nextProps, nextState) {
  // Nếu không muốn component re-render bạn có thể return false luôn ^^.
  if(this.state.someVar === nextState.someVar) return false;
  return true;
}
```

> Tóm lại: Shallow Compare có nghĩa là khi so sánh 2 object(so sánh first level object) thì nó sẽ thực hiện kiểm tra xem hai giá trị có bằng nhau không trong trường hợp giá trị này thuộc primitive types như string, number. Trong trường hợp là object, nó chỉ kiểm tra tham chiếu(reference).Nếu chúng ta Shallow Compare một object phức tạp chứa object khác, nó sẽ chỉ kiểm tra tham chiếu chứ không phải các giá trị bên trong đối tượng đó.

10. Array

💡 **JavaScript Array:** In JavaScript, arrays aren't primitives but are instead `Array` objects with the following core characteristics:

- **JavaScript arrays are resizable** and **can contain a mix of different data types**. (When those characteristics are undesirable, use typed arrays instead.)
- **JavaScript arrays are not associative arrays** and so, array elements cannot be accessed using arbitrary strings as indexes, but must be accessed using nonnegative integers (or their respective string form) as indexes.
- **JavaScript arrays are zero-indexed**: the first element of an array is at index `0`, the second is at index `1`, and so on — and the last element is at the value of the array's `length` property minus `1`.
- **JavaScript array-copy operations create shallow copies**. (All standard built-in copy operations with *any* JavaScript objects create shallow copies, rather than deep copies).

JavaScript Arrays

An array is a special variable, which can hold more than one value:

```
const cars = ["Saab", "Volvo", "BMW"];
```

Why Use Arrays?

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
let car1 = "Saab";
let car2 = "Volvo";
let car3 = "BMW";
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Creating an Array

Using an array literal is the easiest way to create a JavaScript Array.

Syntax:

```
const array_name = [item1, item2, ...];
```

It is a common practice to declare arrays with the const keyword.

Learn more about const with arrays in the chapter: JS Array Const.

Example

```
const cars = ["Saab", "Volvo", "BMW"];
```

Spaces and line breaks are not important. A declaration can span multiple lines:

Example

```
const cars = [
  "Saab",
  "Volvo",
  "BMW"
];
```

You can also create an array, and then provide the elements:

Example

```
const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
```

Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

Example

```
const cars = new Array("Saab", "Volvo", "BMW");
```

The two examples above do exactly the same.

There is no need to use `new Array()`.

For simplicity, readability and execution speed, use the array literal method.

Accessing Array Elements

You access an array element by referring to the **index number**:

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];
```

Note: Array indexes start with 0.

[0] is the first element.

[1] is the second element.

Changing an Array Element

This statement changes the value of the first element in `cars`:

```
cars[0] = "Opel";
```

Example

```
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
```

Access the Full Array

With JavaScript, the full array can be accessed by referring to the array name:

Example

```
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
```

Arrays are Objects

Arrays are a special type of objects. The `typeof` operator in JavaScript returns "object" for arrays.

But, JavaScript arrays are best described as arrays.

Arrays use **numbers** to access its "elements". In this example, `person[0]` returns John:

Array:

```
const person = ["John", "Doe", 46];
```

Objects use **names** to access its "members". In this example, `person.firstName` returns John:

Object:

```
const person = {firstName:"John", lastName:"Doe", age:46};
```

Array Elements Can Be Objects

JavaScript variables can be objects. Arrays are special kinds of objects.

Because of this, you can have variables of different types in the same Array.

You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:

```
myArray[0] = Date.now;
myArray[1] = myFunction;
myArray[2] = myCars;
```

Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods:

cars.length // Returns the number of elementscars.sort() // Sorts the array

Array methods are covered in the next chapters.

The length Property

The `length` property of an array returns the length of an array (the number of array elements).

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.length;
```

The `length` property is always one more than the highest array index.

Accessing the First Array Element

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits[0];
```

Accessing the Last Array Element

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits[fruits.length - 1];
```

Looping Array Elements

One way to loop through an array, is using a `for` loop:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fLen = fruits.length;let text = "<ul>";

for (let i = 0; i < fLen; i++) {
  text += "<li>" + fruits[i] + "</li>";
}

text += "</ul>";
```

You can also use the `Array.forEach()` function:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let text = "<ul>";

fruits.forEach(myFunction);
text += "</ul>";

function myFunction(value) {
  text += "<li>" + value + "</li>";
}
```

Adding Array Elements

The easiest way to add a new element to an array is using the `push()` method:

Example

```
const fruits = ["Banana", "Orange", "Apple"];fruits.push("Lemon");  // Adds a new element (Lemon) to fruits
```

New element can also be added to an array using the `length` property:

Example

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[fruits.length] = "Lemon";  // Adds "Lemon" to fruits
```

WARNING !

Adding elements with high indexes can create undefined "holes" in an array:

Example

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[6] = "Lemon";  // Creates undefined "holes" in fruits
```

Associative Arrays

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** always use **numbered indexes**.

Example

```
const person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
person.length;    // Will return 3
person[0];        // Will return "John"
```

WARNING !!If you use named indexes, JavaScript will redefine the array to an object.

After that, some array methods and properties will produce **incorrect results**.

Example:

```
const person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
person.length;     // Will return 0
person[0];         // Will return undefined
```

The Difference Between Arrays and Objects

In JavaScript, **arrays** use **numbered indexes**.

In JavaScript, **objects** use **named indexes**.

Arrays are a special kind of objects, with numbered indexes.

When to Use Arrays. When to use Objects.

- JavaScript does not support associative arrays.
- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers**.

JavaScript new Array()

JavaScript has a built-in array constructor `new Array()`.

But you can safely use `[]` instead.

These two different statements both create a new empty array named points:

```
const points = new Array();
const points = [];
```

These two different statements both create a new array containing 6 numbers:

```
const points = new Array(40, 100, 1, 5, 25, 10);
const points = [40, 100, 1, 5, 25, 10];
```

The `new` keyword can produce some unexpected results:

```
// Create an array with three elements:
const points = new Array(40, 100, 1);
```

```
// Create an array with two elements:
const points = new Array(40, 100);
```

```
// Create an array with one element ???
const points = new Array(40);
```

A Common Error

```
const points = [40];

is not the same as:

const points = new Array(40);

// Create an array with one element:
const points = [40];
```

```
// Create an array with 40 undefined elements:
const points = new Array(40);
```

How to Recognize an Array

A common question is: How do I know if a variable is an array?

The problem is that the JavaScript operator `typeof` returns "`object`":

```
const fruits = ["Banana", "Orange", "Apple"];
let type = typeof fruits;
```

The typeof operator returns object because a JavaScript array is an object.

Solution 1:

To solve this problem ECMAScript 5 (JavaScript 2009) defined a new method `Array.isArray()`:

```
Array.isArray(fruits); // true
```

Solution 2:

The `instanceof` operator returns true if an object is created by a given constructor:

```
const fruits = ["Banana", "Orange", "Apple"];
fruits instanceof Array; //true
```

1. `Array.prototype.at()`
2. `Array.prototype.concat()`
3. `Array.prototype.copyWithin()`
4. `Array.prototype.entries()`
5. `Array.prototype.every()`
6. `Array.prototype.fill()`
7. `Array.prototype.filter()`
8. `Array.prototype.find()`
9. `Array.prototype.findIndex()`
10. `Array.prototype.findLast()`
11. `Array.prototype.findLastIndex()`
12. `Array.prototype.flat()`
13. `Array.prototype.flatMap()`
14. `Array.prototype.forEach()`
15. `Array.from()`
16. `Array.prototype.group()`
17. `Array.prototype.groupToMap()`
18. `Array.prototype.includes()`
19. `Array.prototype.indexOf()`
20. `Array.isArray()`
21. `Array.prototype.join()`
22. `Array.prototype.keys()`
23. `Array.prototype.lastIndexOf()`
24. `Array.prototype.map()`
25. `Array.of()`
26. `Array.prototype.pop()`

27. `Array.prototype.push()`
28. `Array.prototype.reduce()`
29. `Array.prototype.reduceRight()`
30. `Array.prototype.reverse()`
31. `Array.prototype.shift()`
32. `Array.prototype.slice()`
33. `Array.prototype.some()`
34. `Array.prototype.sort()`
35. `Array.prototype.splice()`
36. `Array.prototype.toLocaleString()`
37. `Array.prototype.toString()`
38. `Array.prototype.unshift()`
39. `Array.prototype.values()`

11. Function

Defining functions

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output. To use a function, you must define it somewhere in the scope from which you wish to call it.

Function declarations

A **function definition** (also called a **function declaration**, or **function statement**) consists of the `function` keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, `{ /* … */ }`.

For example, the following code defines a simple function named `square`:

```
function square(number) {
  return number * number;
}
```

The function `square` takes one parameter, called `number`. The function consists of one statement that says to return the parameter of the function (that is, `number`) multiplied by itself. The statement `return` specifies the value returned by the function:

```
return number * number;
```

Parameters are essentially passed to functions **by value** — so if the code within the body of a function assigns a completely new value to a parameter that was passed to the function, **the change is not reflected globally or in the code which called that function**.

When you pass an object as a parameter, if the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
  theObject.make = 'Toyota';
}
```

```
const mycar = {
  make: 'Honda',
  model: 'Accord',
  year: 1998,
};

// x gets the value "Honda"
const x = mycar.make;

// the make property is changed by the function
myFunc(mycar);
// y gets the value "Toyota"
const y = mycar.make;
```

When you pass an array as a parameter, if the function changes any of the array's values, that change is visible outside the function, as shown in the following example:

```
function myFunc(theArr) {
  theArr[0] = 30;
}

const arr = [45];

console.log(arr[0]); // 45
myFunc(arr);
console.log(arr[0]); // 30
```

Function expressions

While the function declaration above is syntactically a statement, functions can also be created by a function expression.

Such a function can be **anonymous**; it does not have to have a name. For example, the function `square` could have been defined as:

```
const square = function (number) {
  return number * number;
}
const x = square(4); // x gets the value 16
```

However, a name *can* be provided with a function expression. Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces:

```
const factorial = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
}

console.log(factorial(3))
```

Function expressions are convenient when passing a function as an argument to another function. The following example shows a `map` function that should receive a function as first argument and an array as second argument:

```
function map(f, a) {
  const result = new Array(a.length);
  for (let i = 0; i < a.length; i++) {
    result[i] = f(a[i]);
  }
  return result;
}
```

In the following code, the function receives a function defined by a function expression and executes it for every element of the array received as a second argument:

```
function map(f, a) {
  const result = new Array(a.length);
  for (let i = 0; i < a.length; i++) {
    result[i] = f(a[i]);
  }
  return result;
}
```

```
const f = function (x) {
  return x * x * x;
}

const numbers = [0, 1, 2, 5, 10];
const cube = map(f, numbers);
console.log(cube); // [0, 1, 8, 125, 1000]
```

Function returns: `[0, 1, 8, 125, 1000]`.

In JavaScript, a function can be defined based on a condition. For example, the following function definition defines `myFunc` only if `num` equals `0`:

```
let myFunc;
if (num === 0) {
  myFunc = function (theObject) {
    theObject.make = 'Toyota';
  }
}
```

In addition to defining functions as described here, you can also use the `Function` constructor to create functions from a string at runtime, much like `eval()`.

A **method** is a function that is a property of an object. Read more about objects and methods in Working with objects.

Calling functions

Defining a function does not *execute* it. Defining it names the function and specifies what to do when the function is called.

Calling the function actually performs the specified actions with the indicated parameters. For example, if you define the function `square`, you could call it as follows:

```
square(5);
```

The preceding statement calls the function with an argument of `5`. The function executes its statements and returns the value `25`.

Functions must be *in scope* when they are called, but the function declaration can be hoisted (appear below the call in the code). The scope of a function declaration is the function in which it is declared (or the entire program, if it is declared at the top level).

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function.
The `showProps()` function (defined in Working with objects) is an example of a function that takes an object as an argument.

A function can call itself. For example, here is a function that computes factorials recursively:

```
function factorial(n) {
  if (n === 0 || n === 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

You could then compute the factorials of `1` through `5` as follows:

```
const a = factorial(1); // a gets the value 1
const b = factorial(2); // b gets the value 2
const c = factorial(3); // c gets the value 6
const d = factorial(4); // d gets the value 24
const e = factorial(5); // e gets the value 120
```

There are other ways to call functions. There are often cases where a function needs to be called dynamically, or the number of arguments to a function vary, or in which the context of the function call needs to be set to a specific object determined at runtime.

It turns out that *functions are themselves objects* — and in turn, these objects have methods. (See the `Function` object.) The `call()` and `apply()` methods can be used to achieve this goal.

Function hoisting

Consider the example below:

```
console.log(square(5)); // 25

function square(n) {
  return n * n;
}
```

This code runs without any error, despite the `square()` function being called before it's declared. This is because the JavaScript interpreter hoists the entire function declaration to the top of the current scope, so the code above is equivalent to:

```
// All function declarations are effectively at the top of the scope
function square(n) {
  return n * n;
}

console.log(square(5)); // 25
```

Function hoisting only works with function *declarations* — not with function *expressions*. The code below will not work.

```
console.log(square); // ReferenceError: Cannot access 'square' before initialization
const square = function (n) {
  return n * n;
}
```

Function scope

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.

In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function, and any other variables to which the parent function has access.

```
// The following variables are defined in the global scope
const num1 = 20;
const num2 = 3;
const name = 'Chamakh';

// This function is defined in the global scope
function multiply() {
  return num1 * num2;
}

multiply(); // Returns 60

// A nested function example
function getScore() {
  const num1 = 2;
  const num2 = 3;

  function add() {
    return `${name} scored ${num1 + num2}`;
  }

  return add();
}

getScore(); // Returns "Chamakh scored 5"
```

Scope and the function stack

Recursion

A function can refer to and call itself. There are three ways for a function to refer to itself:

1. The function's name
2. `arguments.callee`
3. An in-scope variable that refers to the function

For example, consider the following function definition:

```
const foo = function bar() {
  // statements go here
}
```

Within the function body, the following are all equivalent:

1. `bar()`
2. `arguments.callee()`
3. `foo()`

A function that calls itself is called a *recursive function*. In some ways, recursion is analogous to a loop. Both execute the same code multiple times, and both require a condition (to avoid an infinite loop, or rather, infinite recursion in this case).

For example, consider the following loop:

```
let x = 0;
while (x < 10) { // "x < 10" is the loop condition
  // do stuff
  x++;
}
```

It can be converted into a recursive function declaration, followed by a call to that function:

```
function loop(x) {
  // "x >= 10" is the exit condition (equivalent to "!(x < 10)")
  if (x >= 10) {
    return;
  }
  // do stuff
  loop(x + 1); // the recursive call
}
loop(0);
```

However, some algorithms cannot be simple iterative loops. For example, getting all the nodes of a tree structure (such as the DOM) is easier via recursion:

```
function walkTree(node) {
  if (node === null) {
    return;
  }
  // do something with node
  for (let i = 0; i < node.childNodes.length; i++) {
    walkTree(node.childNodes[i]);
  }
}
```

Compared to the function `loop`, each recursive call itself makes many recursive calls here.

It is possible to convert any recursive algorithm to a non-recursive one, but the logic is often much more complex, and doing so requires the use of a stack.

In fact, recursion itself uses a stack: the function stack. The stack-like behavior can be seen in the following example:

```
function foo(i) {
  if (i < 0) {
```

```
    return;
  }
  console.log(`begin: ${i}`);
  foo(i - 1);
  console.log(`end: ${i}`);
}
foo(3);

// Logs:
// begin: 3
// begin: 2
// begin: 1
// begin: 0
// end: 0
// end: 1
// end: 2
// end: 3
```

Nested functions and closures

You may nest a function within another function. The nested (inner) function is private to its containing (outer) function.

It also forms a *closure*. A closure is an expression (most commonly, a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Since a nested function is a closure, this means that a nested function can "inherit" the arguments and variables of its containing function. In other words, the inner function contains the scope of the outer function.

To summarize:

- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

The following example shows nested functions:

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}
const a = addSquares(2, 3); // returns 13
const b = addSquares(3, 4); // returns 25
const c = addSquares(4, 5); // returns 41
```

Since the inner function forms a closure, you can call the outer function and specify arguments for both the outer and inner function:

```
function outside(x) {
  function inside(y) {
    return x + y;
  }
  return inside;
}
const fnInside = outside(3); // Think of it like: give me a function that adds 3 to whatever you give it
const result = fnInside(5); // returns 8
const result1 = outside(3)(5); // returns 8
```

Preservation of variables

Notice how `x` is preserved when `inside` is returned. A closure must preserve the arguments and variables in all scopes it references. Since each call provides potentially different arguments, a new closure is created for each call to `outside`. The memory can be freed only when the returned `inside` is no longer accessible.

This is not different from storing references in other objects, but is often less obvious because one does not set the references directly and cannot inspect them.

Multiply-nested functions

Functions can be multiply-nested. For example:

- A function (`A`) contains a function (`B`), which itself contains a function (`C`).
- Both functions `B` and `C` form closures here. So, `B` can access `A`, and `C` can access `B`.
- In addition, since `C` can access `B` which can access `A`, `C` can also access `A`.

Thus, the closures can contain multiple scopes; they recursively contain the scope of the functions containing it. This is called *scope chaining*. (The reason it is called "chaining" is explained later.)

Consider the following example:

```
function A(x) {
  function B(y) {
    function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
A(1); // Logs 6 (which is 1 + 2 + 3)
```

In this example, `C` accesses `B`'s `y` and `A`'s `x`.

This can be done because:

1. `B` forms a closure including `A` (i.e., `B` can access `A`'s arguments and variables).
2. `C` forms a closure including `B`.
3. Because `C`'s closure includes `B` and `B`'s closure includes `A`, then `C`'s closure also includes `A`. This means `C` can access *both* `B` *and* `A`'s arguments and variables. In other words, `C` *chains* the scopes of `B` and `A`, *in that order*.

The reverse, however, is not true. `A` cannot access `C`, because `A` cannot access any argument or variable of `B`, which `C` is a variable of. Thus, `C` remains private to only `B`.

Name conflicts

When two arguments or variables in the scopes of a closure have the same name, there is a *name conflict*. More nested scopes take precedence. So, the innermost scope takes the highest precedence, while the outermost scope takes the lowest. This is the scope chain. The first on the chain is the innermost scope, and the last is the outermost scope. Consider the following:

```
function outside() {
  const x = 5;
  function inside(x) {
    return x * 2;
  }
  return inside;
}

outside()(10); // returns 20 instead of 10
```

The name conflict happens at the statement `return x * 2` and is between `inside`'s parameter `x` and `outside`'s variable `x`. The scope chain here is {`inside`, `outside`, global object}. Therefore, `inside`'s `x` takes precedences over `outside`'s `x`, and `20` (`inside`'s `x`) is returned instead of `10` (`outside`'s `x`).

Closures

Closures are one of the most powerful features of JavaScript. JavaScript allows for the nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to).

However, the outer function does *not* have access to the variables and functions defined inside the inner function. This provides a sort of encapsulation for the variables of the inner function.

Also, since the inner function has access to the scope of the outer function, the variables and functions defined in the outer function will live longer than the duration of the outer function execution, if the inner function manages to survive beyond the life of the outer function. A closure is created when the inner function is somehow made available to any scope outside the outer function.

```
const pet = function (name) {   // The outer function defines a variable called "name"
  const getName = function () {
    // The inner function has access to the "name" variable of the outer function
    return name;
  }
  return getName; // Return the inner function, thereby exposing it to outer scopes
}
const myPet = pet('Vivie');

myPet(); // Returns "Vivie"
```

It can be much more complex than the code above. An object containing methods for manipulating the inner variables of the outer function can be returned.

```
const createPet = function (name) {
  let sex;

  const pet = {
    // setName(newName) is equivalent to setName: function (newName)
    // in this context
    setName(newName) {
      name = newName;
    },

    getName() {
      return name;
    },

    getSex() {
      return sex;
    },

    setSex(newSex) {
      if (typeof newSex === 'string' &&
        (newSex.toLowerCase() === 'male' || newSex.toLowerCase() === 'female')) {
        sex = newSex;
      }
    }
  };

  return pet;
}

const pet = createPet('Vivie');
pet.getName();                  // Vivie

pet.setName('Oliver');
pet.setSex('male');
pet.getSex();                   // male
pet.getName();                  // Oliver
```

In the code above, the `name` variable of the outer function is accessible to the inner functions, and there is no other way to access the inner variables except through the inner functions. The inner variables of the inner functions act as safe stores for the outer arguments and variables. They hold "persistent" and "encapsulated" data for the inner functions to work with. The functions do not even have to be assigned to a variable, or have a name.

```
const getCode = (function () {
  const apiCode = '0]Eal(eh&2';    // A code we do not want outsiders to be able to modify…

  return function () {
    return apiCode;
  };
})();

getCode();    // Returns the apiCode
```

Note: There are a number of pitfalls to watch out for when using closures!

If an enclosed function defines a variable with the same name as a variable in the outer scope, then there is no way to refer to the variable in the outer scope again. (The inner scope variable "overrides" the outer one, until the program exits the inner scope. It can be thought of as a name conflict.)

```
const createPet = function (name) {  // The outer function defines a variable called "name".
  return {
```

```
    setName(name) {    // The enclosed function also defines a variable called "name".
      name = name;              // How do we access the "name" defined by the outer function?
    }
  }
}
```

Using the arguments object

The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it as follows:

`arguments[i]`
Copy to Clipboard

where `i` is the ordinal number of the argument, starting at `0`. So, the first argument passed to a function would be `arguments[0]`. The total number of arguments is indicated by `arguments.length`.

Using the `arguments` object, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use `arguments.length` to determine the number of arguments actually passed to the function, and then access each argument using the `arguments` object.

For example, consider a function that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

```
function myConcat(separator) {
  let result = ''; // initialize list
  // iterate through arguments
  for (let i = 1; i < arguments.length; i++) {
    result += arguments[i] + separator;
  }
  return result;
}
```

You can pass any number of arguments to this function, and it concatenates each argument into a string "list":

```
// returns "red, orange, blue, "
myConcat(', ', 'red', 'orange', 'blue');

// returns "elephant; giraffe; lion; cheetah; "
myConcat('; ', 'elephant', 'giraffe', 'lion', 'cheetah');

// returns "sage. basil. oregano. pepper. parsley. "
myConcat('. ', 'sage', 'basil', 'oregano', 'pepper', 'parsley');
```

Note: The `arguments` variable is "array-like", but not an array. It is array-like in that it has a numbered index and a `length` property. However, it does *not* possess all of the array-manipulation methods.

See the `Function` object in the JavaScript reference for more information.

Function parameters

There are two special kinds of parameter syntax: *default parameters* and *rest parameters*.

Default parameters

In JavaScript, parameters of functions default to `undefined`. However, in some situations it might be useful to set a different default value. This is exactly what default parameters do.

In the past, the general strategy for setting defaults was to test parameter values in the body of the function and assign a value if they are `undefined`.

In the following example, if no value is provided for `b`, its value would be `undefined` when evaluating `a*b`, and a call to `multiply` would normally have returned `NaN`. However, this is prevented by the second line in this example:

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;
  return a * b;
}
```

```
multiply(5); // 5
```

With *default parameters*, a manual check in the function body is no longer necessary. You can put `1` as the default value for `b` in the function head:

```
function multiply(a, b = 1) {
  return a * b;
}

multiply(5); // 5
```

For more details, see default parameters in the reference.

Rest parameters

The rest parameter syntax allows us to represent an indefinite number of arguments as an array.

In the following example, the function `multiply` uses *rest parameters* to collect arguments from the second one to the end. The function then multiplies these by the first argument.

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map((x) => multiplier * x);
}

const arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Arrow functions

An arrow function expression (also called a *fat arrow* to distinguish from a hypothetical `->` syntax in future JavaScript) has a shorter syntax compared to function expressions and does not have its own `this`, `arguments`, `super`, or `new.target`. Arrow functions are always anonymous.

Two factors influenced the introduction of arrow functions: *shorter functions* and *non-binding* of `this`.

Shorter functions

In some functional patterns, shorter functions are welcome. Compare:

```
const a = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

const a2 = a.map(function(s) { return s.length; });

console.log(a2); // [8, 6, 7, 9]

const a3 = a.map((s) => s.length);

console.log(a3); // [8, 6, 7, 9]
```

No separate this

Until arrow functions, every new function defined its own `this` value (a new object in the case of a constructor, undefined in strict mode function calls, the base object if the function is called as an "object method", etc.). This proved to be less than ideal with an object-oriented style of programming.

```
function Person() {
  // The Person() constructor defines `this` as itself.
  this.age = 0;

  setInterval(function growUp() {
    // In nonstrict mode, the growUp() function defines `this`
    // as the global object, which is different from the `this`
```

```
    // defined by the Person() constructor.
    this.age++;
  }, 1000);
}

const p = new Person();
```

In ECMAScript 3/5, this issue was fixed by assigning the value in `this` to a variable that could be closed over.

```
function Person() {
  const self = this; // Some choose `that` instead of `self`.
                     // Choose one and be consistent.
  self.age = 0;

  setInterval(function growUp() {
    // The callback refers to the `self` variable of which
    // the value is the expected object.
    self.age++;
  }, 1000);
}
```

Alternatively, a bound function could be created so that the proper `this` value would be passed to the `growUp()` function.

An arrow function does not have its own `this`; the `this` value of the enclosing execution context is used. Thus, in the following code, the `this` within the function that is passed to `setInterval` has the same value as `this` in the enclosing function:

```
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++; // `this` properly refers to the person object
  }, 1000);
}

const p = new Person();
```

Predefined functions

JavaScript has several top-level, built-in functions:

`eval()`

The `eval()` method evaluates JavaScript code represented as a string.

`isFinite()`

The global `isFinite()` function determines whether the passed value is a finite number. If needed, the parameter is first converted to a number.

`isNaN()`

The `isNaN()` function determines whether a value is `NaN` or not. Note: coercion inside the `isNaN` function has interesting rules; you may alternatively want to use `Number.isNaN()` to determine if the value is Not-A-Number.

`parseFloat()`

The `parseFloat()` function parses a string argument and returns a floating point number.

`parseInt()`

The `parseInt()` function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).

`decodeURI()`

The `decodeURI()` function decodes a Uniform Resource Identifier (URI) previously created by `encodeURI` or by a similar routine.

`decodeURIComponent()`

The `decodeURIComponent()` method decodes a Uniform Resource Identifier (URI) component previously created by `encodeURIComponent` or by a similar routine.

`encodeURI()`

The `encodeURI()` method encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

`encodeURIComponent()`

The `encodeURIComponent()` method encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

`escape()`

The deprecated `escape()` method computes a new string in which certain characters have been replaced by a hexadecimal escape sequence. Use `encodeURI` or `encodeURIComponent` instead.

`unescape()`

The deprecated `unescape()` method computes a new string in which hexadecimal escape sequences are replaced with the character that it represents. The escape sequences might be introduced by a function like `escape`. Because `unescape()` is deprecated, use `decodeURI()` or `decodeURIComponent` instead.

12. This keyword

this

A function's `this` keyword behaves a little differently in JavaScript compared to other languages. It also has some differences between strict mode and non-strict mode.

In most cases, the value of `this` is determined by how a function is called (runtime binding). It can't be set by assignment during execution, and it may be different each time the function is called. The `bind()` method can set the value of a function's `this` regardless of how it's called, and arrow functions don't provide their own `this` binding (it retains the `this` value of the enclosing lexical context).

Syntax

```
this
```

Value

In non–strict mode, `this` is always a reference to an object. In strict mode, it can be any value. For more information on how the value is determined, see the description below.

Description

The value of `this` depends on in which context it appears: function, class, or global.

Function context

Inside a function, the value of `this` depends on how the function is called. Think about `this` as a hidden parameter of a function — just like the parameters declared in the function definition, `this` is a binding that the language creates for you when the function body is evaluated.

For a typical function, the value of `this` is the object that the function is accessed on. In other words, if the function call is in the form `obj.f()`, then `this` refers to `obj`. For example:

```
function getThis() {
  return this;
}

const obj1 = { name: "obj1" };
const obj2 = { name: "obj2" };

obj1.getThis = getThis;
obj2.getThis = getThis;

console.log(obj1.getThis()); // { name: 'obj1', getThis: [Function: getThis] }
console.log(obj2.getThis()); // { name: 'obj2', getThis: [Function: getThis] }
```

Note how the function is the same, but based on how it's invoked, the value of `this` is different. This is analogous to how function parameters work.

The value of `this` is not the object that has the function as an own property, but the object that is used to call the function. You can prove this by calling a method of an object up in the prototype chain.

```
const obj3 = {
  __proto__: obj1,
  name: "obj3",
};

console.log(obj3.getThis()); // { name: 'obj3' }
```

The value of `this` always changes based on how a function is called, even when the function was defined on an object at creation:

```
const obj4 = {
  name: "obj4",
  getThis() {
    return this;
  },
};

const obj5 = { name: "obj5" };

obj5.getThis = obj4.getThis;
console.log(obj5.getThis()); // { name: 'obj5', getThis: [Function: getThis] }
```

If the value that the method is accessed on is a primitive, `this` will be a primitive value as well — but only if the function is in strict mode.

```
function getThisStrict() {
  "use strict"; // Enter strict mode
  return this;
}

// Only for demonstration — you should not mutate built-in prototypes
Number.prototype.getThisStrict = getThisStrict;
console.log(typeof (1).getThisStrict()); // "number"
```

If the function is called without being accessed on anything, `this` will be `undefined` — but only if the function is in strict mode.

```
console.log(typeof getThisStrict()); // "undefined"
```

In non-strict mode, a special process called `this` substitution ensures that the value of `this` is always an object. This means:

- If a function is called with `this` set to `undefined` or `null`, `this` gets substituted with `globalThis`.
- If the function is called with `this` set to a primitive value, `this` gets substituted with the primitive value's wrapper object.

```
function getThis() {
  return this;
}

// Only for demonstration — you should not mutate built-in prototypes
Number.prototype.getThis = getThis;
console.log(typeof (1).getThis()); // "object"
console.log(getThis() === globalThis); // true
```

In typical function calls, `this` is implicitly passed like a parameter through the function's prefix (the part before the dot). You can also explicitly set the value of `this` using the `Function.prototype.call()`, `Function.prototype.apply()`, or `Reflect.apply()` methods. Using `Function.prototype.bind()`, you can create a new function with a specific value of `this` that doesn't change regardless of how the function is called. When using these methods, the `this` substitution rules above still apply if the function is non-strict.

Callbacks

When a function is passed as a callback, the value of `this` depends on how the callback is called, which is determined by the implementor of the API. Callbacks are *typically* called with a `this` value of `undefined` (calling it directly without attaching it to any object), which means if the function is non–strict, the value of `this` is the global object (`globalThis`). This is the case for iterative array methods, the `Promise()` constructor, `setTimeout()`, etc.

```
function logThis() {
  "use strict";
  console.log(this);
}

[1, 2, 3].forEach(logThis); // undefined, undefined, undefined
setTimeout(logThis, 1000); // undefined
```

Some APIs allow you to set a `this` value for invocations of the callback. For example, all iterative array methods and related ones like `Set.prototype.forEach()` accept an optional `thisArg` parameter.

```
[1, 2, 3].forEach(logThis, { name: "obj" });
// { name: 'obj' }, { name: 'obj' }, { name: 'obj' }
```

Occasionally, a callback is called with a `this` value other than `undefined`. For example, the `reviver` parameter of `JSON.parse()` and the `replacer` parameter of `JSON.stringify()` are both called with `this` set to the object that the property being parsed/serialized belongs to.

Arrow functions

In arrow functions, `this` retains the value of the enclosing lexical context's `this`. In other words, when evaluating an arrow function's body, the language does not create a new `this` binding.

For example, in global code, `this` is always `globalThis` regardless of strictness, because of the global context binding:

```
const globalObject = this;
const foo = () => this;
console.log(foo() === globalObject); // true
```

Arrow functions create a closure over the `this` value of its surrounding scope, which means arrow functions behave as if they are "auto-bound" — no matter how it's invoked, `this` is set to what it was when the function was created (in the example above, the global object). The same applies to arrow functions created inside other functions: their `this` remains that of the enclosing lexical context. See example below.

Furthermore, when invoking arrow functions using `call()`, `bind()`, or `apply()`, the `thisArg` parameter is ignored. You can still pass other arguments using these methods, though.

```
const obj = { name: "obj" };

// Attempt to set this using call
console.log(foo.call(obj) === globalObject); // true

// Attempt to set this using bind
const boundFoo = foo.bind(obj);
console.log(boundFoo() === globalObject); // true
```

Constructors

When a function is used as a constructor (with the `new` keyword), its `this` is bound to the new object being constructed, no matter which object the constructor function is accessed on. The value of `this` becomes the value of the `new` expression unless the constructor returns another non–primitive value.

```
function C() {
  this.a = 37;
}

let o = new C();
console.log(o.a); // 37

function C2() {
  this.a = 37;
```

```
  return { a: 38 };
}

o = new C2();
console.log(o.a); // 38
```

In the second example (`C2`), because an object was returned during construction, the new object that `this` was bound to gets discarded. (This essentially makes the statement `this.a = 37;` dead code. It's not exactly dead because it gets executed, but it can be eliminated with no outside effects.)

super

When a function is invoked in the `super.method()` form, the `this` inside the `method` function is the same value as the `this` value around the `super.method()` call, and is generally not equal to the object that `super` refers to. This is because `super.method` is not an object member access like the ones above — it's a special syntax with different binding rules. For examples, see the `super` reference.

Class context

A class can be split into two contexts: static and instance. Constructors, methods, and instance field initializers (public or private) belong to the instance context. Static methods, static field initializers, and static initialization blocks belong to the static context. The `this` value is different in each context.

Class constructors are always called with `new`, so their behavior is the same as function constructors: the `this` value is the new instance being created. Class methods behave like methods in object literals — the `this` value is the object that the method was accessed on. If the method is not transferred to another object, `this` is generally an instance of the class.

Static methods are not properties of `this`. They are properties of the class itself. Therefore, they are generally accessed on the class, and `this` is the value of the class (or a subclass). Static initialization blocks are also evaluated with `this` set to the current class.

Field initializers are also evaluated in the context of the class. Instance fields are evaluated with `this` set to the instance being constructed. Static fields are evaluated with `this` set to the current class. This is why arrow functions in field initializers are bound to the class.

```
class C {
  instanceField = this;
  static staticField = this;
}

const c = new C();
console.log(c.instanceField === c); // true
console.log(C.staticField === C); // true
```

Derived class constructors

Unlike base class constructors, derived constructors have no initial `this` binding. Calling `super()` creates a `this` binding within the constructor and essentially has the effect of evaluating the following line of code, where `Base` is the base class:

```
this = new Base();
```

Warning: Referring to `this` before calling `super()` will throw an error.

Derived classes must not return before calling `super()`, unless the constructor returns an object (so the `this` value is overridden) or the class has no constructor at all.

```
class Base {}
class Good extends Base {}
class AlsoGood extends Base {
  constructor() {
    return { a: 5 };
  }
}
class Bad extends Base {
  constructor() {}
}

new Good();
```

```
new AlsoGood();
new Bad(); // ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived construct
```

Global context

In the global execution context (outside of any functions or classes; may be inside blocks or arrow functions defined in the global scope), the `this` value depends on what execution context the script runs in. Like callbacks, the `this` value is determined by the runtime environment (the caller).

At the top level of a script, `this` refers to `globalThis` whether in strict mode or not. This is generally the same as the global object — for example, if the source is put inside an HTML `<script>` element and executed as a script, `this === window`.

Note: `globalThis` is generally the same concept as the global object (i.e. adding properties to `globalThis` makes them global variables) — this is the case for browsers and Node — but hosts are allowed to provide a different value for `globalThis` that's unrelated to the global object.

```
// In web browsers, the window object is also the global object:
console.log(this === window); // true

this.b = "MDN";
console.log(window.b); // "MDN"
console.log(b); // "MDN"
```

If the source is loaded as a module (for HTML, this means adding `type="module"` to the `<script>` tag), `this` is always `undefined` at the top level.

If the source is executed with `eval()`, `this` is the same as the enclosing context for direct eval, or `globalThis` (as if it's run in a separate global script) for indirect eval.

```
function test() {
  // Direct eval
  console.log(eval("this") === this);
  // Indirect eval, non-strict
  console.log(eval?.("this") === globalThis);
  // Indirect eval, strict
  console.log(eval?.("'use strict'; this") === globalThis);
}

test.call({ name: "obj" }); // Logs 3 "true"
```

Note that some source code, while looking like the global scope, is actually wrapped in a function when executed. For example, Node.js CommonJS modules are wrapped in a function and executed with the `this` value set to `module.exports`. Event handler attributes are executed with `this` set to the element they are attached to.

Object literals don't create a `this` scope — only functions (methods) defined within the object do. Using `this` in an object literal inherits the value from the surrounding scope.

```
const obj = {
  a: this,
};

console.log(obj.a === window); // true
```

Examples

this in function contexts

The value of `this` depends on how the function is called, not how it's defined.

```
// An object can be passed as the first argument to call
// or apply and this will be bound to it.
const obj = { a: "Custom" };

// Variables declared with var become properties of the global object.
var a = "Global";

function whatsThis() {
  return this.a; // The value of this is dependent on how the function is called
}
```

```
whatsThis(); // 'Global'; this in the function isn't set, so it defaults to the global/window object in non-strict mode
obj.whatsThis = whatsThis;
obj.whatsThis(); // 'Custom'; this in the function is set to obj
```

Using `call()` and `apply()`, you can pass the value of `this` as if it's an actual parameter.

```
function add(c, d) {
  return this.a + this.b + c + d;
}

const o = { a: 1, b: 3 };

// The first parameter is the object to use as 'this'; subsequent
// parameters are used as arguments in the function call
add.call(o, 5, 7); // 16

// The first parameter is the object to use as 'this', the second is an
// array whose members are used as arguments in the function call
add.apply(o, [10, 20]); // 34
```

this and object conversion

In non–strict mode, if a function is called with a `this` value that's not an object, the `this` value is substituted with an object. `null` and `undefined` become `globalThis`. Primitives like `7` or `'foo'` are converted to an object using the related constructor, so the primitive number `7` is converted to a `Number` wrapper class and the string `'foo'` to a `String` wrapper class.

```
function bar() {
  console.log(Object.prototype.toString.call(this));
}

bar.call(7); // [object Number]
bar.call("foo"); // [object String]
bar.call(undefined); // [object Window]
```

The bind() method

Calling `f.bind(someObject)` creates a new function with the same body and scope as `f`, but the value of `this` is permanently bound to the first argument of `bind`, regardless of how the function is being called.

```
function f() {
  return this.a;
}

const g = f.bind({ a: "azerty" });
console.log(g()); // azerty

const h = g.bind({ a: "yoo" }); // bind only works once!
console.log(h()); // azerty

const o = { a: 37, f, g, h };
console.log(o.a, o.f(), o.g(), o.h()); // 37,37, azerty, azerty
```

this in arrow functions

Arrow functions create closures over the `this` value of the enclosing execution context. In the following example, we create `obj` with a method `getThisGetter` that returns a function that returns the value of `this`. The returned function is created as an arrow function, so its `this` is permanently bound to the `this` of its enclosing function. The value of `this` inside `getThisGetter` can be set in the call, which in turn sets the return value of the returned function.

```
const obj = {
  getThisGetter() {
    const getter = () => this;
    return getter;
  },
};
```

We can call `getThisGetter` as a method of `obj`, which sets `this` inside the body to `obj`. The returned function is assigned to a variable `fn`. Now, when calling `fn`, the value of `this` returned is still the one set by the call to `getThisGetter`, which is `obj`. If

the returned function is not an arrow function, such calls would cause the `this` value to be `globalThis` or `undefined` in strict mode.

```
const fn = obj.getThisGetter();
console.log(fn() === obj); // true
```

But be careful if you unbind the method of `obj` without calling it, because `getThisGetter` is still a method that has a varying `this` value. Calling `fn2()()` in the following example returns `globalThis`, because it follows the `this` from `fn2`, which is `globalThis` since it's called without being attached to any object.

```
const fn2 = obj.getThisGetter;
console.log(fn2()() === globalThis); // true
```

This behavior is very useful when defining callbacks. Usually, each function expression creates its own `this` binding, which shadows the `this` value of the upper scope. Now, you can define functions as arrow functions if you don't care about the `this` value, and only create `this` bindings where you do (e.g. in class methods). See example with `setTimeout()`.

this with a getter or setter

`this` in getters and setters is based on which object the property is accessed on, not which object the property is defined on. A function used as getter or setter has its `this` bound to the object from which the property is being set or gotten.

```
function sum() {
  return this.a + this.b + this.c;
}

const o = {
  a: 1,
  b: 2,
  c: 3,
  get average() {
    return (this.a + this.b + this.c) / 3;
  },
};

Object.defineProperty(o, "sum", {
  get: sum,
  enumerable: true,
  configurable: true,
});

console.log(o.average, o.sum); // 2, 6
```

As a DOM event handler

When a function is used as an event handler, its `this` is set to the element on which the listener is placed (some browsers do not follow this convention for listeners added dynamically with methods other than `addEventListener()`).

```
// When called as a listener, turns the related element blue
function bluify(e) {
  // Always true
  console.log(this === e.currentTarget);
  // true when currentTarget and target are the same object
  console.log(this === e.target);
  this.style.backgroundColor = "#A5D9F3";
}

// Get a list of every element in the document
const elements = document.getElementsByTagName("*");

// Add bluify as a click listener so when the
// element is clicked on, it turns blue
for (const element of elements) {
  element.addEventListener("click", bluify, false);
}
```

this in inline event handlers

When the code is called from an inline event handler attribute, its `this` is set to the DOM element on which the listener is placed:

```
<button onclick="alert(this.tagName.toLowerCase());">Show this</button>
```

The above alert shows `button`. Note, however, that only the outer code has its `this` set this way:

```
<button onclick="alert((function () { return this; })());">
  Show inner this
</button>
```

In this case, the inner function's `this` isn't set, so it returns the global/window object (i.e. the default object in non–strict mode where `this` isn't set by the call).

Bound methods in classes

Just like with regular functions, the value of `this` within methods depends on how they are called. Sometimes it is useful to override this behavior so that `this` within classes always refers to the class instance. To achieve this, bind the class methods in the constructor:

```
class Car {
  constructor() {
    // Bind sayBye but not sayHi to show the difference
    this.sayBye = this.sayBye.bind(this);
  }
  sayHi() {
    console.log(`Hello from ${this.name}`);
  }
  sayBye() {
    console.log(`Bye from ${this.name}`);
  }
  get name() {
    return "Ferrari";
  }
}

class Bird {
  get name() {
    return "Tweety";
  }
}

const car = new Car();
const bird = new Bird();

// The value of 'this' in methods depends on their caller
car.sayHi(); // Hello from Ferrari
bird.sayHi = car.sayHi;
bird.sayHi(); // Hello from Tweety

// For bound methods, 'this' doesn't depend on the caller
bird.sayBye = car.sayBye;
bird.sayBye(); // Bye from Ferrari
```

Note: Classes are always in strict mode. Calling methods with an undefined `this` will throw an error if the method tries to access properties on `this`.

Note, however, that auto-bound methods suffer from the same problem as using arrow functions for class properties: each instance of the class will have its own copy of the method, which increases memory usage. Only use it where absolutely necessary. You can also mimic the implementation of `Intl.NumberFormat.prototype.format()`: define the property as a getter that returns a bound function when accessed and saves it, so that the function is only created once and only created when necessary.

this in with statements

Although `with` statements are deprecated and not available in strict mode, they still serve as an exception to the normal `this` binding rules. If a function is called within a `with` statement and that function is a property of the scope object, the `this` value is set to the scope object, as if the `obj1.` prefix exists.

```
const obj1 = {
  foo() {
    return this;
  },
};
```

```
with (obj1) {
  console.log(foo() === obj1); // true
}
```

13. Callback, Promise & Async/Await

Understanding the Event Loop, Callbacks, Promises, and Async/Await in JavaScript

Introduction

In the early days of the internet, websites often consisted of static data in an HTML page. But now that web applications have become more interactive and dynamic, it has become increasingly necessary to do intensive operations like make external network requests to retrieve API data. To handle these operations in JavaScript, a developer must use *asynchronous programming* techniques.

Since JavaScript is a *single-threaded* programming language with a *synchronous* execution model that processes one operation after another, it can only process one statement at a time. However, an action like requesting data from an API can take an indeterminate amount of time, depending on the size of data being requested, the speed of the network connection, and other factors. If API calls were performed in a synchronous manner, the browser would not be able to handle any user input, like scrolling or clicking a button, until that operation completes. This is known as *blocking*.

In order to prevent blocking behavior, the browser environment has many Web APIs that JavaScript can access that are *asynchronous*, meaning they can run in parallel with other operations instead of sequentially. This is useful because it allows the user to continue using the browser normally while the asynchronous operations are being processed.

As a JavaScript developer, you need to know how to work with asynchronous Web APIs and handle the response or error of those operations. In this article, you will learn about the event loop, the original way of dealing with asynchronous behavior through callbacks, the updated ECMAScript 2015 addition of promises, and the modern practice of using `async/await`.

Note: This article is focused on client-side JavaScript in the browser environment. The same concepts are generally true in the Node.js environment, however Node.js uses its own C++ APIs as opposed to the browser's Web APIs. For more information on asynchronous programming in Node.js, check out How To Write Asynchronous Code in Node.js.

The Event Loop

This section will explain how JavaScript handles asynchronous code with the event loop. It will first run through a demonstration of the event loop at work, and will then explain the two elements of the event loop: the stack and the queue.

JavaScript code that does not use any asynchronous Web APIs will execute in a synchronous manner—one at a time, sequentially. This is demonstrated by this example code that calls three functions that each print a number to the console:

```
// Define three example functions
function first() {
  console.log(1)
}

function second() {
  console.log(2)
}

function third() {
  console.log(3)
}
```

In this code, you define three functions that print numbers with `console.log()`.

Next, write calls to the functions:

```
// Execute the functions
first()
second()
third()
```

The output will be based on the order the functions were called—`first()`, `second()`, then `third()`:

```
Output
1
2
3
```

When an asynchronous Web API is used, the rules become more complicated. A built-in API that you can test this with is `setTimeout`, which sets a timer and performs an action after a specified amount of time. `setTimeout` needs to be asynchronous, otherwise the entire browser would remain frozen during the waiting, which would result in a poor user experience.

Add `setTimeout` to the `second` function to simulate an asynchronous request:

```
// Define three example functions, but one of them contains asynchronous code
function first() {
  console.log(1)
}

function second() {
  setTimeout(() => {
    console.log(2)
  }, 0)
}

function third() {
  console.log(3)
}
```

`setTimeout` takes two arguments: the function it will run asynchronously, and the amount of time it will wait before calling that function. In this code you wrapped `console.log` in an anonymous function and passed it to `setTimeout`, then set the function to run after `0` milliseconds.

Now call the functions, as you did before:

```
// Execute the functions
first()
second()
third()
```

You might expect with a `setTimeout` set to `0` that running these three functions would still result in the numbers being printed in sequential order. But because it is asynchronous, the function with the timeout will be printed last:

```
Output
1
3
2
```

Whether you set the timeout to zero seconds or five minutes will make no difference—the `console.log` called by asynchronous code will execute after the synchronous top-level functions. This happens because the JavaScript host environment, in this case the browser, uses a concept called the *event loop* to handle concurrency, or parallel events. Since JavaScript can only execute one statement at a time, it needs the event loop to be informed of when to execute which specific statement. The event loop handles this with the concepts of a *stack* and a *queue*.

Stack

The *stack*, or call stack, holds the state of what function is currently running. If you're unfamiliar with the concept of a stack, you can imagine it as an array with "Last in, first out" (LIFO) properties, meaning you can only add or remove items from the end of the stack. JavaScript will run the current *frame* (or function call in a specific environment) in the stack, then remove it and move on to the next one.

For the example only containing synchronous code, the browser handles the execution in the following order:

- Add `first()` to the stack, run `first()` which logs `1` to the console, remove `first()` from the stack.
- Add `second()` to the stack, run `second()` which logs `2` to the console, remove `second()` from the stack.
- Add `third()` to the stack, run `third()` which logs `3` to the console, remove `third()` from the stack.

The second example with `setTimout` looks like this:

- Add `first()` to the stack, run `first()` which logs `1` to the console, remove `first()` from the stack.
- Add `second()` to the stack, run `second()`.
  - Add `setTimeout()` to the stack, run the `setTimeout()` Web API which starts a timer and adds the anonymous function to the *queue*, remove `setTimeout()` from the stack.
- Remove `second()` from the stack.
- Add `third()` to the stack, run `third()` which logs `3` to the console, remove `third()` from the stack.
- The event loop checks the queue for any pending messages and finds the anonymous function from `setTimeout()`, adds the function to the stack which logs `2` to the console, then removes it from the stack.

Using `setTimeout`, an asynchronous Web API, introduces the concept of the *queue*, which this tutorial will cover next.

Queue

The *queue*, also referred to as message queue or task queue, is a waiting area for functions. Whenever the call stack is empty, the event loop will check the queue for any waiting messages, starting from the oldest message. Once it finds one, it will add it to the stack, which will execute the function in the message.

In the `setTimeout` example, the anonymous function runs immediately after the rest of the top-level execution, since the timer was set to `0` seconds. It's important to remember that the timer does not mean that the code will execute in exactly `0` seconds or whatever the specified time is, but that it will add the anonymous function to the queue in that amount of time. This queue system exists because if the timer were to add the anonymous function directly to the stack when the timer finishes, it would interrupt whatever function is currently running, which could have unintended and unpredictable effects.

Note: There is also another queue called the *job queue* or *microtask queue* that handles promises. Microtasks like promises are handled at a higher priority than macrotasks like `setTimeout`.

Now you know how the event loop uses the stack and queue to handle the execution order of code. The next task is to figure out how to control the order of execution in your code. To do this, you will first learn about the original way to ensure asynchronous code is handled correctly by the event loop: callback functions.

Callback Functions

In the `setTimeout` example, the function with the timeout ran after everything in the main top-level execution context. But if you wanted to ensure one of the functions, like the `third` function, ran after the timeout, then you would have to use asynchronous coding methods. The timeout here can represent an asynchronous API call that contains data. You want to work with the data from the API call, but you have to make sure the data is returned first.

The original solution to dealing with this problem is using *callback functions*. Callback functions do not have special syntax; they are just a function that has been passed as an argument to another function. The function that takes another function as an argument is called a *higher-order function*. According to this definition, any function can become a callback function if it is passed as an argument. Callbacks are not asynchronous by nature, but can be used for asynchronous purposes.

Here is a syntactic code example of a higher-order function and a callback:

```
// A function
function fn() {
  console.log('Just a function')
}

// A function that takes another function as an argument
function higherOrderFunction(callback) {
  // When you call a function that is passed as an argument, it is referred to as a callback
  callback()
}

// Passing a function
higherOrderFunction(fn)
```

In this code, you define a function `fn`, define a function `higherOrderFunction` that takes a function `callback` as an argument, and pass `fn` as a callback to `higherOrderFunction`.

Running this code will give the following:

```
Output
Just a function
```

Let's go back to the `first`, `second`, and `third` functions with `setTimeout`. This is what you have so far:

```
function first() {
  console.log(1)
}

function second() {
  setTimeout(() => {
    console.log(2)
  }, 0)
}

function third() {
  console.log(3)
}
```

The task is to get the `third` function to always delay execution until after the asynchronous action in the `second` function has completed. This is where callbacks come in. Instead of executing `first`, `second`, and `third` at the top-level of execution, you will pass the `third` function as an argument to `second`. The `second` function will execute the callback after the asynchronous action has completed.

Here are the three functions with a callback applied:

```
// Define three functions
function first() {
  console.log(1)
}

function second(callback) {
  setTimeout(() => {
    console.log(2)

    // Execute the callback function
callback()}, 0)
}

function third() {
  console.log(3)
}
```

Now, execute `first` and `second`, then pass `third` as an argument to `second`:

```
first()
second(third)
```

After running this code block, you will receive the following output:

```
Output
1
2
3
```

First `1` will print, and after the timer completes (in this case, zero seconds, but you can change it to any amount) it will print `2` then `3`. By passing a function as a callback, you've successfully delayed execution of the function until the asynchronous Web API (`setTimeout`) completes.

The key takeaway here is that callback functions are not asynchronous— `setTimeout` is the asynchronous Web API responsible for handling asynchronous tasks. The callback just allows you to be informed of when an asynchronous task has completed and handles the success or failure of the task.

Now that you have learned how to use callbacks to handle asynchronous tasks, the next section explains the problems of nesting too many callbacks and creating a "pyramid of doom."

Nested Callbacks and the Pyramid of Doom

Callback functions are an effective way to ensure delayed execution of a function until another one completes and returns with data. However, due to the nested nature of callbacks, code can end up getting messy if you have a lot of consecutive asynchronous requests that rely on each other. This was a big frustration for JavaScript developers early on, and as a result code containing nested callbacks is often called the “pyramid of doom” or “callback hell.”

Here is a demonstration of nested callbacks:

```
function pyramidOfDoom() {
  setTimeout(() => {
    console.log(1)
    setTimeout(() => {
      console.log(2)
      setTimeout(() => {
        console.log(3)
      }, 500)
    }, 2000)
  }, 1000)
}
```

In this code, each new `setTimeout` is nested inside a higher order function, creating a pyramid shape of deeper and deeper callbacks. Running this code would give the following:

```
Output
1
2
3
```

In practice, with real world asynchronous code, this can get much more complicated. You will most likely need to do error handling in asynchronous code, and then pass some data from each response onto the next request. Doing this with callbacks will make your code difficult to read and maintain.

Here is a runnable example of a more realistic “pyramid of doom” that you can play around with:

```
// Example asynchronous function
function asynchronousRequest(args, callback) {
  // Throw an error if no arguments are passed
  if (!args) {
    return callback(new Error('Whoa! Something went wrong.'))
  } else {
    return setTimeout(
      () => callback(null, {body: args + ' ' + Math.floor(Math.random() * 10)}),
      500,
    )
    /**
      - Just adding in a random number so it seems like the contrived
      asynchronous function
      - Returned different data
    */
  }
}

// Nested asynchronous requests
function callbackHell() {
  asynchronousRequest('First', function first(error, response) {
    if (error) {
      console.log(error)
      return
    }
    console.log(response.body)
    asynchronousRequest('Second', function second(error, response) {
      if (error) {
        console.log(error)
        return
      }
      console.log(response.body)
      asynchronousRequest(null, function third(error, response) {
        if (error) {
          console.log(error)
          return
        }
        console.log(response.body)
      })
    })
  })
}
```

```
// Execute
callbackHell()
```

In this code, you must make every function account for a possible `response` and a possible `error`, making the function `callbackHell` visually confusing.

Running this code will give you the following:

```
Output

First 9
Second 3
Error: Whoa! Something went wrong.
    at asynchronousRequest (<anonymous>:4:21)
    at second (<anonymous>:29:7)
    at <anonymous>:9:13
```

This way of handling asynchronous code is difficult to follow. As a result, the concept of *promises* was introduced in ES6. This is the focus of the next section.

Promises

A *promise* represents the completion of an asynchronous function. It is an object that might return a value in the future. It accomplishes the same basic goal as a callback function, but with many additional features and a more readable syntax. As a JavaScript developer, you will likely spend more time consuming promises than creating them, as it is usually asynchronous Web APIs that return a promise for the developer to consume. This tutorial will show you how to do both.

Creating a Promise

You can initialize a promise with the `new Promise` syntax, and you must initialize it with a function. The function that gets passed to a promise has `resolve` and `reject` parameters. The `resolve` and `reject` functions handle the success and failure of an operation, respectively.

Write the following line to declare a promise:

```
// Initialize a promise
const promise = new Promise((resolve, reject) => {})
```

If you inspect the initialized promise in this state with your web browser's console, you will find it has a `pending` status and `undefined` value:

```
Output
__proto__: Promise
[[PromiseStatus]]:"pending"
[[PromiseValue]]:undefined
```

So far, nothing has been set up for the promise, so it's going to sit there in a `pending` state forever. The first thing you can do to test out a promise is fulfill the promise by resolving it with a value:

```
const promise = new Promise((resolve, reject) => {
resolve('We did it!')})
```

Now, upon inspecting the promise, you'll find that it has a status of `fulfilled`, and a `value` set to the value you passed to `resolve`:

```
Output
__proto__: Promise
[[PromiseStatus]]: "fulfilled"
[[PromiseValue]]: "We did it!"
```

As stated in the beginning of this section, a promise is an object that may return a value. After being successfully fulfilled, the `value` goes from `undefined` to being populated with data.

A promise can have three possible states: pending, fulfilled, and rejected.

- **Pending** - Initial state before being resolved or rejected
- **Fulfilled** - Successful operation, promise has resolved
- **Rejected** - Failed operation, promise has rejected

After being fulfilled or rejected, a promise is settled.

Now that you have an idea of how promises are created, let's look at how a developer may consume these promises.

Consuming a Promise

The promise in the last section has fulfilled with a value, but you also want to be able to access the value. Promises have a method called `then` that will run after a promise reaches `resolve` in the code. `then` will return the promise's value as a parameter.

This is how you would return and log the `value` of the example promise:

```
promise.then((response) => {
  console.log(response)
})
```

Copy

The promise you created had a `[[PromiseValue]]` of `We did it!`. This value is what will be passed into the anonymous function as `response`:

```
Output
We did it!
```

So far, the example you created did not involve an asynchronous Web API—it only explained how to create, resolve, and consume a native JavaScript promise. Using `setTimeout`, you can test out an asynchronous request.

The following code simulates data returned from an asynchronous request as a promise:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Resolving an asynchronous request!'), 2000)
})

// Log the result
promise.then((response) => {
  console.log(response)
})
```

Using the `then` syntax ensures that the `response` will be logged only when the `setTimeout` operation is completed after `2000` milliseconds. All this is done without nesting callbacks.

Now after two seconds, it will resolve the promise value and it will get logged in `then`:

```
Output
Resolving an asynchronous request!
```

Promises can also be chained to pass along data to more than one asynchronous operation. If a value is returned in `then`, another `then` can be added that will fulfill with the return value of the previous `then`:

```
// Chain a promise
promise
  .then((firstResponse) => {
    // Return a new value for the next then
    return firstResponse + ' And chaining!'
  })
  .then((secondResponse) => {
    console.log(secondResponse)
  })
```

The fulfilled response in the second `then` will log the return value:

```
Output
Resolving an asynchronous request! And chaining!
```

Since `then` can be chained, it allows the consumption of promises to appear more synchronous than callbacks, as they do not need to be nested. This will allow for more readable code that can be maintained and verified easier.

Error Handling

So far, you have only handled a promise with a successful `resolve`, which puts the promise in a `fulfilled` state. But frequently with an asynchronous request you also have to handle an error—if the API is down, or a malformed or unauthorized request is sent. A promise should be able to handle both cases. In this section, you will create a function to test out both the success and error case of creating and consuming a promise.

This `getUsers` function will pass a flag to a promise, and return the promise:

```
function getUsers(onSuccess) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Handle resolve and reject in the asynchronous API
    }, 1000)
  })
}
```

Set up the code so that if `onSuccess` is `true`, the timeout will fulfill with some data. If `false`, the function will reject with an error:

```
function getUsers(onSuccess) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Handle resolve and reject in the asynchronous API
if (onSuccess) {resolve([{id: 1, name: 'Jerry'},{id: 2, name: 'Elaine'},{id: 3, name: 'George'},])} else {reject('Failed to fetch d
ata!')}}, 1000)
  })
}
```

For the successful result, you return JavaScript objects that represent sample user data.

In order to handle the error, you will use the `catch` instance method. This will give you a failure callback with the `error` as a parameter.

Run the `getUser` command with `onSuccess` set to `false`, using the `then` method for the success case and the `catch` method for the error:

```
// Run the getUsers function with the false flag to trigger an error
getUsers(false)
  .then((response) => {
    console.log(response)
  })
  .catch((error) => {
    console.error(error)
  })
```

Copy

Since the error was triggered, the `then` will be skipped and the `catch` will handle the error:

```
Output
Failed to fetch data!
```

If you switch the flag and `resolve` instead, the `catch` will be ignored, and the data will return instead:

```
// Run the getUsers function with the true flag to resolve successfully
getUsers(true)
  .then((response) => {
    console.log(response)
  })
  .catch((error) => {
    console.error(error)
  })
```

This will yield the user data:

```
Output
(3) [{…}, {…}, {…}]
0: {id: 1, name: "Jerry"}
1: {id: 2, name: "Elaine"}
3: {id: 3, name: "George"}
```

For reference, here is a table with the handler methods on `Promise` objects:

Method	Description
`then()`	Handles a `resolve`. Returns a promise, and calls `onFulfilled` function asynchronously
`catch()`	Handles a `reject`. Returns a promise, and calls `onRejected` function asynchronously
`finally()`	Called when a promise is settled. Returns a promise, and calls `onFinally` function asynchronously

Promises can be confusing, both for new developers and experienced programmers that have never worked in an asynchronous environment before. However as mentioned, it is much more common to consume promises than create them. Usually, a browser's Web API or third party library will be providing the promise, and you only need to consume it.

In the final promise section, this tutorial will cite a common use case of a Web API that returns promises: the Fetch API.

Using the Fetch API with Promises

One of the most useful and frequently used Web APIs that returns a promise is the Fetch API, which allows you to make an asynchronous resource request over a network. `fetch` is a two-part process, and therefore requires chaining `then`. This example demonstrates hitting the GitHub API to fetch a user's data, while also handling any potential error:

```
// Fetch a user from the GitHub API
fetch('https://api.github.com/users/octocat')
  .then((response) => {
    return response.json()
  })
  .then((data) => {
    console.log(data)
  })
  .catch((error) => {
    console.error(error)
  })
```

The `fetch` request is sent to the `https://api.github.com/users/octocat` URL, which asynchronously waits for a response. The first `then` passes the response to an anonymous function that formats the response as JSON data, then passes the JSON to a second `then` that logs the data to the console. The `catch` statement logs any error to the console.

Running this code will yield the following:

```
Output
login: "octocat",
id: 583231,
avatar_url: "https://avatars3.githubusercontent.com/u/583231?v=4"
blog: "https://github.blog"
company: "@github"
followers: 3203
...
```

This is the data requested from `https://api.github.com/users/octocat`, rendered in JSON format.

This section of the tutorial showed that promises incorporate a lot of improvements for dealing with asynchronous code. But, while using `then` to handle asynchronous actions is easier to follow than the pyramid of callbacks, some developers still prefer a synchronous format of writing asynchronous code. To address this need, ECMAScript 2016 (ES7) introduced `async` functions and the `await` keyword to make working with promises easier.

Async Functions with `async/await`

An *`async` function* allows you to handle asynchronous code in a manner that appears synchronous. `async` functions still use promises under the hood, but have a more traditional JavaScript syntax. In this section, you will try out examples of this syntax.

You can create an `async` function by adding the `async` keyword before a function:

```
// Create an async function
async function getUser() {
  return {}
}
```

Although this function is not handling anything asynchronous yet, it behaves differently than a traditional function. If you execute the function, you'll find that it returns a promise with a `[[PromiseStatus]]` and `[[PromiseValue]]` instead of a return value.

Try this out by logging a call to the `getUser` function:

```
console.log(getUser())
```

This will give the following:

```
Output
__proto__: Promise
[[PromiseStatus]]: "fulfilled"
[[PromiseValue]]: Object
```

This means you can handle an `async` function with `then` in the same way you could handle a promise. Try this out with the following code:

```
getUser().then((response) => console.log(response))
```

This call to `getUser` passes the return value to an anonymous function that logs the value to the console.

You will receive the following when you run this program:

```
Output
{}
```

An `async` function can handle a promise called within it using the `await` operator. `await` can be used within an `async` function and will wait until a promise settles before executing the designated code.

With this knowledge, you can rewrite the Fetch request from the last section using `async`/`await` as follows:

```
// Handle fetch with async/await
async function getUser() {
  const response =await fetch('https://api.github.com/users/octocat')
  const data =await response.json()

  console.log(data)
}

// Execute async function
getUser()
```

The `await` operators here ensure that the `data` is not logged before the request has populated it with data.

Now the final `data` can be handled inside the `getUser` function, without any need for using `then`. This is the output of logging `data`:

```
Output
login: "octocat",
id: 583231,
avatar_url: "https://avatars3.githubusercontent.com/u/583231?v=4"
blog: "https://github.blog"
company: "@github"
followers: 3203
...
```

Note: In many environments, `async` is necessary to use `await`—however, some new versions of browsers and Node allow using top-level `await`, which allows you to bypass creating an async function to wrap the `await` in.

Finally, since you are handling the fulfilled promise within the asynchronous function, you can also handle the error within the function. Instead of using the `catch` method with `then`, you will use the `try`/`catch` pattern to handle the exception.

Add the following highlighted code:

```
// Handling success and errors with async/await
async function getUser() {
try {// Handle success in try
    const response = await fetch('https://api.github.com/users/octocat')
    const data = await response.json()

    console.log(data)
} catch (error) {// Handle error in catch
console.error(error)}}
```

The program will now skip to the `catch` block if it receives an error and log that error to the console.

Modern asynchronous JavaScript code is most often handled with `async`/`await` syntax, but it is important to have a working knowledge of how promises work, especially as promises are capable of additional features that cannot be handled with `async`/`await`, like combining promises with `Promise.all()`.

Note: `async`/`await` can be reproduced by using generators combined with promises to add more flexibility to your code. To learn more, check out our Understanding Generators in JavaScript tutorial.

Conclusion

Because Web APIs often provide data asynchronously, learning how to handle the result of asynchronous actions is an essential part of being a JavaScript developer. In this article, you learned how the host environment uses the event loop to handle the order of execution of code with the *stack* and *queue*. You also tried out examples of three ways to handle the success or failure of an asynchronous event, with callbacks, promises, and `async`/`await` syntax. Finally, you used the Fetch Web API to handle asynchronous actions.

14. Closures

Closures

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

Lexical scoping

Consider the following example code:

```
function init() {
  var name = 'Mozilla'; // name is a local variable created by init
  function displayName() {
    // displayName() is the inner function, a closure
    console.log(name); // use variable declared in the parent function
  }
  displayName();
}
init();
```

`init()` creates a local variable called `name` and a function called `displayName()`. The `displayName()` function is an inner function that is defined inside `init()` and is available only within the body of the `init()` function. Note that the `displayName()` function has no local variables of its own. However, since inner functions have access to the variables of outer functions, `displayName()` can access the variable `name` declared in the parent function, `init()`.

Run the code using this JSFiddle link and notice that the `console.log()` statement within the `displayName()` function successfully displays the value of the `name` variable, which is declared in its parent function. This is an example of *lexical scoping*, which describes how a parser resolves variable names when functions are nested. The word *lexical* refers to the fact that lexical

scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

In this particular example, the scope is called a *function scope*, because the variable is accessible and only accessible within the function body where it's declared.

Scoping with let and const

Traditionally (before ES6), JavaScript only had two kinds of scopes: *function scope* and *global scope*. Variables declared with `var` are either function-scoped or global-scoped, depending on whether they are declared within a function or outside a function. This can be tricky, because blocks with curly braces do not create scopes:

```
if (Math.random() > 0.5) {
  var x = 1;
} else {
  var x = 2;
}
console.log(x);
```

For people from other languages (e.g. C, Java) where blocks create scopes, the above code should throw an error on the `console.log` line, because we are outside the scope of `x` in either block. However, because blocks don't create scopes for `var`, the `var` statements here actually create a global variable. There is also a practical example introduced below that illustrates how this can cause actual bugs when combined with closures.

In ES6, JavaScript introduced the `let` and `const` declarations, which, among other things like temporal dead zones, allow you to create block-scoped variables.

```
if (Math.random() > 0.5) {
  const x = 1;
} else {
  const x = 2;
}
console.log(x); // ReferenceError: x is not defined
```

In essence, blocks are finally treated as scopes in ES6, but only if you declare variables with `let` or `const`. In addition, ES6 introduced modules, which introduced another kind of scope. Closures are able to capture variables in all these scopes, which we will introduce later.

Closure

Consider the following code example:

```
function makeFunc() {
  const name = 'Mozilla';
  function displayName() {
    console.log(name);
  }
  return displayName;
}

const myFunc = makeFunc();
myFunc();
```

Running this code has exactly the same effect as the previous example of the `init()` function above. What's different (and interesting) is that the `displayName()` inner function is returned from the outer function *before being executed*.

At first glance, it might seem unintuitive that this code still works. In some programming languages, the local variables within a function exist for just the duration of that function's execution. Once `makeFunc()` finishes executing, you might expect that the `name` variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form closures. A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created. In this case, `myFunc` is a reference to the instance of the function `displayName` that is created when `makeFunc` is run. The instance of `displayName` maintains a reference to its lexical environment, within which the

variable `name` exists. For this reason, when `myFunc` is invoked, the variable `name` remains available for use, and "Mozilla" is passed to `console.log`.

Here's a slightly more interesting example—a `makeAdder` function:

```
function makeAdder(x) {
  return function (y) {
    return x + y;
  };
}

const add5 = makeAdder(5);
const add10 = makeAdder(10);

console.log(add5(2)); // 7
console.log(add10(2)); // 12
```

In this example, we have defined a function `makeAdder(x)`, that takes a single argument `x`, and returns a new function. The function it returns takes a single argument `y`, and returns the sum of `x` and `y`.

In essence, `makeAdder` is a function factory. It creates functions that can add a specific value to their argument. In the above example, the function factory creates two new functions—one that adds five to its argument, and one that adds 10.

`add5` and `add10` are both closures. They share the same function body definition, but store different lexical environments. In `add5`'s lexical environment, `x` is 5, while in the lexical environment for `add10`, `x` is 10.

Practical closures

Closures are useful because they let you associate data (the lexical environment) with a function that operates on that data. This has obvious parallels to object-oriented programming, where objects allow you to associate data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Situations where you might want to do this are particularly common on the web. Much of the code written in front-end JavaScript is event-based. You define some behavior, and then attach it to an event that is triggered by the user (such as a click or a keypress). The code is attached as a callback (a single function that is executed in response to the event).

For instance, suppose we want to add buttons to a page to adjust the text size. One way of doing this is to specify the font-size of the `body` element (in pixels), and then set the size of the other elements on the page (such as headers) using the relative `em` unit:

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 1.5em;
}

h2 {
  font-size: 1.2em;
}
```

Such interactive text size buttons can change the `font-size` property of the `body` element, and the adjustments are picked up by other elements on the page thanks to the relative units.

Here's the JavaScript:

```
function makeSizer(size) {
  return function () {
    document.body.style.fontSize = `${size}px`;
  };
}

const size12 = makeSizer(12);
const size14 = makeSizer(14);
const size16 = makeSizer(16);
```

`size12`, `size14`, and `size16` are now functions that resize the body text to 12, 14, and 16 pixels, respectively. You can attach them to buttons (in this case hyperlinks) as demonstrated in the following code example.

```
document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
```

```
<button id="size-12">12</button>
<button id="size-14">14</button>
<button id="size-16">16</button>
```

Emulating private methods with closures

Languages such as Java allow you to declare methods as private, meaning that they can be called only by other methods in the same class.

JavaScript, prior to classes, didn't have a native way of declaring private methods, but it was possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code. They also provide a powerful way of managing your global namespace.

The following code illustrates how to use closures to define public functions that can access private functions and variables. Note that these closures follow the Module Design Pattern.

```
const counter = (function () {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment() {
      changeBy(1);
    },

    decrement() {
      changeBy(-1);
    },

    value() {
      return privateCounter;
    },
  };
})();

console.log(counter.value()); // 0.

counter.increment();
counter.increment();
console.log(counter.value()); // 2.

counter.decrement();
console.log(counter.value()); // 1.
```

In previous examples, each closure had its own lexical environment. Here though, there is a single lexical environment that is shared by the three functions: `counter.increment`, `counter.decrement`, and `counter.value`.

The shared lexical environment is created in the body of an anonymous function, *which is executed as soon as it has been defined* (also known as an IIFE). The lexical environment contains two private items: a variable called `privateCounter`, and a function called `changeBy`. You can't access either of these private members from outside the anonymous function. Instead, you can access them using the three public functions that are returned from the anonymous wrapper.

Those three public functions are closures that share the same lexical environment. Thanks to JavaScript's lexical scoping, they each have access to the `privateCounter` variable and the `changeBy` function.

```
const makeCounter = function () {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
```

```
  return {
    increment() {
      changeBy(1);
    },

    decrement() {
      changeBy(-1);
    },

    value() {
      return privateCounter;
    },
  };
};

const counter1 = makeCounter();
const counter2 = makeCounter();

console.log(counter1.value()); // 0.

counter1.increment();
counter1.increment();
console.log(counter1.value()); // 2.

counter1.decrement();
console.log(counter1.value()); // 1.
console.log(counter2.value()); // 0.
```

Notice how the two counters maintain their independence from one another. Each closure references a different version of the `privateCounter` variable through its own closure. Each time one of the counters is called, its lexical environment changes by changing the value of this variable. Changes to the variable value in one closure don't affect the value in the other closure.

Note: Using closures in this way provides benefits that are normally associated with object-oriented programming. In particular, *data hiding* and *encapsulation*.

Closure scope chain

Every closure has three scopes:

- Local scope (Own scope)
- Enclosing scope (can be block, function, or module scope)
- Global scope

A common mistake is not realizing that in the case where the outer function is itself a nested function, access to the outer function's scope includes the enclosing scope of the outer function—effectively creating a chain of function scopes. To demonstrate, consider the following example code.

```
// global scope
const e = 10;
function sum(a) {
  return function (b) {
    return function (c) {
      // outer functions scope
      return function (d) {
        // local scope
        return a + b + c + d + e;
      };
    };
  };
}

console.log(sum(1)(2)(3)(4)); // 20
```

You can also write without anonymous functions:

```
// global scope
const e = 10;
function sum(a) {
  return function sum2(b) {
    return function sum3(c) {
      // outer functions scope
      return function sum4(d) {
        // local scope
        return a + b + c + d + e;
      };
```

```
    };
  };
}

const sum2 = sum(1);
const sum3 = sum2(2);
const sum4 = sum3(3);
const result = sum4(4);
console.log(result); // 20
```

In the example above, there's a series of nested functions, all of which have access to the outer functions' scope. In this context, we can say that closures have access to *all* outer function scopes.

Closures can capture variables in block scopes and module scopes as well. For example, the following creates a closure over the block-scoped variable `y`:

```
function outer() {
  const x = 5;
  if (Math.random() > 0.5) {
    const y = 6;
    return () => console.log(x, y);
  }
}

outer()(); // Logs 5 6
```

Closures over modules can be more interesting.

```
// myModule.js
let x = 5;
export const getX = () => x;
export const setX = (val) => {
  x = val;
}
```

Here, the module exports a pair of getter-setter functions, which close over the module-scoped variable `x`. Even when `x` is not directly accessible from other modules, it can be read and written with the functions.

```
import { getX, setX } from "./myModule.js";

console.log(getX()); // 5
setX(6);
console.log(getX()); // 6
```

Closures can close over imported values as well, which are regarded as *live bindings*, because when the original value changes, the imported one changes accordingly.

```
// myModule.js
export let x = 1;
export const setX = (val) => {
  x = val;
}
```

```
// closureCreator.js
import { x } from "./myModule.js";

export const getX = () => x; // Close over an imported live binding
```

```
import { getX } from "./closureCreator.js";
import { setX } from "./myModule.js";

console.log(getX()); // 1
setX(2);
console.log(getX()); // 2
```

Creating closures in loops: A common mistake

Prior to the introduction of the `let` keyword, a common problem with closures occurred when you created them inside a loop. To demonstrate, consider the following example code.

```
<p id="help">Helpful notes will appear here</p>
<p>Email: <input type="text" id="email" name="email" /></p>
<p>Name: <input type="text" id="name" name="name" /></p>
<p>Age: <input type="text" id="age" name="age" /></p>
```

```
function showHelp(help) {
  document.getElementById('help').textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: 'email', help: 'Your email address' },
    { id: 'name', help: 'Your full name' },
    { id: 'age', help: 'Your age (you must be over 16)' },
  ];

  for (var i = 0; i < helpText.length; i++) {
    // Culprit is the use of `var` on this line
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function () {
      showHelp(item.help);
    };
  }
}

setupHelp();
```

The `helpText` array defines three helpful hints, each associated with the ID of an input field in the document. The loop cycles through these definitions, hooking up an `onfocus` event to each one that shows the associated help method.

If you try this code out, you'll see that it doesn't work as expected. No matter what field you focus on, the message about your age will be displayed.

The reason for this is that the functions assigned to `onfocus` are closures; they consist of the function definition and the captured environment from the `setupHelp` function's scope. Three closures have been created by the loop, but each one shares the same single lexical environment, which has a variable with changing values (`item`). This is because the variable `item` is declared with `var` and thus has function scope due to hoisting. The value of `item.help` is determined when the `onfocus` callbacks are executed. Because the loop has already run its course by that time, the `item` variable object (shared by all three closures) has been left pointing to the last entry in the `helpText` list.

One solution in this case is to use more closures: in particular, to use a function factory as described earlier:

```
function showHelp(help) {
  document.getElementById('help').textContent = help;
}

function makeHelpCallback(help) {
  return function () {
    showHelp(help);
  };
}

function setupHelp() {
  var helpText = [
    { id: 'email', help: 'Your email address' },
    { id: 'name', help: 'Your full name' },
    { id: 'age', help: 'Your age (you must be over 16)' },
  ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
  }
}

setupHelp();
```

This works as expected. Rather than the callbacks all sharing a single lexical environment, the `makeHelpCallback` function creates *a new lexical environment* for each callback, in which `help` refers to the corresponding string from the `helpText` array.

One other way to write the above using anonymous closures is:

```
function showHelp(help) {
  document.getElementById('help').textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: 'email', help: 'Your email address' },
    { id: 'name', help: 'Your full name' },
    { id: 'age', help: 'Your age (you must be over 16)' },
  ];

  for (var i = 0; i < helpText.length; i++) {
    (function () {
      var item = helpText[i];
      document.getElementById(item.id).onfocus = function () {
        showHelp(item.help);
      };
    })(); // Immediate event listener attachment with the current value of item (preserved until iteration).
  }
}

setupHelp();
```

If you don't want to use more closures, you can use the `let` or `const` keyword:

```
function showHelp(help) {
  document.getElementById('help').textContent = help;
}

function setupHelp() {
  const helpText = [
    { id: 'email', help: 'Your email address' },
    { id: 'name', help: 'Your full name' },
    { id: 'age', help: 'Your age (you must be over 16)' },
  ];

  for (let i = 0; i < helpText.length; i++) {
    const item = helpText[i];
    document.getElementById(item.id).onfocus = () => {
      showHelp(item.help);
    };
  }
}

setupHelp();
```

This example uses `const` instead of `var`, so every closure binds the block-scoped variable, meaning that no additional closures are required.

Another alternative could be to use `forEach()` to iterate over the `helpText` array and attach a listener to each `<input>`, as shown:

```
function showHelp(help) {
  document.getElementById('help').textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: 'email', help: 'Your email address' },
    { id: 'name', help: 'Your full name' },
    { id: 'age', help: 'Your age (you must be over 16)' },
  ];

  helpText.forEach(function (text) {
    document.getElementById(text.id).onfocus = function () {
      showHelp(text.help);
    };
  });
}

setupHelp();
```

Performance considerations

As mentioned previously, each function instance manages its own scope and closure. Therefore, it is unwise to unnecessarily create functions within other functions if closures are not needed for a particular task, as it will negatively affect script

performance both in terms of processing speed and memory consumption.

For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. The reason is that whenever the constructor is called, the methods would get reassigned (that is, for every object creation).

Consider the following case:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function () {
    return this.name;
  };

  this.getMessage = function () {
    return this.message;
  };
}
```

Because the previous code does not take advantage of the benefits of using closures in this particular instance, we could instead rewrite it to avoid using closures as follows:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype = {
  getName() {
    return this.name;
  },
  getMessage() {
    return this.message;
  },
};
```

However, redefining the prototype is not recommended. The following example instead appends to the existing prototype:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function () {
  return this.name;
};
MyObject.prototype.getMessage = function () {
  return this.message;
};
```

15. IIFE - Immediately Invoked Function Expression

IIFE

An **IIFE** (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined. The name IIFE is promoted by Ben Alman in his blog.

```
(function () {
  // …
})();

(() => {
  // …
})();

(async () => {
  // …
})();
```

It is a design pattern which is also known as a Self-Executing Anonymous Function and contains two major parts:

1. The first is the anonymous function with lexical scope enclosed within the `Grouping Operator` `()`. This prevents accessing variables within the IIFE idiom as well as polluting the global scope.
2. The second part creates the immediately invoked function expression `()` through which the JavaScript engine will directly interpret the function.

Use cases

Avoid polluting the global namespace

Because our application could include many functions and global variables from different source files, it's important to limit the number of global variables. If we have some initiation code that we don't need to use again, we could use the IIFE pattern. As we will not reuse the code again, using IIFE in this case is better than using a function declaration or a function expression.

```
(() => {
  // some initiation code
  let firstVariable;
  let secondVariable;
})();

// firstVariable and secondVariable will be discarded after the function is executed.
```

Execute an async function

An `async` IIFE allows you to use `await` and `for-await` even in older browsers and JavaScript runtimes that have no top-level await:

```
const getFileStream = async (url) => {
  // implementation
};

(async () => {
  const stream = await getFileStream("https://domain.name/path/file.ext");
  for await (const chunk of stream) {
    console.log({ chunk });
  }
})();
```

The module pattern

We would also use IIFE to create private and public variables and methods. For a more sophisticated use of the module pattern and other use of IIFE, you could see the book Learning JavaScript Design Patterns by Addy Osmani.

```
const makeWithdraw = (balance) =>
  ((copyBalance) => {
    let balance = copyBalance; // This variable is private
    const doBadThings = () => {
      console.log("I will do bad things with your money");
    };
    doBadThings();
    return {
      withdraw(amount) {
        if (balance >= amount) {
          balance -= amount;
          return balance;
        }
        return "Insufficient money";
      },
    };
  })(balance);

const firstAccount = makeWithdraw(100); // "I will do bad things with your money"
console.log(firstAccount.balance); // undefined
console.log(firstAccount.withdraw(20)); // 80
console.log(firstAccount.withdraw(30)); // 50
console.log(firstAccount.doBadThings); // undefined; this method is private
const secondAccount = makeWithdraw(20); // "I will do bad things with your money"
console.log(secondAccount.withdraw(30)); // "Insufficient money"
console.log(secondAccount.withdraw(20)); // 0
```

For loop with var before ES6

We could see the following use of IIFE in some old code, before the introduction of the statements **let** and **const** in **ES6** and the block scope. With the statement **var**, we have only function scopes and the global scope. Suppose we want to create 2 buttons with the texts Button 0 and Button 1 and when we click them, we would like them to alert 0 and 1. The following code doesn't work:

```
for (var i = 0; i < 2; i++) {
  const button = document.createElement("button");
  button.innerText = `Button ${i}`;
  button.onclick = function () {
    console.log(i);
  };
  document.body.appendChild(button);
}
console.log(i); // 2
```

When clicked, both Button 0 and Button 1 alert 2 because `i` is global, with the last value 2. To fix this problem before ES6, we could use the IIFE pattern:

```
for (var i = 0; i < 2; i++) {
  const button = document.createElement("button");
  button.innerText = `Button ${i}`;
  button.onclick = (function (copyOfI) {
    return () => {
      console.log(copyOfI);
    };
  })(i);
  document.body.appendChild(button);
}
console.log(i); // 2
```

When clicked, Buttons 0 and 1 alert 0 and 1. The variable `i` is globally defined. Using the statement **let**, we could simply do:

```
for (let i = 0; i < 2; i++) {
  const button = document.createElement("button");
  button.innerText = `Button ${i}`;
  button.onclick = function () {
    console.log(i);
  };
  document.body.appendChild(button);
}
console.log(i); // Uncaught ReferenceError: i is not defined.
```

When clicked, these buttons alert 0 and 1.

16. JavaScript Collection - Map & Set

Understanding Map and Set in JavaScript

In JavaScript, developers often spend a lot of time deciding the correct data structure to use. This is because choosing the correct data structure can make it easier to manipulate that data later on, saving time and making code easier to comprehend. The two predominant data structures for storing collections of data are Objects and Arrays (a type of object). Developers use Objects to store key/value pairs and Arrays to store indexed lists. However, to give developers more flexibility, the ECMAScript 2015 specification introduced two new types of iterable objects: Maps, which are ordered collections of key/value pairs, and Sets, which are collections of unique values.

In this article, you will go over the Map and Set objects, what makes them similar or different to Objects and Arrays, the properties and methods available to them, and examples of some practical uses.

Maps

A Map is a collection of key/value pairs that can use any data type as a key and can maintain the order of its entries. Maps have elements of both Objects (a unique key/value pair collection) and Arrays (an ordered collection), but are more similar to Objects conceptually. This is because, although the size and order of entries is preserved like an Array, the entries themselves are key/value pairs like Objects.

Maps can be initialized with the `new Map()` syntax:

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

This gives us an empty Map:

```
Map(0) {}
```

Adding Values to a Map

You can add values to a map with the `set()` method. The first argument will be the key, and the second argument will be the value.

The following adds three key/value pairs to `map`:

```
map.set('firstName', 'Luke')
map.set('lastName', 'Skywalker')
map.set('occupation', 'Jedi Knight')
```

Here we begin to see how Maps have elements of both Objects and Arrays. Like an Array, we have a zero-indexed collection, and we can also see how many items are in the Map by default. Maps use the `=>` syntax to signify key/value pairs as `key => value`:

```
Map(3)
0: {"firstName" => "Luke"}
1: {"lastName" => "Skywalker"}
2: {"occupation" => "Jedi Knight"}
```

This example looks similar to a regular object with string-based keys, but we can use any data type as a key with Map.

In addition to manually setting values on a Map, we can also initialize a Map with values already. We do this using an Array of Arrays containing two elements that are each key/value pairs, which looks like this:

```
[ ['key1', 'value1'], ['key2', 'value2'] ]
```

Using the following syntax, we can recreate the same Map:

```
const map = new Map([
  ['firstName', 'Luke'],
  ['lastName', 'Skywalker'],
  ['occupation', 'Jedi Knight'],
])
```

Incidentally, this syntax is the same as the result of calling `Object.entries()` on an Object. This provides a ready-made way to convert an Object to a Map, as shown in the following code block:

```
const luke = {
  firstName: 'Luke',
  lastName: 'Skywalker',
  occupation: 'Jedi Knight',
}

const map = new Map(Object.entries(luke))
```

Alternatively, you can turn a Map back into an Object or an Array with a single line of code.

The following converts a Map to an Object:

```
const obj = Object.fromEntries(map)
```

This will result in the following value of `obj`:

```
{firstName: "Luke", lastName: "Skywalker", occupation: "Jedi Knight"}
```

Now, let's convert a Map to an Array:

```
const arr = Array.from(map)
```

This will result in the following Array for `arr`:

```
[
  ['firstName', 'Luke'],
  ['lastName', 'Skywalker'],
  ['occupation', 'Jedi Knight']
]
```

Map Keys

Maps accept any data type as a key, and do not allow duplicate key values. We can demonstrate this by creating a map and using non-string values as keys, as well as setting two values to the same key.

First, let's initialize a map with non-string keys:

```
const map = new Map()

map.set('1', 'String one')
map.set(1, 'This will be overwritten')
map.set(1, 'Number one')
map.set(true, 'A Boolean')
```

This example will override the first key of `1` with the subsequent one, and it will treat `'1'` the string and `1` the number as unique keys:

```
0: {"1" => "String one"}
1: {1 => "Number one"}
2: {true => "A Boolean"}
```

Although it is a common belief that a regular JavaScript Object can already handle Numbers, booleans, and other primitive data types as keys, this is actually not the case, because Objects change all keys to strings.

As an example, initialize an object with a numerical key and compare the value for a numerical `1` key and a stringified `"1"` key:

```
// Initialize an object with a numerical key
const obj = { 1: 'One' }

// The key is actually a string
obj[1] === obj['1'] // true
```

This is why if you attempt to use an Object as a key, it will print out the string `object Object` instead.

As an example, create an Object and then use it as the key of another Object:

```
// Create an object
const objAsKey = { foo: 'bar' }

// Use this object as the key of another object
const obj = {
  [objAsKey]: 'What will happen?',
}
```

This will yield the following:

```
{[object Object]: "What will happen?"}
```

This is not the case with Map. Try creating an Object and setting it as the key of a Map:

```
// Create an object
const objAsKey = { foo: 'bar' }

const map = new Map()

// Set this object as the key of a Map
map.set(objAsKey, 'What will happen?')
```

The `key` of the Map element is now the object we created.

```
key: {foo: "bar"}
value: "What will happen?"
```

There is one important thing to note about using an Object or Array as a key: the Map is using the reference to the Object to compare equality, not the literal value of the Object. In JavaScript `{} === {}` returns `false`, because the two Objects are not the same two Objects, despite having the same (empty) value.

That means that adding two unique Objects with the same value will create a Map with two entries:

```
// Add two unique but similar objects as keys to a Map
map.set({}, 'One')
map.set({}, 'Two')
```

This will yield the following:

```
Map(2) {{…} => "One", {…} => "Two"}
```

But using the same Object reference twice will create a Map with one entry.

```
// Add the same exact object twice as keys to a Map
const obj = {}

map.set(obj, 'One')
map.set(obj, 'Two')
```

Which will result in the following:

```
Map(1) {{…} => "Two"}
```

The second `set()` is updating the same exact key as the first, so we end up with a Map that only has one value.

Getting and Deleting Items from a Map

One of the disadvantages of working with Objects is that it can be difficult to enumerate them, or work with all the keys or values. The Map structure, by contrast, has a lot of built-in properties that make working with their elements more direct.

We can initialize a new Map to demonstrate the following methods and properties: `delete()`, `has()`, `get()`, and `size`.

```
// Initialize a new Map
const map = new Map([
  ['animal', 'otter'],
  ['shape', 'triangle'],
  ['city', 'New York'],
  ['country', 'Bulgaria'],
])
```

Use the `has()` method to check for the existence of an item in a map. `has()` will return a Boolean.

```
// Check if a key exists in a Map
map.has('shark') // false
map.has('country') // true
```

Use the `get()` method to retrieve a value by key.

```
// Get an item from a Map
map.get('animal') // "otter"
```

One particular benefit Maps have over Objects is that you can find the size of the Map at any time, like you can with an Array. You can get the count of items in a Map with the `size` property. This involves fewer steps than converting an Object to an Array to find the length.

```
// Get the count of items in a Map
map.size // 4
```

Use the `delete()` method to remove an item from a Map by key. The method will return a Boolean—`true` if an item existed and was deleted, and `false` if it did not match any item.

```
// Delete an item from a Map by key
map.delete('city') // true
```

This will result in the following Map:

```
Map(3) {"animal" => "otter", "shape" => "triangle", "country" => "Bulgaria"}
```

Finally, a Map can be cleared of all values with `map.clear()`.

```
// Empty a Map
map.clear()
```

This will yield:

```
Map(0) {}
```

Keys, Values, and Entries for Maps

Objects can retrieve keys, values, and entries by using the properties of the `Object` constructor. Maps, on the other hand, have prototype methods that allow us to get the keys, values, and entries of the Map instance directly.

The `keys()`, `values()`, and `entries()` methods all return a `MapIterator`, which is similar to an Array in that you can use `for...of` to loop through the values.

Here is another example of a Map, which we can use to demonstrate these methods.

```
const map = new Map([
  [1970, 'bell bottoms'],
  [1980, 'leg warmers'],
  [1990, 'flannel'],
])
```

The `keys()` method returns the keys:

```
map.keys()

MapIterator {1970, 1980, 1990}
```

The `values()` method returns the values:

```
map.values()

MapIterator {"bell bottoms", "leg warmers", "flannel"}
```

The `entries()` method returns an array of key/value pairs:

```
map.entries()

MapIterator {1970 => "bell bottoms", 1980 => "leg warmers", 1990 => "flannel"}
```

Iteration with Map

Map has a built-in `forEach` method, similar to an Array, for built-in iteration. However, there is a bit of a difference in what they iterate over. The callback of a Map's `forEach` iterates through the `value`, `key`, and `map` itself, while the Array version iterates through the `item`, `index`, and `array` itself.

```
// Map
Map.prototype.forEach((value, key, map) = () => {}

// Array
Array.prototype.forEach((item, index, array) = () => {}
```

This is a big advantage for Maps over Objects, as Objects need to be converted with `keys()`, `values()`, or `entries()`, and there is not an simple way to retrieve the properties of an Object without converting it.

To demonstrate this, let's iterate through our Map and log the key/value pairs to the console:

```
// Log the keys and values of the Map with forEach
map.forEach((value, key) => {
  console.log(`${key}: ${value}`)
})
```

This will give:

```
1970: bell bottoms
1980: leg warmers
1990: flannel
```

Since a `for...of` loop iterates over iterables like Map and Array, we can get the exact same result by destructuring the array of Map items:

```
// Destructure the key and value out of the Map item
for (const [key, value] of map) {
  // Log the keys and values of the Map with for...of
  console.log(`${key}: ${value}`)
}
```

Map Properties and Methods

The following table shows a list of Map properties and methods for quick reference:

Properties/Methods	Description	Returns
`set(key, value)`	Appends a key/value pair to a Map	`Map` Object
`delete(key)`	Removes a key/value pair from a Map by key	Boolean
`get(key)`	Returns a value by key	value
`has(key)`	Checks for the presence of an element in a Map by key	Boolean
`clear()`	Removes all items from a Map	N/A
`keys()`	Returns all keys in a Map	`MapIterator` object
`values()`	Returns all values in a Map	`MapIterator` object
`entries()`	Returns all keys and values in a Map as `[key, value]`	`MapIterator` object
`forEach()`	Iterates through the Map in insertion order	N/A
`size`	Returns the number of items in a Map	Number

When to Use Map

Summing up, Maps are similar to Objects in that they hold key/value pairs, but Maps have several advantages over objects:

- **Size** - Maps have a `size` property, whereas Objects do not have a built-in way to retrieve their size.
- **Iteration** - Maps are directly iterable, whereas Objects are not.
- **Flexibility** - Maps can have any data type (primitive or Object) as the key to a value, while Objects can only have strings.
- **Ordered** - Maps retain their insertion order, whereas objects do not have a guaranteed order.

Due to these factors, Maps are a powerful data structure to consider. However, Objects haves some important advantages as well:

- **JSON** - Objects work flawlessly with `JSON.parse()` and `JSON.stringify()`, two essential functions for working with JSON, a common data format that many REST APIs deal with.
- **Working with a single element** - Working with a known value in an Object, you can access it directly with the key without the need to use a method, such as Map's `get()`.

This list will help you decide if a Map or Object is the right data structure for your use case.

Set

A Set is a collection of unique values. Unlike a Map, a Set is conceptually more similar to an Array than an Object, since it is a list of values and not key/value pairs. However, Set is not a replacement for Arrays, but rather a supplement for providing additional support for working with duplicated data.

You can initialize Sets with the `new Set()` syntax.

```
const set = new Set()
```

This gives us an empty Set:

```
Set(0) {}
```

Items can be added to a Set with the `add()` method. (This is not to be confused with the `set()` method available to Map, although they are similar.)

```
// Add items to a Set
set.add('Beethoven')
set.add('Mozart')
set.add('Chopin')
```

Since Sets can only contain unique values, any attempt to add a value that already exists will be ignored.

```
set.add('Chopin') // Set will still contain 3 unique values
```

> Note: The same equality comparison that applies to Map keys applies to Set items. Two objects that have the same value but do not share the same reference will not be considered equal.

You can also initialize Sets with an Array of values. If there are duplicate values in the array, they will be removed from the Set.

```
// Initialize a Set from an Array
const set = new Set(['Beethoven', 'Mozart', 'Chopin', 'Chopin'])

Set(3) {"Beethoven", "Mozart", "Chopin"}
```

Conversely, a Set can be converted into an Array with one line of code:

```
const arr = [...set]

(3) ["Beethoven", "Mozart", "Chopin"]
```

Set has many of the same methods and properties as Map, including `delete()`, `has()`, `clear()`, and `size`.

```
// Delete an item
set.delete('Beethoven') // true

// Check for the existence of an item
set.has('Beethoven') // false

// Clear a Set
set.clear()

// Check the size of a Set
set.size // 0
```

Note that Set does not have a way to access a value by a key or index, like `Map.get(key)` or `arr[index]`.

Keys, Values, and Entries for Sets

Map and Set both have `keys()`, `values()`, and `entries()` methods that return an Iterator. However, while each one of these methods have a distinct purpose in Map, Sets do not have keys, and therefore keys are an alias for values. This means that `keys()` and `values()` will both return the same Iterator, and `entries()` will return the value twice. It makes the most sense to only use `values()` with Set, as the other two methods exist for consistency and cross-compatibility with Map.

```
const set = new Set([1, 2, 3])
// Get the values of a set
set.values()

SetIterator {1, 2, 3}
```

Iteration with Set

Like Map, Set has a built-in `forEach()` method. Since Sets don't have keys, the first and second parameter of the `forEach()` callback return the same value, so there is no use case for it outside of compatibility with Map. The parameters of `forEach()` are `(value, key, set)`.

Both `forEach()` and `for...of` can be used on Set. First, let's look at `forEach()` iteration:

```
const set = new Set(['hi', 'hello', 'good day'])

// Iterate a Set with forEach
set.forEach((value) => console.log(value))
```

Then we can write the `for...of` version:

```
// Iterate a Set with for...of
for (const value of set) {
  console.log(value)
}
```

Both of these strategies will yield the following:

```
hi
hello
good day
```

Set Properties and Methods

The following table shows a list of Set properties and methods for quick reference:

Properties/Methods	Description	Returns
`add(value)`	Appends a new item to a Set	`Set` Object
`delete(value)`	Removes the specified item from a Set	Boolean
`has()`	Checks for the presence of an item in a Set	Boolean
`clear()`	Removes all items from a Set	N/A

`keys()`	Returns all values in a Set (same as `values()`)	`SetIterator` object
`values()`	Returns all values in a Set (same as `keys()`)	`SetIterator` object
`entries()`	Returns all values in a Set as `[value, value]`	`SetIterator` object
`forEach()`	Iterates through the Set in insertion order	N/A
`size`	Returns the number of items in a Set	Number

When to Use Set

Set is a useful addition to your JavaScript toolkit, particularly for working with duplicate values in data.

In a single line, we can create a new Array without duplicate values from an Array that has duplicate values.

```
const uniqueArray = [...new Set([1, 1, 2, 2, 2, 3])]
// (3) [1, 2, 3]
```

This will give:

```
(3) [1, 2, 3]
```

Set can be used for finding the union, intersection, and difference between two sets of data. However, Arrays have a significant advantage over Sets for additional manipulation of the data due to the `sort()`, `map()`, `filter()`, and `reduce()` methods, as well as direct compatibility with `JSON` methods.

Conclusion

In this article, you learned that a Map is a collection of ordered key/value pairs, and that a Set is a collection of unique values. Both of these data structures add additional capabilities to JavaScript and simplify common tasks such as finding the length of a key/value pair collection and removing duplicate items from a data set, respectively. On the other hand, Objects and Arrays have been traditionally used for data storage and manipulation in JavaScript, and have direct compatibility with JSON, which continues to make them the most essential data structures, especially for working with REST APIs. Maps and Sets are primarily useful as supporting data structures for Objects and Arrays.

17. JavaScript Event Loop

JavaScript Event Loop

Summary: in this tutorial, you'll learn about the event loop in JavaScript and how JavaScript achieves the concurrency model based on the event loop.

JavaScript single-threaded model

JavaScript is a single-threaded programming language. This means that JavaScript can do only one thing at a single point in time.

The JavaScript engine executes a script from the top of the file and works its way down. It creates the execution contexts, pushes, and pops functions onto and off the call stack in the execution phase.

If a function takes a long time to execute, you cannot interact with the web browser during the function's execution because the page hangs.

A function that takes a long time to complete is called a blocking function. Technically, a blocking function blocks all the interactions on the webpage, such as mouse click.

An example of a blocking function is a function that calls an API from a remote server.

The following example uses a big loop to simulate a blocking function:

```
function task(message) {
// emulate time consuming tasklet n = 10000000000;
    while (n > 0){
        n--;
```

```
    }
    console.log(message);
}

console.log('Start script...');
task('Call an API');
console.log('Done!');
Code language: JavaScript (javascript)
```

In this example, we have a big `while` loop inside the `task()` function that emulates a time-consuming task. The `task()` function is a blocking function.

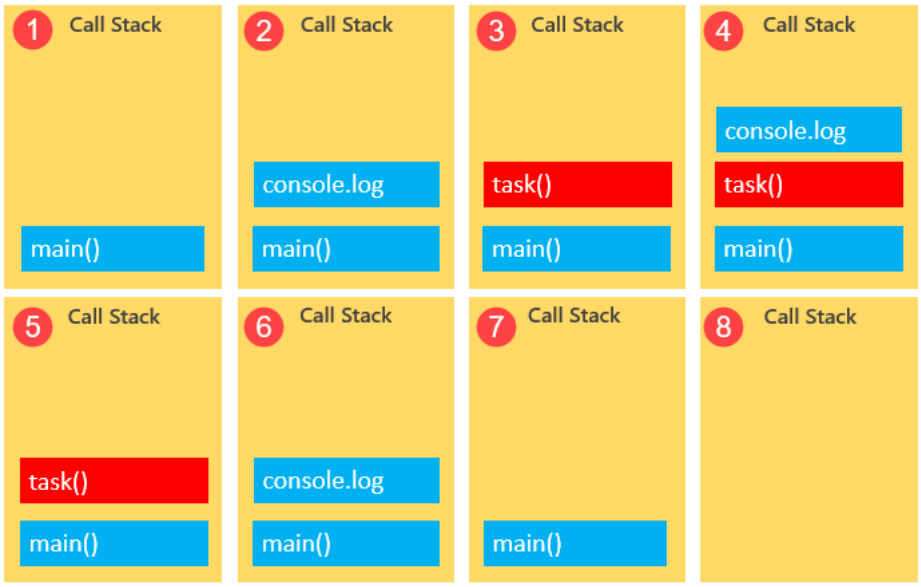
The script hangs for a few seconds (depending on how fast the computer is) and issues the following output:

```
Start script...
Download a file.
Done!
```

To execute the script, the JavaScript engine places the first call `console.log()` on top of the call stack and executes it. Then, it places the `task()` function on top of the call stack and executes the function.

However, it'll take a while to complete the `task()` function. Therefore, you'll see the message `'Download a file.'` a little time later. After the `task()` function completes, the JavaScript engine pops it off the call stack.

Finally, the JavaScript engine places the last call to the `console.log('Done!')` function and executes it, which will be very fast.



Callbacks to the rescue

To prevent a blocking function from blocking other activities, you typically put it in a callback function for execution later. For example:

```
console.log('Start script...');

setTimeout(() => {
    task('Download a file.');
}, 1000);

console.log('Done!');
Code language: JavaScript (javascript)
```

In this example, you'll see the message `'Start script...'` and `'Done!'` immediately. And after that, you'll see the message `'Download a file'`.

Here's the output:

```
Start script...
Done!
Download a file.
```

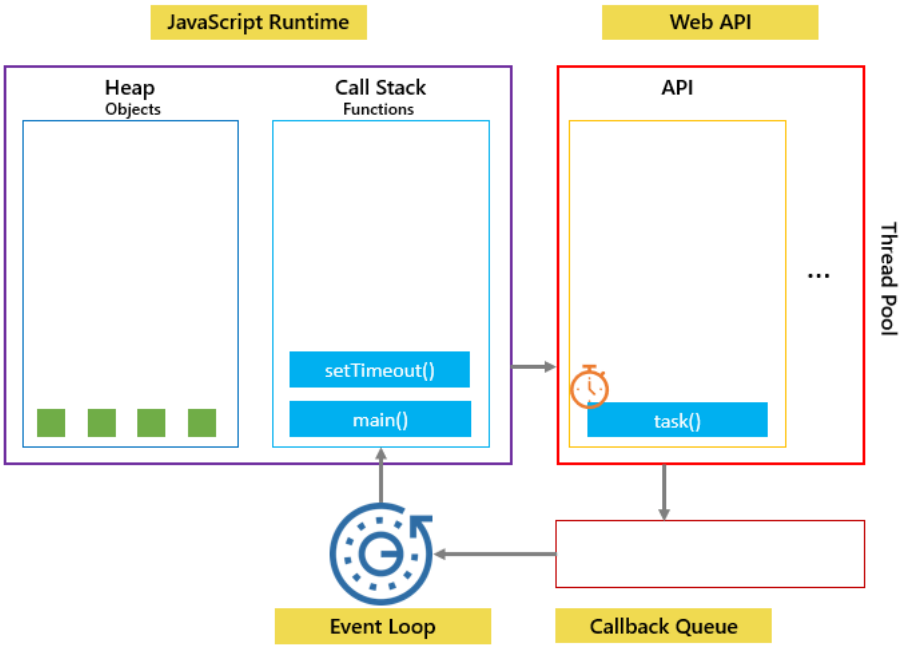
As mentioned earlier, the JavaScript engine can do only one thing at a time. However, it's more precise to say that the JavaScript runtime can do one thing at a time.

The web browser also has other components, not just the JavaScript engine.

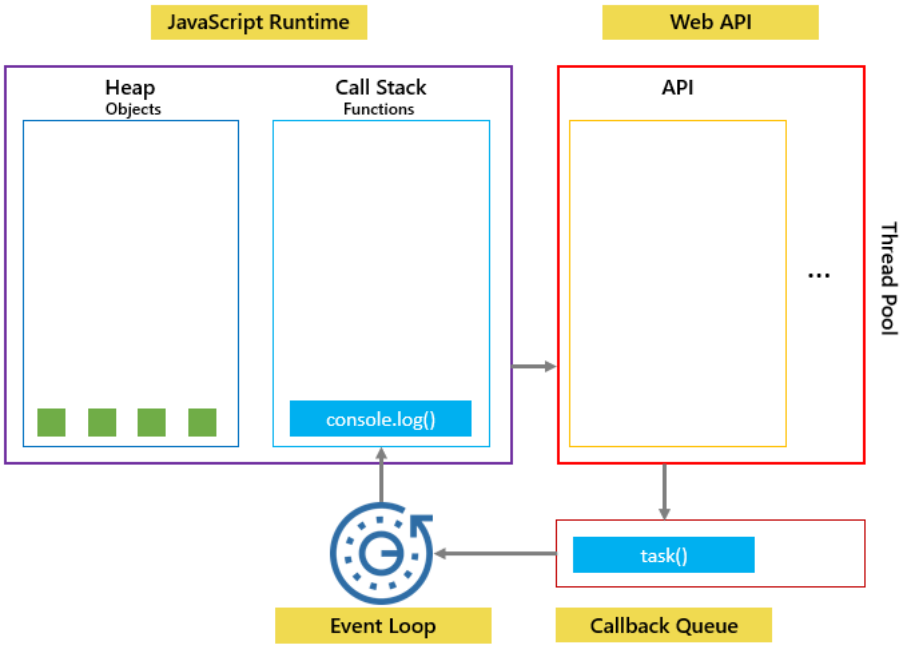
When you call the `setTimeout()` function, make a fetch request, or click a button, the web browser can do these activities concurrently and asynchronously.

The `setTimeout()`, fetch requests, and DOM events are parts of the Web APIs of the web browser.

In our example, when calling the `setTimeout()` function, the JavaScript engine places it on the call stack, and the Web API creates a timer that expires in 1 second.

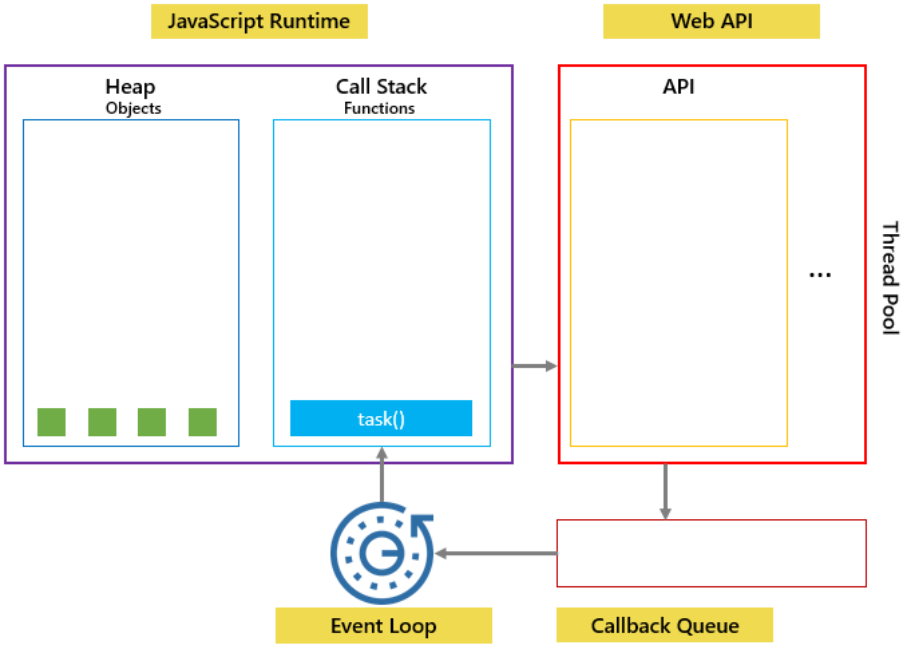


Then JavaScript engine place the `task()` function is into a queue called a callback queue or a task queue:



The event loop is a constantly running process that monitors both the callback queue and the call stack.

If the call stack is not empty, the event loop waits until it is empty and places the next function from the callback queue to the call stack. If the callback queue is empty, nothing will happen:



See another example:

```
console.log('Hi!');

setTimeout(() => {
    console.log('Execute immediately.');
}, 0);

console.log('Bye!');

Code language: JavaScript (javascript)
```

In this example, the timeout is 0 second, so the message `'Execute immediately.'` should appear before the message `'Bye!'`. However, it doesn't work like that.

The JavaScript engine places the following function call on the callback queue and executes it when the call stack is empty. In other words, the JavaScript engine executes it after the `console.log('Bye!')`.

```
console.log('Execute immediately.');
Code language: JavaScript (javascript)
```

Here's the output:

```
Hi!
Bye!
Execute immediately.
```

The following picture illustrates JavaScript runtime, Web API, Call stack, and Event loop: