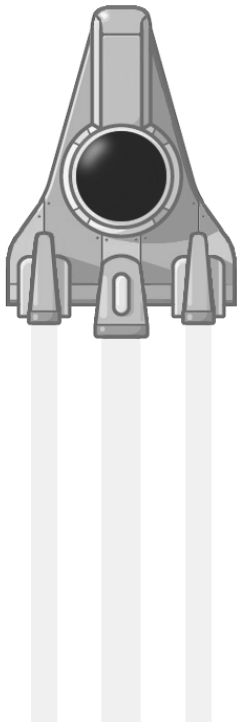
JAVASCRIPT GRAMMAR

JAVASCRIPT GRAMMAR



JavaScript Grammar – Edition I – March 23, 2019

Title: JavaScript Grammar
Edition: I – March 23, 2019
Genre: Software Development
Publisher: Learning Curve Books
Imprint: Independently published
ISBN: 9781091212169
Author: Greg Sidelnikov (greg.sidelnikov@gmail.com)

Editors, volunteers, contributors: Grace Neufeld.

Primary purpose of **Learning Curve Books** publishing company is to provide *effective education* for web designers, software engineers and all readers who are interested in being edified in the area of web development.

This edition of **JavaScript Grammar** was created to speed up the learning process of JavaScript – the language for programming websites, applications and robots!

For questions and comments about the book you may contact the author or send an email directly to our office at the email address mentioned below.

Special Offers & Discounts Available
Schools, libraries and educational organizations may qualify for special prices. Get in touch with our distribution department at **hello@learningcurvebook.net**

Learning Curve Books™

JavaScript Grammar

0.1 Foreword

We often think of the word "feature" as something that belongs to software products and services. Modern apps such as Instagram and Twitter have a "Follow" feature, for example. Uploading a photo to your account is another feature!

But computer languages have features too. A **function** is a feature. A **for loop** is a feature. So is the **class** keyword – all are computer language features.

In JavaScript some of these features are borrowed from other languages, while many remain unique to its own design. Features such as **this**, **class** and **const** may appear similar to their original C++ implementation, but in many cases they are used in a completely unique way to JavaScript.

JavaScript is an evolving language. When EcmaScript 6 came out in June 2015 the language experienced a Cambrian explosion of new features that radically changed how JavaScript code should be written.

New features like **...rest** and **...spread** syntax, **arrow functions**, **template strings**, object **destructuring** are commonplace in modern JavaScript code. But just a few years ago, even seasoned JavaScript developers with over a decade experience with the language couldn't conceive of such concepts.

Functional Programming started to creep into JavaScript community seemingly at the speed of light and higher-order functions (**.map**, **.filter**, **.reduce**) tied to Array methods, that remained dormant for many years, have gained increased popularity.

But JavaScript is a multi-paradigm language. Programmers who come from traditional Object Oriented Programming background will find themselves at home after induction of the **class** keyword and a separate constructor function that provides an alternative to the classic JavaScript object-function constructors.

The ES6 specification triggered a whole new breed of coders who have developed more respect for a language that once was used to write primitive DOM scripts.

JavaScript engines that run in browsers (Chrome browser's V8, for example) have matured and JavaScript is no longer looked at as a simple scripting language.

It's a whole new era of JavaScript development. Today, you may often stumble upon a video titled **Build a robot with JavaScript** on YouTube. It is even possible to build desktop applications for Windows 10 almost entirely in JavaScript.

JavaScript frameworks and libraries like **React** and **Vue** abstract away some of the classic JavaScript principles, making it quicker to build modular applications.

But this often comes at the expense of never having to understand vanilla JavaScript at the beginner level – its common grammar.

JavaScript Grammar was written to solve this problem by using carefully chosen subjects that, hopefully, match a natural learning experience. Content of this book will try to remain faithful to dynamic nature of JavaScript specification.

Finally, it is hoped that this book will encourage the reader to take the next step in the direction of more advanced subjects in the future.

Chapter 1

Presentation Format

This book was structured with continuity in mind: it is meant to be read from first to last page in a consecutive order. However, it can also be used as a desk reference for looking up isolated examples when you need them.

JavaScript Grammar is not a complete JavaScript reference or manual. But, this is probably a good thing. The subjects were reduced to only what's important in modern-day JavaScript environment.

Namely: imports, classes, constructors, key principles behind functional programming, including many features ranging from ES5 - ES10 are covered in this book.

The distinction between "ES" specifications has become less relevant. All of it is JavaScript. But just to give the reader a bit of perspective...

ES10 Sometimes you will see labels like this one.

This simply means that this feature was added to JavaScript as part of the EcmaScript's ES10 specification.

1.1 Creative Communication

Some of JavaScript is easy, some of it is difficult. Not everything can be explained by source code alone. Some things are based on intangible ideas or principles.

Throughout this tutorial book you will come across many creative communication devices, designed to make the learning process a bit easier and perhaps more fun.

One example of that is color-coded diagrams.

1.1.1 Theory

Not all subjects require extensive theory. On the other hand, some things won't make any sense without it. Additional discussion will be included, where it becomes absolutely necessary, in order to fully understand a particular concept.

1.1.2 Practical Examples

A practical example follows the theoretical discussion, so we can actually see the implementation. It will usually be explained by a source code listing.

1.1.3 Source Code

Source code listings will be provided to cement the foundational principles from preceding text.

```
054 // Create (instantiate) a sparrow from class Bird
055 let sparrow = new Bird("sparrow", "gray");
056 sparrow.fly();
057 sparrow.walk();
058 sparrow.lay_egg();
059 sparrow.talk(); // Error, only Parrot can talk
```

This is an example of instantiating `sparrow` object from `Bird` class and using some of its methods.

1.1.4 Color-Coded Diagrams

A significant amount of effort went into creating diagrams describing fundamental ideas behind JavaScript. They were designed for communicative value, hopefully

they will speed up the learning process in places where hard to grasp abstract ideas need to be explained visually.

There are two types of diagrams in this book: **abstract ideas** and **source code close ins**.

Abstract ideas

Sometimes there isn't a way to explain an abstract idea or its structure without a diagram. In places where that's the case, a diagram will be shown.

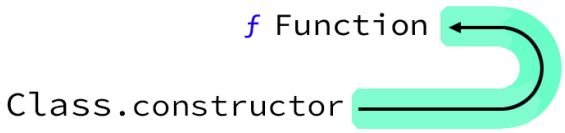


Figure 1.1: Class constructor is an object-function of type Function.

Here is another diagram visualizing anatomy of a JavaScript function:

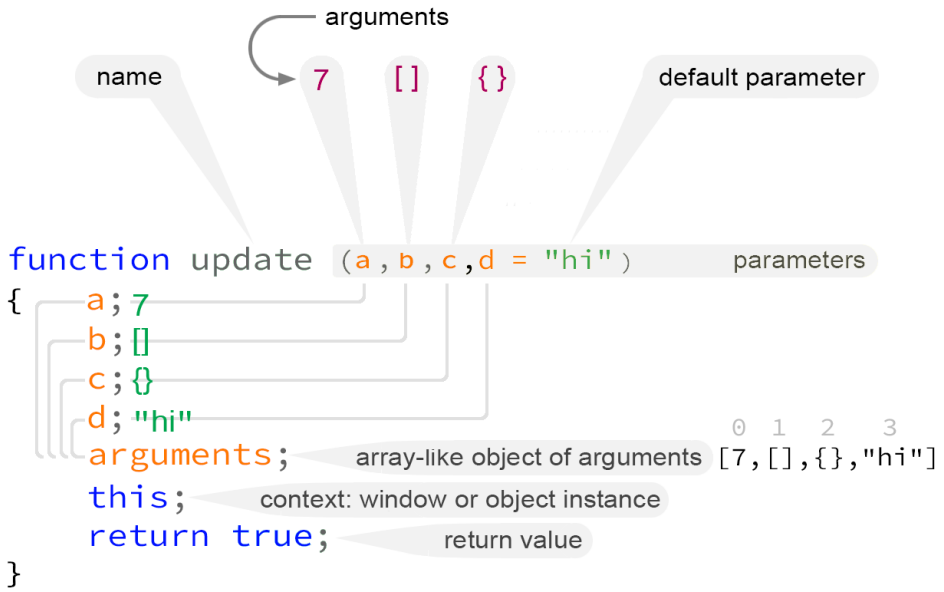


Figure 1.2: JavaScript function anatomy.

Code close ins

Most of the source code is accompanied by source code listings.

But when we need to close in on a particular important subject, a slightly larger diagram with source code and additional color-coded highlighting will be shown. For example, here is exploration of an anonymous function when used in the context of a event callback function:

```
setTimeout(function() {
  console.log("Print something in 1 second.");
  console.log(arguments);
}, 1000);
```

Figure 1.3: Anonymous function used as a **setTimeout** event callback.

In this case the source code will be missing line numbers because it's not important.

Content

We won't spend much book space or your time on countless listings of functions and available methods on every single object. This type of information can be easily looked up and practiced online on demand from Mozilla's MDN web docs, W3Schools and StackOverflow.

Much of content of this book is tailored to modern JavaScript development, which leans toward >= EcmaScript6 specification, functional programming: the use of higher-order Array functions, arrow functions and understanding execution context.

1.1.5 Dos and Dont's

An occasional Dos and Donts section will appear with insightful tips.

Chapter 2

Chrome Console

2.0.1 Beyond Console Log

Many programmers only know Chrome's `console.log` but the console API contains few other methods that have practical use, especially when time is of essence.

copy(obj) function

Copying JSON representation of an existing object to your copy buffer:

```
> let x = { property: 1, prop1: 2, method: function(){} };
> copy(x);
>
```

Now the JSON is in your copy-paste buffer, you can paste it into any text editor.

In this example `x` is a simple self-created object. But imagine a situation where a much more complex object is returned from a database API.

Note: Only JSON is returned, this means that methods will not make it to the copy buffer. (JSON string format does not support methods, only properties.)

2.0.2 console.dir

If you want to take a look at all object's properties and methods, you can print it out directly into the console using **console.dir** method:

```
> console.dir(x);
  ▼ Object
    ▶ method: ƒ ()
      prop1: 2
      property: 1
    ▶ __proto__: Object
>
```

What's fantastic is that you can even output DOM elements:

```
> console.dir(document.body);

  ▼ body
      aLink: ""
      accessKey: ""
      assignedSlot: null
    ▶ attributeStyleMap: StylePropertyMap {size: 0}
    ▶ attributes: NamedNodeMap {length: 0}
      autocapitalize: ""
      background: ""
      baseURI: "http://localhost/experiments/javascript.html"
      bgColor: ""
```

2.0.3 console.error

```
001 let fuel = 99;
002 function launch_rocket() {
003   function warning_msg() {
004     console.error("Not enough fuel.");
005   }
006   if (fuel >= 100) {
007     // looks like everything's ok
008   } else
009     warning_msg();
010 }
011
012 launch_rocket();
```

The great thing about `console.error` is that it also provides the stack trace:

2.0.4 console.time() and console.timeEnd()

You can track the amount of time between function calls. This can be helpful when optimizing code:

```
001 console.time();
002 let arr = new Array(10000);
003 for (let i = 0; i < arr.length; i++) {
004   arr[i] = new Object();
005 }
006 console.timeEnd();
```

Console output:

```
default: 2.51708984375ms
> 
```

2.0.5 console.clear

```
Console was cleared
<· undefined
> 
```

Printing Objects

In JavaScript all objects have `.toString()` method. When providing an object to `console.log(`*value*`)` it can print it either as an object, or as a string.

```
001 let obj = {};
002 console.log(obj);                // obj{}
003 console.log("object = " + obj); // [object Object]
004 console.log(`${obj}`);          // [object Object]
```

Chapter 3

Welcome To JavaScript

3.1 Entry Point

Every computer program has an **entry point**.

You can start writing your code directly into `<script>` tags. But this means it will be executed instantly and simultaneously as the script is being downloaded into the browser, without concern for DOM or other media.

This can create a problem because your code might be accessing DOM elements before they are fully downloaded from the server.

To remedy the situation, you may want to wait until the DOM tree is fully available.

DOMContentLoaded

To wait on the DOM event, add an event listener to the document object. The name of the event is `DOMContentLoaded`.

```
001 <html>
002   <head>
003     <title>DOM Loaded.</title>
004     <script type = "text/javascript">
005       function load() {
006         console.log("DOM Loaded.");
007       }
008       document.addEventListener("DOMContentLoaded", load);
009     </script>
010   </head>
011   <body></body>
012 </html>
```

Figure 3.1: Here the entry point is your own custom function `load()`. This is a good place for initializing your application objects.

You can rename the load function to **start**, **ready** or **initialize** – it doesn't matter.

What matters is that at this entry point we're 100% guaranteed that all DOM elements have been successfully loaded into memory and trying to access them with JavaScript will not produce an error.

3.1.1 Dos and Dont's

Do not write your code just in `<script>` tags, without entry point function.

Do use the entry point to initialize the default state of your data and objects.

Do make your program entry point either **DOMContentLoaded**, **readyState** or the native **window.onload** method for waiting on media (see next,) depending on whether you need to wait for just the DOM or the rest of media.

.readyState

For added safety you might also check the value of `readyState` property before attaching the `DOMContentLoaded` event:

```
<html>
  <head>
    <title>DOM Really Loaded.</title>
    <script type = "text/javascript">
      function load() {
        console.log("DOM Loaded.");
      }

      if (document.readyState == "loading") {
        document.addEventListener("DOMContentLoaded", load);
      } else {
        load();
      }
    </script>
  </head>
  <body></body>
</html>
```

Figure 3.2: Check `document.readyState`

DOM vs Media

We've just created a safe place for initializing our application. But because DOM is simply a tree-like structure of all HTML elements on the page, it usually becomes available *before* the rest of the media such as images and various embeds.

Even though `<image src = "http://url" />` is a DOM element, the URL content specified in image's `src` attribute might take more time to load.

To check if any non-DOM *media content* has finished downloading we can overload the native **window.onload** event as shown in the following example.

window.onload

With `window.onload` method, you can wait until all images and similar media have been fully downloaded:

```
<html>
  <head>
    <title>Window Media Loaded.</title>
    <script type = "text/javascript">
      window.onload = function() {
        /* DOM and media (images, embeds) */
      }
    </script>
  </head>
  <body></body>
</html>
```

Including External Scripts

Let's say we have the following definitions in `my-script.js` file:

```
let variable = 1;
function myfunction() { return 2; }
```

Then you can add them into your main application file as follows:

```
<html>
  <head>
    <title>Include External Script</title>
    <script src = "my-script.js"></script>
    <script type = "text/javascript">
      let result = myfunction();
      console.log(variable);     // 1
      console.log(result); // 2
    </script>
  </head>
  <body></body>
</html>
```

Main JavaScript application file – `index.html`, for example

Import

Starting from ES6 we should use **import** (and **export**) keyword to import variables, functions and classes from an external file.

Let's say we have a file `mouse.js` and it has following definition of a **Mouse** class.

```
001 export function Mouse() { this.x = 0; this.y = 0; }
```

In order to make a variable, object or a function available for export, the **export** keyword must be prepended to its definition.

But that's not enough! The **Mouse** constructor function will be exported as long as a matching **import** is available in main application file.

Not everything in a module will be exported. Some of the items will (and should) remain private to it. Be sure to prepend **export** keyword to anything you want to export from the file. This can be any named definition.

script type = "module"

In order to export the Mouse class and start using it in the application, we must make sure the script tag's **type** attribute is changed to "module" (*this is required.*)

```
001 <html>
002   <head>
003     <title>Import Module</title>
004     <script type = "module">
005       import { Mouse } from "./mouse.js";
006       let mouse = new Mouse();
007     </script>
008   </head>
009   <body></body>
010 </html>
```

Figure 3.3: Now we can safely access Mouse class, instantiate a new object from it and access its properties and methods.

Importing And Exporting Multiple Definitions

It's uncommon for a complex program to import only one class, function or variable.

Here is an example of how to import multiple items from two imaginary files.

```
001 import { Mouse, Keyboard } from "./input.js";
002 import { add, subtract, divide, multiply } from "./math.js";
003
004 // Initialize mouse object and access mouse position
005 let mouse = new Mouse();
006 mouse.x;          // 256
007 mouse.y;          // 128
008
009 // Initialize Keyboard class and check if shift is pressed
010 let keyboard = new Keyboard();
011 keyboard.shiftIsPressed; // false
012
013 // Use math functions from math library file
014 add(2, 5);        // 7
015 subtract(10, 5); // 5
016 divide(10, 5);   // 2
017 multiply(4, 2);  // 8
```

The **Mouse** and **Keyboard** classes were imported together (separated by comma) from **input.js** file. They were instantiated as separate objects then and their properties were accessed to grab some data.

We've also imported some math functions **add**, **subtract**, **divide** and **multiply** from **math.js**. The math library file source code is shown below. After defining 4 functions we can **export** multiple definitions as follows:

```
001 // A collection of math functions
002 function add(a,b) { return a + b; }
003 function subtract(a,b) { return a - b; }
004 function divide(a,b) { return a / b; }
005 function multiply(a,b) { return a * b; }
006
007 // Export multiple items
008 export { add, subtract, divide, multiply }
```

Figure 3.4: Exporting multiple definitions from `math.js`

3.1.2 ES10 Dynamic Import

Imports can be assigned to a variable since EcmaScript 10 (may not be available in your browser yet, at the time of this writing.)

```
001 element.addEventListener('click', async () => {
002   const module = await import('./api-scripts/click.js');
003   module.clickEvent();
004 });
```

3.2 Strict Mode

The **strict mode** is a feature available since ECMAScript 5 that allows you to place your entire program, or an isolated scope, in a "strict" operating context. This strict context prevents certain actions from being taken and throws an exception.

For example, in **strict mode** you cannot *use* undeclared variables. Without strict mode, using an undeclared variable will automatically create that variable.

Without strict mode, certain statements might not generate an error at all – even if they are not allowed – but you wouldn't know something was wrong.

```
001 var variable = 1;
002
003 delete variable; // false
```

Figure 3.5: You cannot use `delete` keyword to delete variables in JavaScript

Without strict mode on, code above will **fail silently**, variable will *not* be deleted, and `delete variable` will return **false** but your program will continue to run.

But what will happen in strict mode?

```
001 "use strict";
002
003 var variable = 1;
004
005 delete variable; // SyntaxError
```

Figure 3.6: In this example strict mode is enabled for entire global scope.

```
Uncaught SyntaxError: Delete of an
unqualified identifier in strict mode.
>
```

Figure 3.7: In strict mode you will generally become aware of more errors. For example, the line `delete variable` can be removed completely without having any impact on the program.

Limiting "strict mode" To A Scope

The strict mode doesn't have to be enabled globally. It is possible to isolate a single block (or function) scope to strict mode:

Final Words

In a professional environment, it is common to have strict mode on, because it can potentially prevent many bugs from happening and generally supports better software practice.

3.3 Literal Values

The literal representation of a number can be the digit `1`, `25`, `100` and so on. A string literal can be `"some text";`

You can combine literals using operators (+,−,/,∗, etc.) to produce a single result.

For example, to perform a $5 + 2$ operation, you will simply use the literal number values 5 and 2:

```
001 // Add two numberic literals to produce 7:
002 5 + 2;                          7
```

Combine two strings to produce a single sentence:

```
001 // Add two strings to produce a sentence:
002 "Hello" + " there.";            "Hello there."
```

Add two literal values of different types to produce a coerced value:

```
001 // Adding string to number to produce coerced value
002 "username” + 2574731;           "username2574731"
```

There is a literal value for just about everything in JavaScript:

```
1;                          // numeric literal
"some text.";               // string literal
[];                         // array literal
{};                         // object literal
true;                       // boolean literal
function() {};              // function is a value
```

There is an array literal `[]` and object literal $\{\}$.

You can add $\{\}$ + `[]` without breaking the program, but the results will not be meaningful. These types of cases are usually non-existent.

Note that a JavaScript function can be used as a value. You can even pass them into other functions as an argument. We don't usually refer to them as function literals, however, but rather *function expressions*.

Each literal value usually has a constructor function associated with it.

value	typeof	constructor
1	'number'	Number()
3.14	'number'	Number()
some text.	'string'	String()
▶ *[]*	'object'	Array()
▶ *{}*	'object'	Object()
true	'boolean'	Boolean()
ƒ ƒ() {}	'function'	Function()

The `typeof` function can be used to determine type of the literal. For example `typeof 1` will return string `"number"`, and `typeof {}` will return string `"object"`. This doesn't mean, however that its an object-literal (for example `typeof new Array` also returns `"object"` as does `typeof new Number`.)

It's a bit unfortunate that there is no `"array"` (you will get `"object"` instead), but there is a classic workaround. To check if a value *is* an array, check if its `typeof` returns `"object"`, but also check for presence of `length` property – because it exists only on objects of type `Array`.

You can instantiate a value using its constructor function. But that's rather uncommon practice. Most of the time you will define basic values using their literal notation.

3.4 Variables

Value Placeholders

Variables are placeholder names for different types of values.

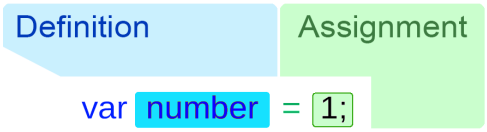


Figure 3.8: Variable declaration is definition + assignment.

Keywords for defining variables include: `var`, `let` and `const`: but they don't determine variable's type, only *how* they can be used. We'll go over the rules in more detail at a later time. Here are some examples:

```
001 var legacy = 1;                    // using legacy var keyword
002 let number = 1;                    // assign a number
003 let string = "Hello.";             // assign a string
004 let array = [];                    // assign an array literal
005 let object = {a:1};                // assign object literal
006 let json = {"a":1};                // assign JSON1 (object)
007 let json2 = '{"a":1}';             // assign JSON2 (string)
008 let boolean = true;                // assign boolean
009 let inf = Infinity;                // assign infinity (number)
010 let func = function(a) {return a};// assign a function
011 let arrow = (a) => { a };          // assign arrow function
012 let fp = a => a;                   // assign arrow expression
013 let n = new Number(1);             // assign number object
014 let o = new Object();              // assign blank object
```

Figure 3.9: Here are some of the most common assignments you will come across.

When you assign `1` to a variable name the **type** of that variable automatically becomes "number". If the value were a string, the variable type would be "string".

Dynamic Typing

JavaScript is a dynamically-typed language. It means that variables created using `var` or `let` keywords can be dynamically re-assigned to a value of another type at some point later in your JavaScript program.

In statically-typed languages doing that would generate an error.

Definition Or Declaration?

In the previous diagram we looked at a JavaScript variable declaration.

Some will argue that the definition is the declaration. But this type of logic comes from statically typed languages, of which JavaScript is not. In statically typed languages the declaration determines the type of the variable – it's what the compiler needs to allocate memory for the variable type (*left hand side*). But JavaScript is a dynamically typed language – the variable type is determined by the the type of value itself (*right hand side*).

Hence, the confusion. Is the left side the declaration, definition or both? These types of details are more relevant in statically typed languages, but in JavaScript it might not make much sense.

3.5 Passing Values By Reference

Copying data from place to place is a common operation in computing. It is natural to think that when we assign a value to a variable from another variable, a copy is made.

But JavaScript assigns values **by reference** without actually making a copy of the original value. Here is an example:

```
001 let x = { p: 1 };      // create new variable x
002 let y = x;             // y is a reference to x
003 x.p = 2;               // change original value in x
004 console.log(y.p);      // 2
```

Here we created a variable `x` and assigned object literal `{p: 1}` to it.

This means that from now on the value of `x.p` will be equal `1`;

We then created a new variable called `y`, and assigned `x` to it.

Now `x` has become a **reference** to `y`, not a copy.

From now on, any changes made to `x` will be also reflected in `y`.

This is why when we changed value of `x.p` to 2, `y.p` was *also* changed.

You can say that now `y` "*points*" to the original *object-literal* defined in `x`.

Only one copy of `{p: 1}` existed in computer memory all along from start to finish of this code block. Multiple assignments are chained by reference:

```
001 let a = { p: 1 };      // create new variable a
002 let b = a;             // b is a reference to a
003 let c = b;
004 let d = c;
005 let e = d;
006 let f = e;
007 let g = f;
008
009 a.p = 5;              // change original value in a
010
011 console.log(g.p);      // 5 now also in g.p
```

Figure 3.10: A chain of references without a single copy of original value.

3.6 Scope Quirks

JavaScript has two known quirks when it comes to scope rules, that you might want to know about to save debugging time later.

Quirk 1 – let and const inside function vs. global variable

A variable defined using **let** or **const** keywords inside a function cannot coexist with global variable of the same name.

```
001 let a = "global a";
002 let b = "global b";
003
004 function x(){
005   console.log("x(): global b = " + b); // "global b"
006   console.log("x(): global a = " + a); // ReferenceError
007   let a = 1; // doesn't hoist
008 }
009
010 x();
```

Figure 3.11: ReferenceError will happen if local variable **a** is defined inside the function body using either **let** or **const** keywords.

The **let** keyword doesn't hoist definitions, and we have a global variable **a**, so logically, inside function **x()** variable **a** should be taken from global scope, before it is defined later with **let a = 1** but that's not what happens.

If variable **a** already exists inside a function (and it's defined using **let** or **const** keywords) then using **a**, prior its definition within the function will produce ReferenceError, even if global variable **a** exists!

Quirk 2 – var latches onto window/this object, let and const don't

In global scope **this** reference points to instance of **window** object / global context.

When variables are defined using **var** keyword they become attached to **window** object, but variables defined using **let** (and **const**) are not.

```
001 console.log(this === window); // window
002
003 var c = "c"; // latches on to window ("this" in global scope)
004 let d = "d"; // exists separately from "this"
005
006 console.log(c);         // "c"
007 console.log(this.c);    // "c"
008 console.log(window.c); // "c"
009
010 console.log(d);         // "d"
011 console.log(this.d);    // undefined
012 console.log(window.d); // undefined
```

Chapter 4

Statements

4.0.1 Evaluating Statements

A statement is the smallest building block of a computer program. In this chapter we will explore a few common cases.

Definitions made with **var**, **let** or **const** keywords return `undefined` because they behave only as value assignments: the value is simply stored in the variable name:

```
001 let a = 1;                                  // undefined
```

Figure 4.1: The assignments statement itself produces `undefined`, while the value is stored in variable `a`.

If, however, the assigned variable `a` is used as a stand-alone statement afterwards, it will produce value of 1:

```
002 a;                                          // 1
```

Figure 4.2: A statement that produces a single value other than `undefined` can be referred to as an *expression*.

Statements *usually* produce a value. But when there isn't anything to return, a statements will evaluate to `undefined`, which can be interpreted as "no value."

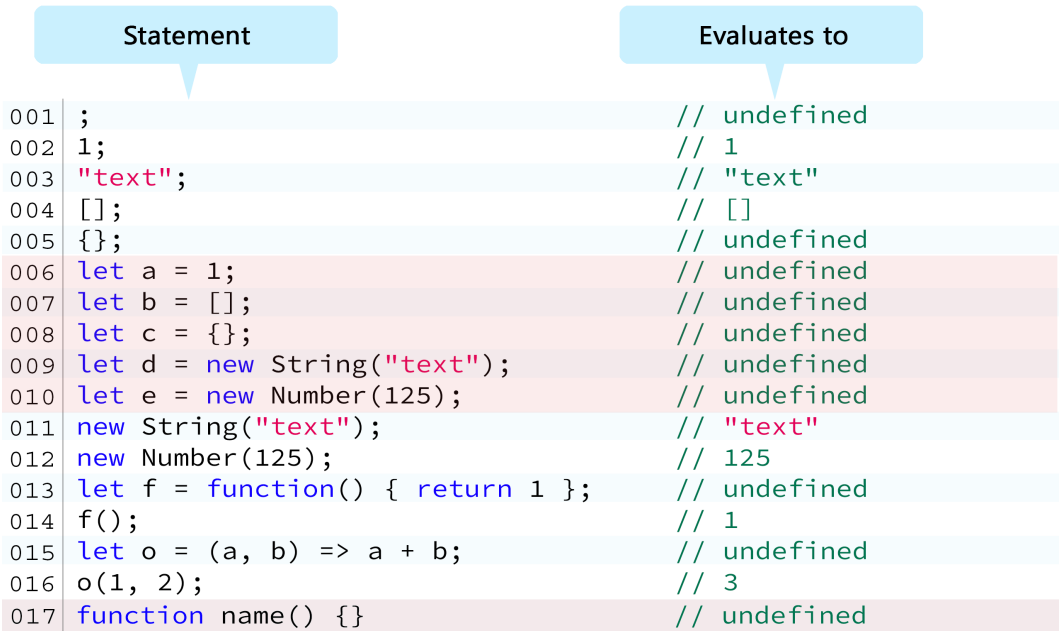


Figure 4.3: An empty statement with semicolon evaluates to `undefined`. Any statement that doesn't produce a value will evaluate to `undefined` – variable assignments (006–010) or function definitions (017), for example.

Some evaluation rules make sense, but special cases should probably be just memorized. For example, what would it mean to evaluate an empty *object literal*? According to JavaScript it should evaluate to `undefined`.

Yet, empty array brackets `[]` (a close relative to empty object literal) evaluate to an empty array: `[]`, and not `undefined`.

4.0.2 Expressions

Here is an *expression*: `1 + 1` that produces the value of 2:

```
003 1 + 1;                                  // 2
```

Figure 4.4: Expressions don't have to be variable definitions. You can create them by simply using some literal values in combination with operators.

There is another distinct types of an expression in JavaScript:

```
013 let f = function() { return 1 };        // undefined
014 f();                                    // 1
```

Function `f()` evaluates to value `1`, because it returns `1`. This is why `f()` is often referred to as a *function expression*.

Chapter 5

Primitive Types

Primitive Types

JavaScript has 7 primitive types: `null`, `undefined`, `number`, `bigint`, `string`, `boolean` and `symbol`. Primitives helps us work with simple values such as strings, numbers and booleans. Let's take a look at some of their possible values:

	type	values	constructor function
	null	null	*none*
	undefined	undefined	*none*
	number	123 3.14	Number()
ES10	bigint	123n 256n	BigInt()
	string	"Hello"	String()
	boolean	true false	Boolean()
ES6	symbol		*none*

Some of the primitives have a constructor function associated with it.

Here's a number of primitives assigned to several variable names:

```
001 let a = undefined;          // undefined
002 let b = null;               // null
003 let c = 12;                 // integer number
004 let d = 4.13;               // floating point number
005 let e = 100n;               // big integer (values over 2^53)
006 let f = "Hello.";           // text string
007 let g = Symbol();           // create symbol
008 console.log(typeof f);      // "symbol"
```

Numbers, strings and booleans are basic value units. You can write them out in literal form: a number can be `123` or `3.14`, a string can be `"string"`, or a template string: `` `I have ${number} apples.` `` (note the back-tick quotes, which allow you to embed variables into the string dynamically.) A boolean can only be either `true` or `false`. You can combine primitive types using operators, pass them to functions or assign them as values to object properties.

Number(), **BigInt()**, **String()** and **Boolean()** are primitive constructor functions. We'll explore constructor functions and classes at a proper time in the book. First, let's briefly go over each primitive individually.

5.0.1 boolean

possible values
`true` `false`

The **boolean** primitive can be assigned either **true** or **false** value.

typeof	constructor
`"boolean" string`	`new Boolean(value)`

5.0.2 null

typeof	constructor
`"object" string`	none

Running **typeof** operator on **null** will say it's an "object".

Some believe this is a bug in JavaScript because **null** is not an object since it doesn't have a constructor. And they are probably right...

5.0.3 undefined

typeof	constructor
`"undefined" string`	none

Undefined is a type of its own. It's not an object. Just a value JavaScript will use when you named a variable but don't assign a value to it. Your hoisted variables will also be automatically assigned a value of undefined.

5.0.4 number

possible values							
-1	5	7	1.14	9.66e+0	Infinity	-Infinity	NaN

The **number** primitive helps us work with values in the numeric domain.

You can define negative and positive values, decimals (*more commonly known as floating-point numbers.*) There is even a negative and positive `Infinity` value. This makes more sense if you have some background in math.

`NaN` is technically a non-numeric value a statement can evaluate to. It's available directly from the `Number.NaN` But literally, it is exactly what it says it is: neither `"number"` primitive nor `Number()` object. (It could be a `"string"`, for example.)

typeof	constructor
"number" string	new Number(*value*)

Using **typeof** operator on a numeric value will produce `"number"` (It helps to note that the return value is in string format.)

```
001 // The typeof operator returns value type in string format
002 typeof -1;          "number"
003 typeof 5;           "number"
004 typeof 7;           "number"
005
006 //Using Number constructor function to create a number
007 let number = new Number(7);   "object"
008 typeof number;                "object"
009 typeof number.valueOf();      "number"
```

This example shows distinction between primitive *literal value* (`-1`, `5`, `7`, etc.) and the `Number` object. Once instantiated, the value is no longer exactly a literal but an object of that type.

To get "number" type from the object use `typeof` on the `valueOf` method as seen in the previous example `typeof number.valueOf();`

5.0.5 ES10 bigint

value
`1n` `32000n` `9007199254740991n` `9007199254740993`

BigInt was added in EcmaScript10 and wasn't available until Summer 2019.

In the past the maximum value of a number created using a number literal or the **Number()** constructor was stored in `Number.MAX_SAFE_INTEGER` and was equal to 9007199254740991.

A bigint type allows you to specify numbers greater than `Number.MAX_SAFE_INTEGER`.

typeof	constructor
`"bigint"` string	`new BigInt(value)`

```
const limit = Number.MAX_SAFE_INTEGER;
// 9007199254740991

limit + 1;
// 9007199254740992

limit + 2;
// Still 9007199254740992 (exceeded MAX_SAFE_INTEGER + 1)

const small = 1n; // 1n
const larger = 9007199254740991n; // 9007199254740991n

const integer = BigInt(9007199254740991); // init as number
// 9007199254740991n

const big = BigInt("9007199254740991"); // init as string
// 9007199254740991n

big + 1;
// 9007199254740993n - exceeds older numeric limit
```

5.0.6 typeof

Difference between numeric types:

```
typeof 10;  // 'number'
typeof 10n; // 'bigint'
```

Equality operators can be used between the two types

```
10n === BigInt(10); // true
10n == 10;          // true
```

Math operators only work within their own type

```
200n / 10n; // 20n
200n / 20;  // Uncaught TypeError:
            // Cannot mix BigInt and other types,
            // use explicit conversions
```

Leading - works, but + doesn't

```
-100n // -100n
+100n // Uncaught TypeError:
      // Cannot convert a BigInt value to a number
```

5.0.7 string

value
`"text"` `'text'` `` `text` `` `` `Cat "Felix" knows best` ``

The **string** value is defined using any of the available quote characters: double quotes, single quotes, and back-tick quotes (Located on tilde key.) You can nest double quotes inside single quotes, and the other way around.

typeof	constructor
`"string"` string	`new String(value)`

Running `typeof` on a string value returns `"string"`:

```
001 // The typeof operator returns value type in string format
002 typeof "text";                  "string"
003 typeof "JavaScript Grammar";    "string"
004 typeof "username" + 25;         "string"
```

You can also use `String` constructor function to build an object of string type:

```
001 // Using Number constructor creates an object of that type
002 let string = new String("hi."); "object"
003 typeof string;                  "object"
004 typeof string.valueOf();        "string"
```

Note that the first `typeof` returns `"object"`, because at this point the object is instantiated (this is different from the primitive's literal value which is still just a `"string"` primitive). To get the value of the instantiated object use `valueOf()` method and use `typeof string.valueOf()` to determine the object's type.

5.0.8 ES6 Template Strings

Strings defined using the backtick quotes have special function.

You can use them to create Template Strings (also known as Template Literals) to embed dynamic variable values inside the string:

```
Define a variable:
  let apples = 10;
Embed variable inside template string:
  `There are ${apples} apples in the basket.`
Result:
   There are 10 apples in the basket.
```

The back-tick cannot be used to define an **object-literal** property name (*You still have to use either single or double quotes.*)

JSON format *requires* **double quotes** around object's property names (back-ticks won't do any good here either, without generating an error):

```
001 // Attempt creating an object literal
002 let object_literal = {`a` : 1}; // Unexpected token error
003
004 // Attempt creating a well-formed JSON format string:
005 let json1 = '{`a` : 1}'; // malformed json(back-tick quotes)
006 let json2 = '{ a  : 1}'; // malformed json(no quotes)
007 let json3 = "{'a' : 1}"; // malformed json(single quotes)
008 let json4 = '{"a" : 1}'; // correct json  (' + double quotes)
009 let json5 = `{"a" : 1}`; // correct json  (` + double quotes)
```

We'll take a look at JSON in greater detail in a later chapter.

Creative Use Case

Template strings can be used to solve the problem of forming a message that has proper language form, based on a dynamic number. One of the classic cases is forming an alert message sentence.

```
001 for (let alerts = 0; alerts < 4; alerts++) {
002   let one = (alerts == 1);      // one alert?
003   let is = one ? "is" : "are"; // choose between "is" or "are
004   let s = one == 1 ? "" : "s"; // trailing "s" or empty space
005
006   // Form the message in proper English:
007   let message = `There ${is} ${apples} alert${s}.`;
008   console.log(message);
009 }
```

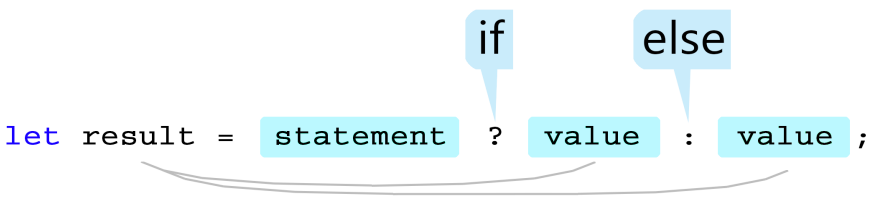
Whenever there is only `1` alert, the trailing `"s"` in the word `"alerts"` must be removed. But we don't want to create a second string just to cover one case.

Instead, we can calculate it dynamically. We also need to decide which verb should be used (`"is"` or `"are"`) based on the number of alerts.

```
There are 0 alerts.
There is 1 alert.
There are 2 alerts.
There are 3 alerts.
> |
```

Here the **ternary operator** consisting of `?` and `:` is used.

You can think of **ternary operator** as an inline **if**-statement. It doesn't need {} brackets because it doesn't support multiple statements:



Question mark can be interpreted as `"if-then"` or as `"if the previous statement evaluates to true"` and the colon `:` can be interpreted as `"else"`.

5.0.9 ES6 Symbol

typeof	constructor
"symbol" string	none

The **Symbol** primitive provides a way to define a completely unique key.

Symbol doesn't have a constructor and cannot be initialized using **new** keyword:

```
001 let sym = new Symbol('sym'); // TypeError
```

Instead, just an assignment to Symbol will create a new symbol with a unique ID:

```
001 let sym = Symbol('sym'); // symbol created
```

The ID, however, is not the used-defined string `"sym"`, it is created internally. This is demonstrated in the following example.

At first it might be surprising that the following statement evaluates to **false**:

```
001 Symbol('sym') === Symbol('sym'); // false
```

Whenever you call **Symbol('sym')** a unique symbol is created. The comparison is made between two logically distinct IDs and therefore evaluates to false.

Symbols can be used to define private object properties. This is not the same as regular (public) object properties. However, both public and private properties created with symbols can live on the same object:

```
001 let sym = Symbol('unique');
002 let bol = Symbol( distinct');
003 let one = Symbol('only-one');
004 let obj = { property: "regular property",
005             [sym]: 1,
006             [bol]: 2 };
007 obj[one] = 3;
```

Here we created an object **obj**, using *object literal* syntax, and assigned one of its properties **property** to a string, while second property was defined using the **[sym]** symbol created on the first line. **[sym]** was assigned value of 1. Second symbol property **[bol]** was added in the same way and assigned value of 2.

Third object symbol property **[one]** was added directly to the object via **obj[one]**.

Printing the object shows both private and public properties:

```
001 console.log(obj);
002 property: "regular property"
003 Symbol(distinct): 2
004 Symbol(only-one): 3
005 Symbol(unique): 1
```

Private (symbol-based) properties are hidden from **Object.entries**, **Object.keys** and other *iterators* (for example **for...in** loop):

```
004 for (let prop in obj)
005     console.log(prop + ": " + obj[prop]);
006 // property: regular property
007
008 console.log(Object.entries(obj));
009 // (2) ["property", "regular property"]
010 // length: 1
```

In addition symbol properties are also hidden from **JSON.stringify** method:

```
001 console.log(JSON.stringify(obj));
002 // {"property":"regular property"}
003
```

Why would we want to hide symbol-based properties from JSON stringify?

Actually it makes sense. What if our object needs to have private properties that are only relevant to how that object *works*, and not what data it *represents*? These private properties can be used for miscellaneous counters or temporary storage.

The idea behind private methods or properties is to keep them hidden from the outside world. They are only needed for internal implementation. Private implementation is rarely important when it comes to marshalling objects.

But symbols *can* be exposed via **Object.getOwnPropertySymbols** method:

```
012 console.log(Object.getOwnPropertySymbols(obj));
013 // [Symbol(unique)]
014 // 0: Symbol(unique)
015 // 1: Symbol(distinct)
016 // 2: Symbol(only-one)
017 // length: 3
```

Note that you probably shouldn't use **Object.getOwnPropertySymbols** to expose properties that are intended to be private. Debugging should be the only use case for this function.

You can use symbols to separate your private and public properties. This is like separating "goats from the sheep" because even though they provide similar functionality, symbols will not be taken into account when used in iterators or `console.log` function.

Symbols can be used whenever you need unique IDs. Hence, they can also be used to create constants in enumerable lists of IDs:

```
001 const seasons = {
002     Winter: Symbol('Winter');
003     Spring: Symbol('Spring');
004     Summer: Symbol('Summer');
005     Autumn: Symbol('Autumn');
006 }
```

Figure 5.1: Enumerating seasons.

Global Symbol Registry

As we saw earlier `Symbol("string") === Symbol("string")` is **false** because two completely unique symbols are created.

But there is a way to create string keys that can overwrite symbols created using the same name. There is a global registry for symbols, that can be accessed using methods **Symbol.for** and **Symbol.keyFor**.

```
let sym = Symbol.for('age');
let bol = Symbol.for('age');

obj[sym] = 20;
obj[bol] = 25;

console.log(obj[sym]);
// 25
```

The private symbolic object property **obj[sym]** outputs the value of 25 (which was originally assigned to **obj[bol]**) when it was defined, because both variables **sym** and **bol** are tied to the same key "age" in global symbol registry.

In other words the definitions share the same key.

Constructors And Instances

There is a distinction between **constructors** and **instances**. The constructor function is the definition of a custom object *type*. The instance is the object that was *instantiated* from that constructor function using the **new** operator.

Let's create a custom **Pancake** constructor, containing one object property `number` and one method `bake()` which will increase pancake number by 1 when called:

```
001 // Create a custom constructor function
002 let Pancake = function() {
003     // Create object property:
004     this.number = 0;
005     // Create object method:
006     this.bake = function() {
007         console.log("Baking the pancake...");
008         // Increase number of pancakes baked:
009         this.number++;
010     };
011 }
```

Note that properties and methods are attached to the object via `this` keyword

The constructor is only a design of the object type. In order to start using it, we have to *instantiate* it. When we do that, an instance of the object is created in computer memory:

```
001 // Instantiate pancake maker:
002 let pancake = new Pancake();
```

Let's bake 3 pancakes by using **bake()** method which increments pancake counter:

```
001 // Bake 3 pancakes:
002 pancake.bake(); // "Baking the pancake..."
003 pancake.bake(); // "Baking the pancake..."
004 pancake.bake(); // "Baking the pancake..."
```

3 pancakes successfully baked! Let's take a look at **pancake.number** now:

```
001 console.log( pancake.number ); // 3
```

You can look up the constructor function's type. The constructor function `Pancake` is an object of type `Function`. This is true of all custom objects. It makes sense because the function itself is the constructor:

```
001 console.log(Pancake.constructor); // function Function() {}
```

But, if you output constructor via the instantiated object, it will show you the entire function body in *string* format:

```
001 console.log(pancake.constructor);
002
003 let Pancake = function() {
004     // Create object property:
005     this.number = 0;
006     // Create object method:
007     this.bake = function() {
008         console.log("Baking the pancake...")
009     };
010 }
```

You can actually create a brand new function by supplying the body in string format to **Function** constructor:

```
001 let body = "console.log('Hello from f() function!')";
002
003 let f = new Function(body);
004
005 f(); // Hello from f() function!
```

This tells us that `Function` is the constructor for creating JavaScript functions.

But when we created our own **Pancake** function, Pancake became the constructor of our custom class that we could also initialize using the **new** keyword.

5.0.10 Executing Methods On Primitive Types

Parenthesis And Object Property Access

The parenthesis operator gives you control over which statement should evaluate first. That's its primary purpose.

For example statement `5 * 10 + 2` is not the same as `5 * (10 + 2)`.

But sometimes it is used to access a member method or property. Which is demonstrated in the next source code listing.

You can execute methods directly on the literal values of primitive types. Which automatically converts them to objects, so that the method can be executed.

In some cases – like with the primitives of type `"number"` – we must first wrap the literal value in parenthesis, or you'll freeze your program.

```
001 // We cannot execute Number object methods directly on literal values
002 1.toString();                    // this will freeze execution flow
003
004 // But by using parenthesis, you can convert numeric literal to object
005 (1).toString();                   // "1"
006
007 // You don't need parenthesis to execute methods on strings:
008 "hello".toUpperCase();            // "HELLO"
009
010 // But if you use them, it still works:
011 ("hello").toUpperCase();          // "HELLO"
012
013 // Execute toString() method directly on Number object:
014 new Number(1).toString();         // "1"
```

A literal *is* just a literal value. By accessing its properties, it turns into a reference to the object instance so you can execute object methods on that value.

Chaining Methods

Because in JavaScript functions can return **this** keyword, or any other value, including functions, it's possible to chain multiple methods using the dot operator.

```
"hello".toUpperCase().substr(1, 4); // "ELLO"
```

Chapter 6

Type Coercion Madness

When learning JavaScript from scratch you may be puzzled by some decisions made by the language when it comes to evaluating statements.

For example, what will happen if we sporadically add up different types of values and stitch them together using the + operator?

```
001 console.log(null + {} + true + [] + [5]);
```

```
null[object Object]true5
> 
```

A string? This might seem confusing. After all, not a single value in this statement is a string! So how did that happen?

Answer: When + operator encounters objects of incompatible type, it will attempt to coerce those objects to their values in string format. In this case, leaving us with a new statement: `"null[object Object]" + true + [] + [5]`.

Furthermore, when + operator encounters a string at least on one side of the operator, it will try to coerce the other side to string and perform string addition.

Calling **.toString** on `true` results in `"true"`. Calling **.toString** on empty array brackets `[]` when the other side of operator is also a string evaluates it to `""` which is why it appears missing from the result. And finally adding `[5]` to a string calls `[5]`**.toString** which results in `"5"`.

6.0.1 Examples of Type Coercion

Here are some classic examples of type coercion.

```
001 //classic coersion cases sometimes produce surprising results
002 let a = true + 1;                  // becomes 1 + 1 or 2
003 let b = true + true;               // becomes 1 + 1 or 2
004 let c = true + false;              // becomes 1 + 0 or 1
005 let d = "Hello" + " " + "there.";// becomes "Hello there."
006 let e = "Username" + 1523462;     // becomes "Username1523462"
007 let f = 1 / "string";             // NaN (not a number)
008 let g = NaN === NaN;              // becomes false
009 let h = [1] + [2];                // becomes "12" <string>
010 let i = Infinity;                 // remains Infinity
011 let j = [] + [];                  // becomes "" <string>
012 let k = Infinity;                 // is Infinity <number>
013 let l = Infinity;                 // remains Infinity
014 let m = Infinity;                 // remains Infinity
015 let n = Infinity;                 // remains Infinity
016 let o = [] + {};                  // becomes [object Object]
```

JavaScript will try to come up with best value available if you supply meaningless combinations of types to some of its operators.

After all, what would it mean to "add" an *object literal* {} to an *array* []? Exactly – it doesn't make any sense. But by evaluating to object [] at least we don't break the code in that one little odd case where it *may* happen.

This safety mechanism will prevent the program from breaking. In reality, however, these types of cases will almost never happen. We can treat majority of these cases as examples – not something you should be actually trying to do in code.

Type Coercion In Constructors

Coercion also occurs when we provide an initialization value to a type constructor:

```
let A = Boolean(true);              // true
let B = Boolean([]);                // true
let C = Boolean({});                // true
```

In the last two cases we supplied an *array literal* {} and an *object literal* [] to `Boolean` constructor. What does this mean? Not much, but the point is that at least it evaluates to `true` in this odd case.

This is just a safety net to prevent bugs.

```
// Initialize some booleans based on an argument
let p = Boolean(false);             // false
let q = Boolean(NaN);               // false
let r = Boolean(null);              // false
let s = Boolean(undefined);         // false
let t = Boolean('');                // false
let u = Boolean(0);                 // false
let v = Boolean(-0);                // false
```

Meaningless values still evaluate to either `true` or `false`, because these are the only values available for boolean types.

Other built-in data type constructors behave in the same way. JavaScript will try to coerce to an ideal value specific to that type.

Type Coercion

Coercion is the process of converting a value from one type into another. For example, number to string, object to string, string to number (if the entire string consists of numeric characters) and so on...

But when values are used together with different operators not all cases are straightforward to the untrained eye.

To someone new to the language, the following logic might seem obscure:

```
001 [] == [];      // false
```

Let's say that it is false because two instances of `[]` are not the same, because JavaScript `==` operator tests objects by reference and not by value.

```
001 let a = [];
002 a == a;      // true
```

But this statement evaluates to **true** because variable a points to the same instance of the array literal. They refer to the same location in memory.

But what about cases like this? Even though you would never write code like this in production environment, it calls for understanding of type coercion:

```
002 [] == ![];     // true
```

JavaScript will often coerce different types of values to either strings or numbers. The Boolean type is no exception:

```
003 true + false; // 1
```

The above statement is the same as `1 + 0`. And here's the absolute classic:

```
001 NaN == NaN; // false
```

These types of cases might appear bizarre at first, but as your knowledge of **types** and **operators** deepens it will start to make a lot more sense.

Let's start simple. The unary plus and minus operators force the value to a number. If the value is not a number, NaN is generated:

```
001 const s = "text";
002 console.log(-s); // NaN
```

Here unary minus (-) struggles to convert the string `"text"` to a number. What does `-"text"` mean anyway? So it returns `NaN` because "text" is not a number.

Here is the same logic demonstrated using the **Number** type function:

```
001 Number("text"); // NaN ("text" is not a numeric string)
002 Number("1");    // 1 ("1" is a numeric string)
```

But when *unary* minus (-) is applied to a number, it produces expected value:

```
001 const a = 1;
002 console.log(-a); // -1
003
004 const b = 1;
005 console.log(+b); // 1
```

This rule is specific to the **unary** operator.

Number And String Arithmetics

Naturally the arithmetic + operator requires two values.

```
001 5 + 7; // 12
```

If both values are integers, **arithmetic** operation is performed. If one of them is a string then coercion happens and **string** addition is invoked.

If the type of the two values provided to the arithmetic + operator is different, this conflict must be resolved. JavaScript will use **type coercion** to change one of the values before evaluating the entire statement to a more meaningful result.

What will happen if left value is a string and right value is a number?

```
"1" + 1; // ???
```

Here + is treated as a string addition operator. The right value is converted to `"1"` via `String(1)` and then the statement is evaluated as follows:

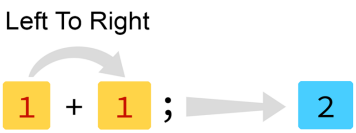
```
"1" + "1"; // "11"
```

In JavaScript there are actually three + operators: unary, arithmetic and string.

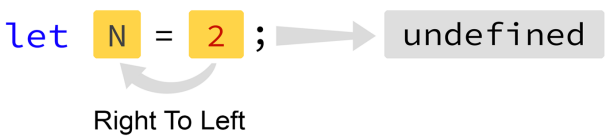
Here JavaScript treats + not as the unary addition operator, but as the arithmetic addition operator instead. But... when it sees that one of the values is a string, it invokes the string addition operator. It makes no difference whether the string is on the left or right side. The statement still evaluates to a string:

```
1 + "o1"; // "1o1"
```

Operators follow specific associativity rules. Like + and most other operators, the arithmetic addition operator (+) is evaluated from **left to right**:



But the assignment operator is evaluated in **right to left** order:



Note that in example above, while `N` is assigned value of 2, the statement itself evaluates to **undefined**.

6.0.2 Adding Multiple Values

Often you will encounter statements tied together by multiple operators. What should the following statement evaluate to?

First, all of the purely numeric values will be combined, ending up with the sum of `5` on the left hand side and `""` on the right hand side:

But this is still not enough to produce the final result. Adding a numeric value to a string value will coerce the numeric value to a string and then add them together:

Finally we arrive at `"5"` in string format.

When adding numbers and strings, numeric values always take precedence. This seems to be a trend in JavaScript. In the next example we we will compare numbers to strings using the equality operator. JavaScript chooses to convert strings to numbers first, instead of numbers to strings.

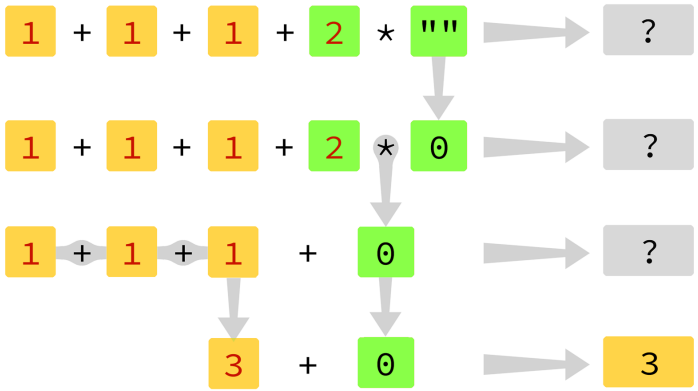
6.0.3 Operator Precedence

Some operators take precedence over others. What this means is that multiplication will be evaluated before addition.

Let's take this statement for example:

Several things will happen here.

The string `""` will coerce to `0` and `2 * 0` will evaluate to `0`.



After multiplication, the numbers `1 + 1 + 1` will be added up to produce `3`.

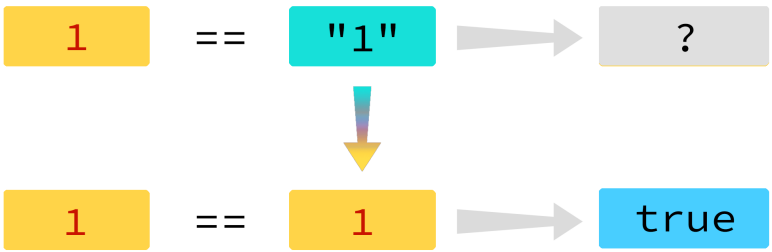
Finally: `3 + 0` will evaluate to `3`.

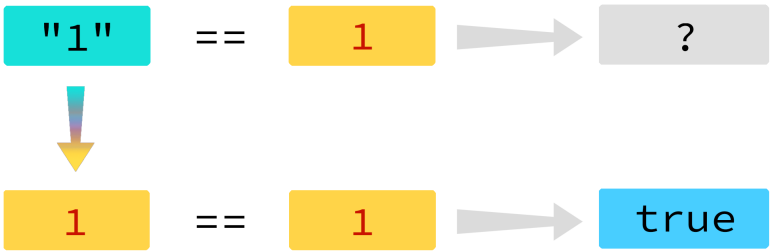
6.0.4 String To Number Comparison

When it comes to equality operator == numeric strings are evaluated to numbers in the same way the **Number(string)** function evaluates to numbers (or **NaN**).

According to EcmaScript specification, coercion between a string and a numeric value on both sides of the == operator can be visualized as follows.

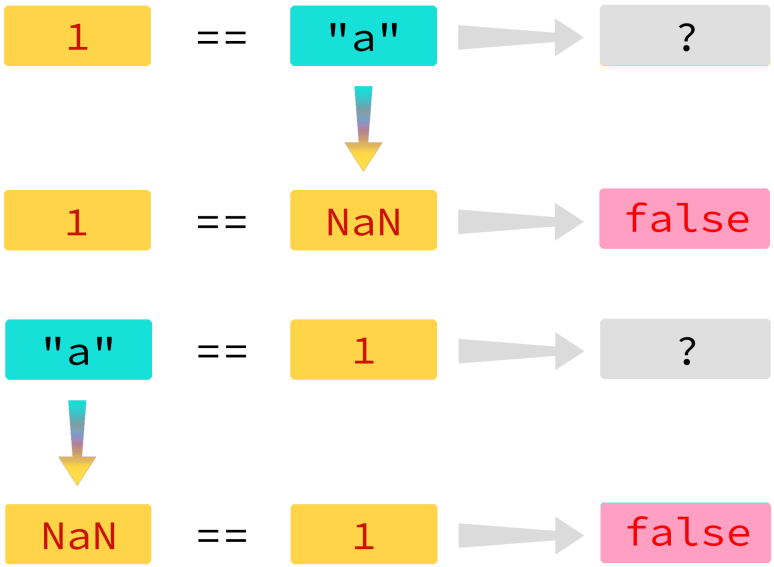
Comparing Numeric String To Number





Comparing Non-Numeric String To Number

If the string does not contain a numeric value, it will evaluate to **NaN** and therefore further evaluating to **false**:



Other Comparisons

Other comparisons between different types (boolean to string, boolean to number, etc) follow similar rules. As you continue writing JavaScript code, you will eventually develop intuition for them and it will become second nature.

The operator precedence and associativity table on the next page might help you when things get tough.

6.0.5 Operator Precedence & Associativity Table

There are roughly 20 operator **precedence** levels. Parenthesis () overrides the natural order. Red values are first in **associativity** order: for example, subtraction operator subtracts blue from red. Assignment operators follow right to left order.

	Operator Precedence	Visual
19	Grouping	(●)
18	Member Access	●.●
	Computer Member Access	●[●]
	new (with argument list)	new ●(●)
17	Function Call	●(●)
	new (without argument list)	new ●
16	Postfix Increment	●++
	Postfix Decrement	●--
15	Logical NOT	!●
	Bitwise NOT	~●
	Unary Plus	+●
	Unary Negation	-●
	Prefix Increment	++●
	Prefix Decrement	--●
	typeof	typeof ●
	void	void ●
	delete	delete ●
14	Multiplication	●*●
	Division	●/●
	Remainder	●%●
13	Addition	●+●
	Subtraction	●-●
12	Bitwise Left Shift	●<<●
	Bitwise Right Shift	●>>●
	Bitwise Unsigned Right Shift	●>>>●
11	Less Than	●<●
	Less Than or Equal	●<=●
	Greater Than	●>●
	Greater Than or Equal	●>=●

	in	●in●
	instanceof	●instanceof●
10	Equality	●==●
	Inequality	●!=●
	Strict Equality	●===●
	Strict Inequality	●!==●
9	Bitwise AND	●&●
8	Bitwise XOR	●^●
7	Bitwise OR	●|●
6	Logical AND	●&&●
5	Logical OR	●||●
4	Conditional	●?●:●
3	Assignment	●=●
		●+=●
		●-=●
		●*=●
		●/=●
		●%=●
		●<<=●
		●>>=●
		●>>>=●
		●&=●
		●^=●
		●|=●
2	yield	yield●
1	Spread	...●
0	Comma /Sequence	●,●

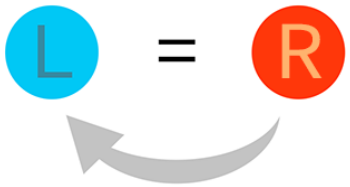
Associativity flows in either **left-to-right** or **right-to-left** direction: it determines the order of the operation, usually for operators that require more than one value.

6.0.6 L-value and R-value

In many computer languages values on the left and right side of the operator are referred to as **L-value** and **R-value**. In EcmaScript spec they are often referred to as **x** and **y** values.

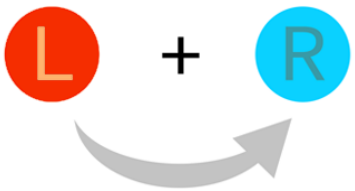
Assignment Operator

The assignment operator takes the **R-value** and transfers it over to **L-value**, which is usually a variable identifier name.



Arithmetic Addition Operator

But the arithmetic addition operator takes the **L-value** and adds **R-value** to it:



Following this logic, it is possible to use the precedence table from the previous page to figure out the order in which complex statements will be evaluated.

6.0.7 `null` vs `undefined`

The `null` primitive is not an object (*although some may believe it is,*) – so it doesn't have a built in constructor, like some of the other types. Luckily, we can (and should) use its *literal value*: `null`.

Think of `null` as a unique type for *explicitly* assigning a "nothing" or "empty" value to a variable. This way it doesn't end up *undefined*.

If you don't assign a variable to `null`, its value will be *undefined* by default:

```
001 // Define a variable, without assigning a value(not so good.)
002 let bike;
003
004 console.log(bike); // undefined
```

To same effect, you can also *explicitly* assign variable to `undefined`:

```
001 // Explicitly assign undefined as default variable name
002 let bike = undefined;
003
004 console.log(bike); // undefined
```

But that's something we should avoid. If the value is unknown at the time of variable definition it is always best to use `null` instead of *undefined*.

The `null` keyword is used to assign a *temporary* default value to a variable before it's populated with actual object data at a later time in your program.

Initialize or Update

In a real-case scenario the `null` value can help us determine whether the data needs to be initialized for the first time, or existing data merely needs to be updated.

We'll take a look at a practical example in the next source code listing.

Let's take a look at this scenario:

```
001  // Explicitly assign null as default variable name
002  let bike = null;
003
004  // Class definition
005  class Motorcycle {
006    constructor(make, model, year) {
007      this.make = make;
008      this.model = model;
009      this.year = year;
010      this.features = null;
011    }
012    getFeatures() {
013
014      // 1.) Download features from database for the first time
015      if (this.features == null) {
016        this.features = { /* get features from database */ }
017
018      // 2.) Or update features object if it already contains data
019      } else {
020        this.features = { /* get features from database */ }
021      }
022  }
023
024  // Instantiate new bike class
025  let bike = new Motorcycle("Kawasaki", "Z900RS CAFE", 2019);
026
027  // Get features from database
028  bike.getFeatures();
```

Here we assigned `null` to bike. Later at some point in code, the variable was instantiated with a real object. At no point in our program `bike` was `undefined`, even before it was initialized for the first time.

Inside the object itself, the `this.features` property was also assigned to `null`. Maybe at a later time, we can download feature list from a database. Until then, we can be sure that feature object was not yet populated.

This gives us a distinction between two classic cases: downloading data for the first time (if `this.features == null`) or updating existing data (*that has already been downloaded at some point in the past.*)

Chapter 7

Scope

7.0.1 Scope

Scope is simply the area enclosed by {} brackets. But be careful not to confuse it with the identical empty *object-literal* syntax.

There are 3 unique scope types:

The **global** scope, **block** scope and **function** scope. Each expects different things and has unique rules when it comes to variable definitions.

Event callback functions follow the same rules as function scope, they are just used in a slightly different context. Loops can also have their own block-scope.

7.1 Variable Definitions

Case-Sensitivity

Variables are case-sensitive. This means **a** and **A** are two different variables:

```
001 let a = 1;
002 let A = "hello";
003
004 console.log(a);                    // 1
005 console.log(A);                    // "hello"
```

Definitions

Variables can be defined using **var**, **let** or **const** keywords.

Of course, if you tried to refer to a variable that wasn't defined anywhere, you would generate a **ReferenceError** error "*variable name is not defined*":

```
001 console.log( apple ); // ReferenceError: apple is not defined
002
003 {
004
005 }
```

Let's use this setup to explore variable definitions using **var** keyword and hoisting.

Prior to **let** and **const** the traditional model allowed only **var** definitions:

```
001 var apple = 1;
002
003 {
004     console.log( apple ); // 1
005 }
```

Here **apple** is defined in global scope. But it can also be accessed from an inner block-scope. Anything (even a function definition) defined in global scope becomes available anywhere in your program. The value propagates into all inner scopes.

When a variable is defined in global scope using **var** keyword, it also automatically becomes available as a property on **window** object.

Hoisting

If **apple** was defined using **var** keyword inside a block-scope, it would be hoisted back to global scope! Hoisting simply means "raised" or "placed on top of".

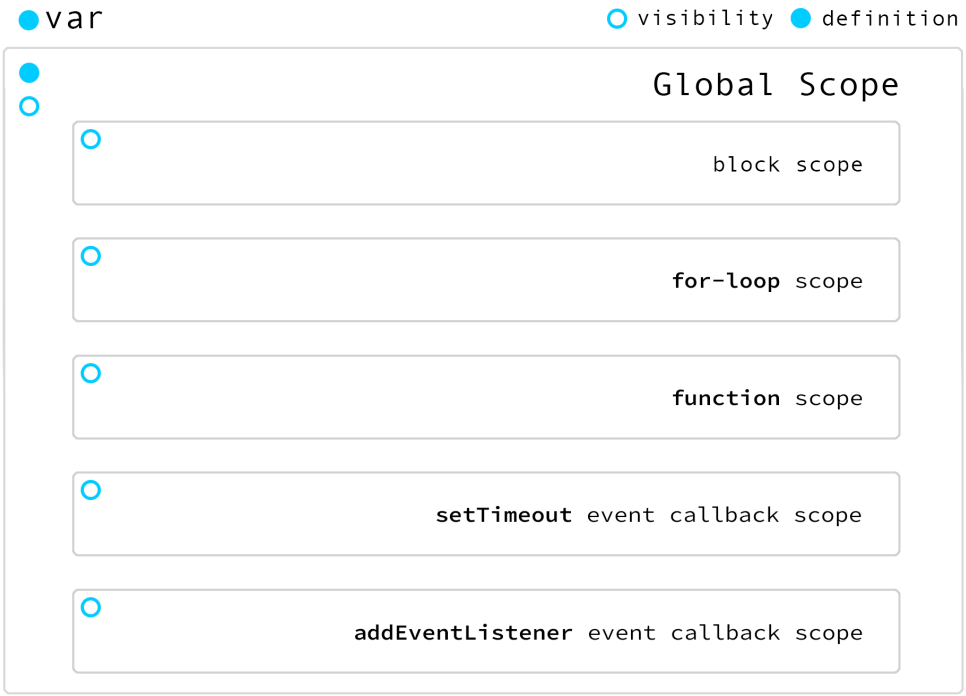
Hoisting is limited to variables defined using **var** keyword and function name defined using **function** keyword.

Variables defined using **let** and **const** are not hoisted and their use remains limited only to the scope in which they were defined.

As an exception, variables defined **var** keyword inside function-level scope are not hoisted. Commonly, when we talk about hoisting block-scope is implied.

We will talk more about hoisting in just a moment!

Likewise, variables defined in global scope will propagate to pretty much every other scope defined in global context, including **block**-level scope, **for-loop** scope, **function**-level scope, and event callback functions created using **setTimeout**, **setInterval** or **addEventListener** functions.



But what happens if we define a variable *inside* a block scope?

```
001 console.log( apple ); // undefined
002
003 {
004     var apple = 1;
005 }
```

Variable **apple** is hoisted to global scope. But the value of the hoisted variable is now **undefined** – not **1**. Only its name definition was hoisted.

Hoisting is like a safety feature. You should not rely on it when writing code. You may not retain the *value* of a hoisted variable in global scope, but you will still save your program from generating an error and halting execution flow.

Thankfully, hoisting in JavaScript is automatic. When writing your program more than half of the time, you won't even need to think about it.

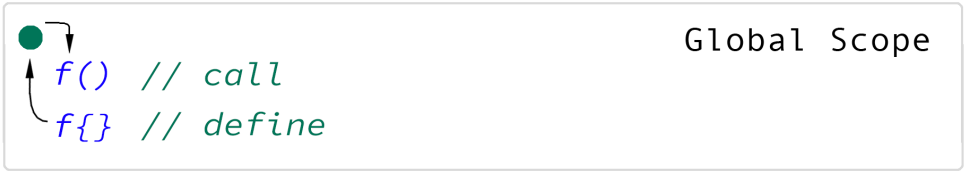
Function Name Hoisting

Hoisting also applies to function names. But variable hoisting always takes precedence. We'll see how that works in this section.

You can call a function in your code, as long as it is defined at some point later:

```
001 fun(); // Hello from fun() function.
002
003 function fun() {
004   console.log("Hello from fun() function.");
005 }
```

Note that the function was defined after it was called. This is legal in JavaScript. Just make sure you understand that it happened because of function name hoisting:



It goes without saying if the function was already defined prior to being called, there'd be no hoisting but everything would still work as planned. Statements inside a function's body are executed when the function is called by its name. Nameless functions *can* still be assigned as values themselves. (See next example.)

```
function fun() {
  console.log("Hello from fun() function 1.");
}

// The code above is the same as:
var fun = function() {
  console.log("Hello from fun() function 2.");
}
```

It is possible to assign an anonymous function expression *to* a variable name.

It's important to note, however, that anonymous functions that were assigned to variable names are *not* hoisted unlike named functions.

This valid JavaScript code will not produce a function redefinition error. The function will be simply overwritten by second definition.

Even though `fun()` was a function, when we created a new variable `fun` and assigned another function to it, we rewired the name of the original function.

Having said this, what do you think will happen if we call `fun()` at this point?

```
fun(); // ?
```

Which function body will be executed?

```
"Hello from fun() function 2."
```

You might think that the following code will produce a redefinition error:

```
function fun() {
  console.log("Hello from fun() function 1.");
}

function fun() {
  console.log("Hello from fun() function 2.");
}
```

However, this is still perfectly valid code – no error is generated. Whenever you have two function defined using `function` keyword and they happen to share the same name, the function that was defined last will take precedence.

In this case if you call `fun();` the console will output the second message:

```
"Hello from fun() function 2."
```

This actually makes sense.

In following scenario variable name will take precedence over function definitions even if it was defined *prior* to the second function definition with the same name:

```
var fun = function() {
  console.log("Hello from fun() function 1.");
}

function fun() {
  console.log("Hello from fun() function 2.");
}
```

And now let's call `fun()` to see what happens in this case:

```
fun(); // ?
```

But this time the output is:

```
"Hello from fun() function 1."
```

You can see the order in which JavaScript hoists variables and functions. Functions are hoisted first. Then variables.

Defining Variables Inside Function Scope

At this point you might want to know that variables defined inside a function will be limited only to the scope of that function. Trying to access them outside of the function will result in a reference error:

```
function fun() {
    var apple = 1;
}

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

Simple scope accessibility rules:

Figure 7.1: Here `var` is defined in Global Scope, but its value propagates into the block scope as well. What actually happens is, when block scope 1 cannot find var definition in within its own brackets, it looks for it in the parent scope. If it finds it there, it inherits its value.

Figure 7.2: Defining variables inside function scope is basically one way street ordeal. Nothing can leave the confines of a function into its parent scope.

Functions enable closure pattern, because their variables are concealed from global scope, but can still be accessed from other function scopes within them:

Figure 7.3: Nothing can get out of function scope into its outer scope. This enables the closure pattern. We'll take a look at it in just a moment!

The idea is to *protect* variables from the **global scope** but still be able to call the function from it. We'll take a look at this in greater detail in just a moment.

7.1.1 Variable Types

JavaScript is a dynamically-typed language.

The type of the variable (defined using **var** or **let** keyword) can be assigned and changed at any time during the run-time of your application, after it was already compiled by browser's JavaScript engine.

The keywords **var**, **let** and **const** do not determine the variable's type. Instead, they determine *how* the variable can be used: can it be used outside of the scope in which defined? Can it be re-assigned to another value during run-time? For example, **var** and **let** can, but **const** can't.

ES5 var

The **var** keyword is still with us from original specification. You should probably start using **let** and **const** instead. For the most part it is still available but only to support legacy code.

ES6 let

let defines a variable but limits its use to the scope in which it was defined.

ES6 const

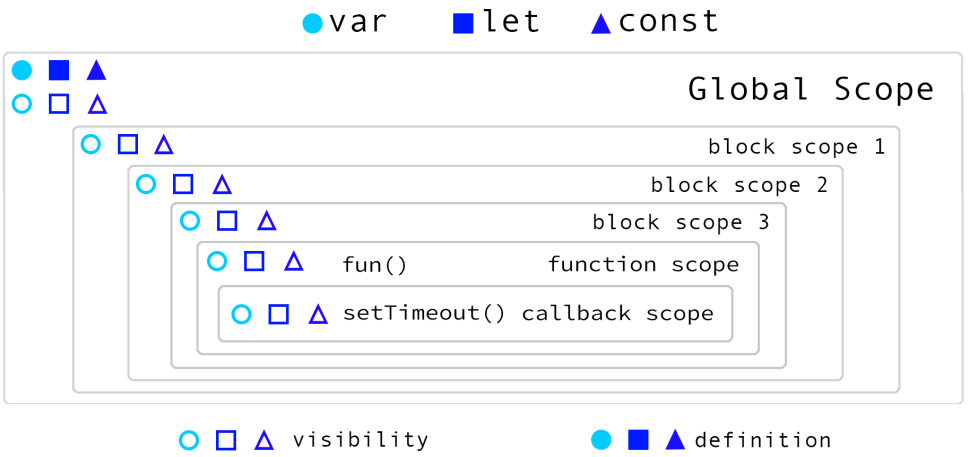
const is the same as **let** but you can't re-assign it to a new value once defined.

7.1.2 Scope Visibility Differences

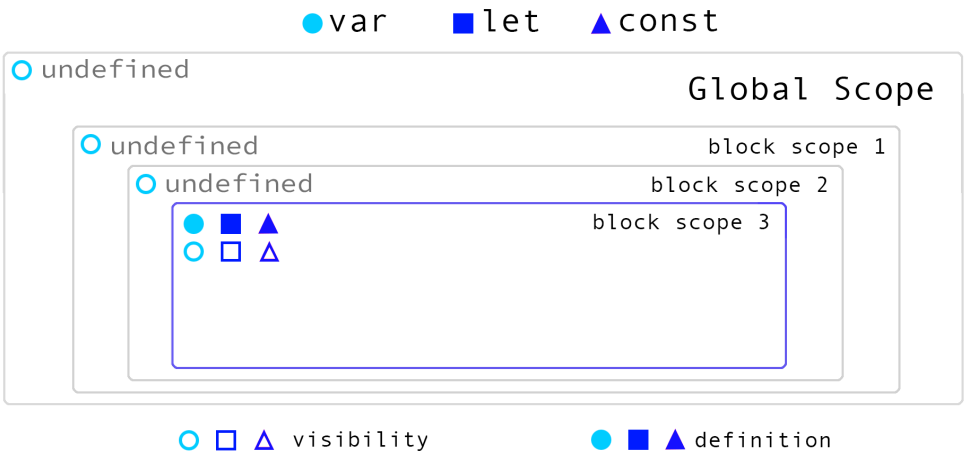
No Difference In Global Scope

When variables are defined in *global scope* there is no differences between **var**, **let** and **const** in terms of scope visibility.

They all propagate into inner block-level, function-level and event callback scopes:



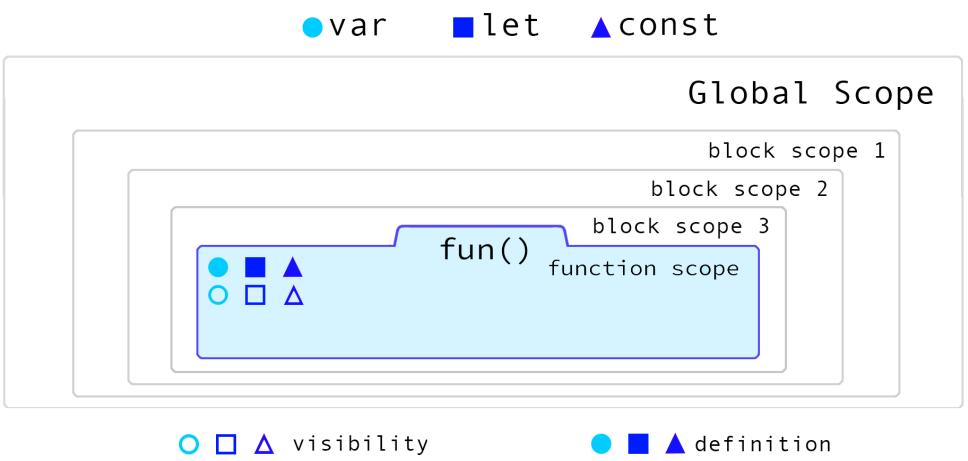
Keywords `let` and `const` limit variable to the scope in which they were defined:



Variables defined using **let** and **const** are not hoisted. Only **var** is.

In Function Scope

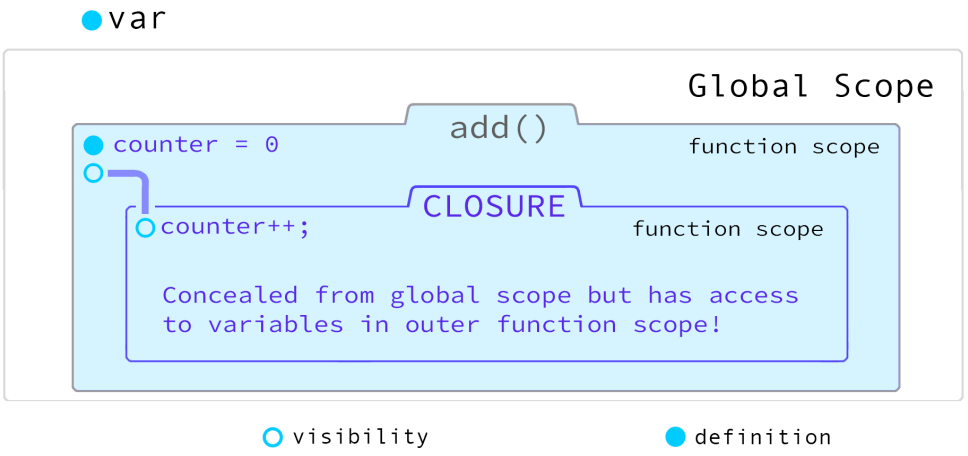
However, when it comes to functions, all variable types, including **var** remain limited to their scope:



You cannot access variables outside of the `function` scope in which they were defined regardless of which keyword was used.

Closures

A function closure is a function trapped inside another function:



Calling `add()` increments counter. This is not possible using other scope patterns.

In the previous diagram, add() returns an anonymous function which increments the counter variable that was defined in an outer scope.

Let's try to use that pattern to create our own closure:

```
001 var plus = (function () {
002   var counter = 0; // defined only once
003   return function () {
004     counter += 1;
005     return counter;
006   }
007 })();
008
009 plus(); // 1
010 plus(); // 2
011 plus(); // 3
```

The **plus()** function is defined by an anonymous function that executes itself.

Why Are We Doing This?

Inside the scope of **plus**, *another* anonymous function is created – it increments a private variable **counter** and sends the result back into global scope as a function's return value.

Take away: Global Scope cannot directly *access* nor *modify* the **counter** variable at any time. Only the code inside the **closure** allows its inner function to modify the variable, still, without the variable leaking into Global Scope...

The whole point is that Global Scope does not need to know or understand how the code inside **plus()** works. It only cares about receiving the result of **plus()** operation so it can pass it to other functions, etc.

So why did we even bother explaining them? Beside that closures are one of the top-asked questions on JavaScript interviews?

Closures are similar to the idea of **encapsulation** – one of the key principles of Object Oriented Programming, where you hide the inner workings of a function or a method from the environment from which it was called.

This idea of making some variables private is key to understanding many other programming concepts.

If you think about it, this is exactly why **let** was added to JavaScript. It provides automatic privacy for variables defined in block-level scope. Variable privacy is a fundamental feature of many programming languages in general.

In Block-level Local Scope

The **let** and **const** keywords conceal variable visibility to scope in which they were defined and its inner scopes.

Scope visibility differences surface when you start defining variables inside local block-level scope or function-level scope.

In Classes

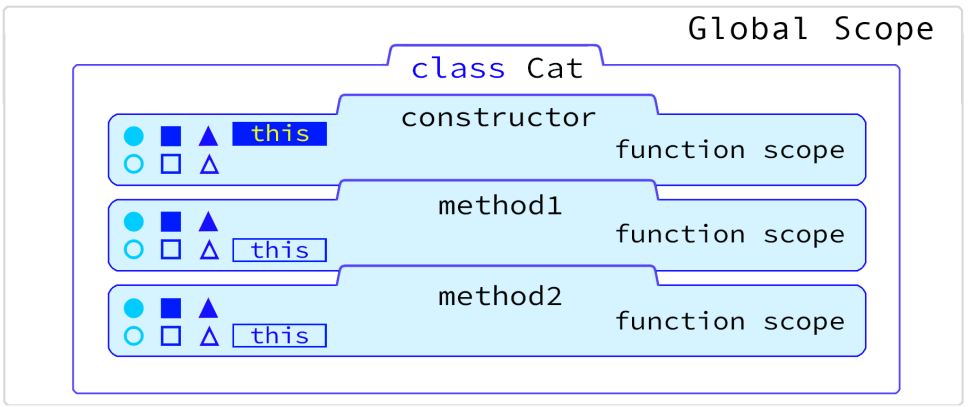
The **class** scope is simply a placeholder. Trying to define variables *directly* in class scope will produce an error:

```
001 class Cat {
002   let property = 1;  // "Unexpected token" Error
003   this.property = 2; // "Unexpected token" Error
004 }
```

Here are the proper places for defining local variables and properties. Note, in **class** methods, **let** (or **var** or **const**) only create a local variable to that scope. Therefore, it cannot be accessed outside of the method in which it was defined.

```
001 class Cat {
002   constructor() {
003     let property = 1;   // Ok: local variable
004     this.something = 2; // Ok: object property
005   }
006   method() {
007       console.log(this.property);  // undefined
008       console.log(this.something); // 1
009   }
010 }
```

In classes variables are defined inside its `constructor` function or its methods:



7.1.3 ES6 `const`

The **const** keyword is distinct from **let** and **var**.

It requires assignment on definition:

```
001 let a;
002 console.log(a); // undefined
003
004 const b; // Error: Missing initializer in const declaration
```

Figure 7.4: const *requires* initial value assignment.

This makes sense because value of **const** cannot be reassigned.

```
001 const speed_of_light = 186000; // Miles per second
002 speed_of_light = 1; // Error: Assignment to constant variable
```

It's still possible to change *values* of a more complex data structure such as `Array` or objects, even if variable was defined using `const`. Let's take a look!

7.1.4 `const` and Arrays

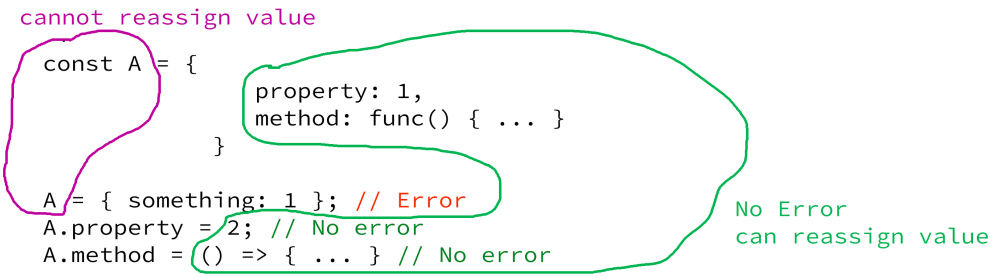
Changing a value in the `const` array is still allowed:

```
001 const A = [];
002 A[0] = "a"; // OK
003 A = []; // TypeError: Assignment to constant variable.
```

You just can't assign any new objects to the original variable name again.

7.1.5 `const` and Object Literals

Similar to arrays, when it comes to object literals, **const** only makes the definition constant. But it doesn't mean you can't change values of the properties assigned to a variable that was defined with const:



Conclusion

In case of a more complex data structure (object or array) you can think of **const** as something that does not allow you to reassign it to a new object again. The variable is locked to the original object, but you can still change the value of its properties (or indexes, in case of an array.)

If the value of a variable was defined with `const` and a single primitive (string, number, boolean,) such as speed of light, PI, etc, it cannot be changed.

7.1.6 Dos and Dont's

Do not use **var** unless for some reason you *want* to hoist the variable name. (These cases are rare and usually don't comply with good software design.)

Do use **let** and **const** instead of **var**, wherever possible. Variable hoisting (variables defined using **var**) can be the cause of unpredictable bugs, because only the variable name is hoisted, the value becomes `undefined`.

Do use **const** to define constants such as `PI`, `speed_of_light`, `tax_rate`, etc. – values that you *know* shouldn't change during the lifetime of your application.

Chapter 8

Operators

8.0.1 Arithmetic

operator	name	example	value
+	Addition	1 + 1;	2
-	Subtraction	5 - 3;	2
*	Multiplication	2 * 2;	4
/	Division	10 / 5;	2
%	Modulus	10 % 4;	2
++	Increment	a++;	a + 1
--	Decrement	a--;	a - 1

The arithmetic operators are pretty basic. They do exactly what you expect.

The modulus operator returns the number of times one number fits into the other. Here, 4 fits into 10 only 2 times – it is also often used to determine the remainder.

You can create statements without assigning the value to a variable name. It is possible to type them directly into your browser's developer console for practice:

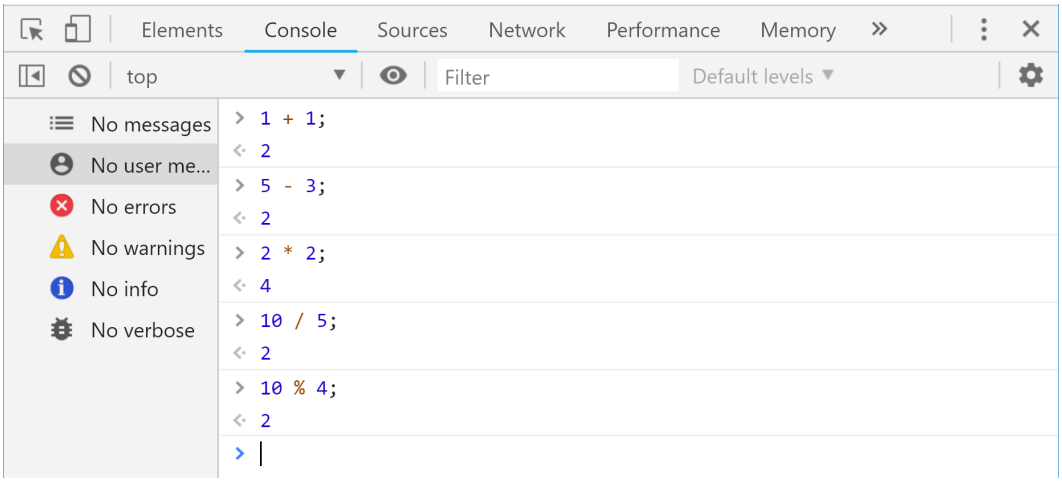


Figure 8.1: Typing JavaScript statements directly into Chrome console.

That works in Chrome console. But in your source code, evaluating simple statements is meaningless:

```
001 1 + 1;                              // 2
002 1 - 2;                              // -1
003 8 / 2;                              // 4
004 5 * 3;                              // 15
```

More often, you will perform operations directly on variable names:

```
006 // Define a variable
007 let variable = 1;                   // undefined
008
009 variable + 2;                       // 3
010 variable - 1;                       // 2
011 variable / 2;                       // 1
012 variable * 5;                       // 5
013 variable * variable;                // 25
014
015 variable++;                         // 26
016 variable++;                         // 27
017 variable++;                         // 28
018 variable--;                         // 29
```

8.0.2 Assignment

operator	name	example	result
=	Assignment	x = 1;	1
+=	Addition	x += 2;	3
-=	Subtraction	x -= 1;	2
*=	Multiplication	x *= 2;	4
/=	Division	x /= 2;	2
%=	Modulus	x %= 1;	0

Assignment operators assign a value to a variable. There are several assignment operators that can also combine assignment with one of the arithmetic operations.

8.0.3 String

Strings can be assigned to variable names or each other using the + operator which we earlier saw used as arithmetic addition. But when one or both of the values on either side of + operator are strings, it is treated as a string addition operator.

operator	name	example	result
=	Assignment	x = 'a'	'a'
+=	Concatenation	x += 'b'	'ab'
+	Addition	'x' + 'y'	'xy'

In this context the += operator can be thought of as string concatenation operator.

8.0.4 Comparison

operator	name	example	result
==	equality	1 == 1	true
		'1'== 1	true
		1 == 2	false
===	equality of value *and* type	'1' === '1'	true
		1 === 1	true
		1 === '1'	false
!=	inequality	1 != 1	false
		1 != 2	true
!==	inequality of value *and* type	'1'!== 1	true
		1 !== 1	false
>	greater than	2 > 1	true
<	less than	5 < 7	true
>=	greater or equal	2 >= 1	true
<=	less or equal	2 <= 1	false

Figure 8.2: Triple equality operator checks for **value** *and* **type**.

8.0.5 Logical

operator	name	example	result
&&	Logical AND	(5 < 1 && 3 > 2)	true
||	Logical OR	1 == 1 || 2 == 2	true
!	Logical NOT	!true	false
		!(1 == 2)	true

Logical operators are used to determine logic between the values of expressions or variables.

8.0.6 Bitwise

operator	name	example	result
&	Bitwise AND	a&b	1
|	Bitwise OR	a|b	13
^	Bitwise AND	a^b	12
~	Bitwise AND	~a	-6
<<	Bitwise AND	a<<1	10
>>	Bitwise AND	a>>2	2

In binary number system decimal numbers have an equivalent represented by a series of 0's and 1's. For example 5 is 0101 and 1 is 0001. Bitwise operators work on those bits, rather than number's decimal values.

We won't go into great detail about how they work, but you can easily look them up online. They have unique properties: for example: the << operator is the same as multiplying a whole number by 2 and >> operator is the same as dividing a whole number by 2. They are sometimes used as performance optimizations because they are faster than * and / operators in terms of processor cycles.

8.0.7 typeof

```
001 typeof 1;                          number (primitive)
002 typeof Number(1)                   number (primitive)
003 typeof new Number(1);              object (instance)
```

The **typeof** operator is used to check the type of a value. It will often evaluate to either primitive type, object or function. The value produced by the **typeof** operator is always string format:

operator	result
`typeof 125;`	`'number'`
`typeof 100n;`	`'bigint'`
`typeof 'text';`	`'string'`
`typeof NaN;`	`'number'`
`typeof true;`	`'boolean'`
`typeof [];`	`'object'`
`typeof {};`	`'object'`
`typeof Object;`	`'function'`
`typeof new Object();`	`'object'`
`typeof null;`	`'object'`

Figure 8.3: `NaN` (Not a Number) evaluates to 'number'. This is just one of many JavaScript quirks. However, they are not bugs and usually start to make more sense as your knowledge of JavaScript deepens.

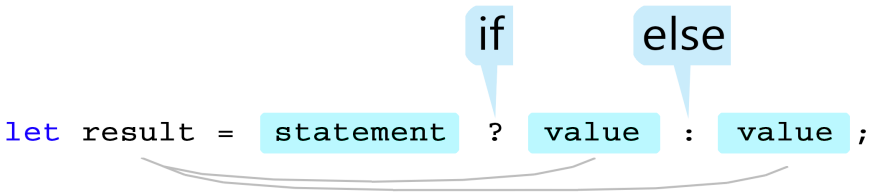
NaN lives natively on **Number.NaN** – it is considered to be a primitive value.

NaN is the symbol usually produced in the context of a numeric operation. One such example is trying to instantiate a number object by passing a string value to its constructor: **new Number("str")** in which case **NaN** would be returned.

8.0.8 Ternary (?:)

The ternary operator has the form of: `statement?statement:statement;`

Statements can be expressions or a single value:



The ternary operator is like an inline **if**-statement. It does not support {} brackets or multiple statements.

8.0.9 delete

The `delete` keyword can be used to delete an object property:

```
001 let bird = {
002   name: "raven",
003   speed: "30mpg"
004 };
005
006 console.log(bird); // {name: "raven", speed: "30mpg"}
007 delete bird.speed;
008 console.log(bird); // {name: "raven"}
```

You cannot use `delete` to remove stand-alone variables. Even though, if you try to do that, no error will be generated (unless you are in strict mode.)

8.0.10 in

The **in** operator can be used to check if a property *name* exists in an object:

```
001 "c" in { "a" : 1, "b" : 2, "c" : 3 }; // true
002   1 in { "a" : 1, "b" : 2, "c" : 3 }; // false
```

The **in** operator, when used together with arrays, will check if an index exists. Note, it is ignorant of actual *value* (in either arrays or objects.)

```
"c" in ["a", "b", "c"];     // false
  0 in ["a", "b", "c"];     // true
  1 in ["a", "b", "c"];     // true
  2 in ["a", "b", "c"];     // true
  3 in ["a", "b", "c"];     // false
```

You can check for properties on built-in data types. The `length` property is native to all arrays:

```
"length" in [];             // true
"length" in [0, 1, 2];      // true
```

The "length" property does not exist natively on an object unless it's added explicitly:

```
"length" in {};             // false
"length" in {"length": 1}; // true
```

Check for presence of constructor or prototype property on an object constructor function:

```
"constructor" in Object;    // true
  "prototype" in Object;    // true
```

Chapter 9

...rest and ...spread

9.0.1 Rest Properties

The `...rest` syntax can help you refer to a larger number of items by extracting them from a single function parameter name. The single `...rest` parameter is assumed to contain one or more arguments passed to the function:

(...🐖) => 🐖.map(🥓 => console.log(🥓));

Console output (if function was passed 3 arguments):

Figure 9.1: Using `...rest` properties to break down a larger number of arguments and passing them to higher-order function `Array.map()`

All joking aside, how can `...rest` parameters simplify our code?

```
001 let f = (...items) => items.map(item => console.log(item));
```

We can further shorten code by moving console output to a separate **print** function:

```
001 let print = item => console.log(item);
002
003 let f = (...items) => items.map(print);
```

Call the `f()` function with an arbitrary number of arguments:

```
001 f(1, 2, 3, 4, 5);
```

The function takes rest parameters: you can specify as many as you need.

Console output:

```
1
2
3
4
5
>
```

After arrow functions were introduced in EcmaScript 6, to further shorten code some started to name their variables using a single character:

```
001 let f = (...i) => i.map(v => console.log(v));
```

We just used multiple language features: an *arrow function*, `...rest`, and `.map()` to abstract our code to something that looks like a math equation without sacrificing original functionality. It definitely looks cleaner than a `for-loop`!

This might make your code even shorter, but it's probably harder to understand.

Remember, if you are working on a team, another person might be reading your code. Sometimes that person will be you in the future.

9.0.2 Spread Properties

You can think of spread as an opposite of rest.

It can help you extract parts from an object.

9.0.3 ...rest and ...spread

It's tempting to call **...rest** or **...spread** syntax *operators*. And you will often hear people refer to them as such. But in fact it is only syntax. An operator is often thought of as something that *modifies* a value. Rest and spread *assign* values.

If we box them into the category of operators, perhaps we could say they would be similar to the equality operator =. Rest and spread simply abstract it for working with multiple assignments.

The **rest** syntax is called *rest parameters*, in the context of using it as a parameter name in a function definition, where it simply means: "the rest of arguments". Sometimes it is referred to as rest *elements*, because it assumes multiple values.

Both **rest** and **spread** syntax takes on the form of `...name`. So what's the difference?

...spread operator – expand iterables into one or more arguments.
...rest parameter – collect all remaining parameters ("the rest of") into an array.

...rest

Let's say we have a simple `sum()` function:

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

This limits us to two arguments.

The **...rest** parameters can gather an unknown number of arguments passed to the function and store them in an array (named **args** in this example):

```
001 function sum(...args) {
002   console.log(args);
003 }
004 sum(1,2,3); // [1,2,3]
```

It is similar to the built-in arguments array-like object. But there is a difference. As the name suggests, rest can capture "the rest of" parameters.

Keep in mind, **...rest** must be either the only arguments token, or the last one on the list. It cannot be the first argument of many:

```
001 function sum(...args, b, c) {} // error
```

Figure 9.2: **...rest** parameters cannot appear as the leading parameter name when followed by more parameters. You can think of it as "the rest of" arguments when used in this context.

```
001 function sum(a, ...args, c) {} // error
```

Figure 9.3: Likeiwise **...rest** cannot appear in the middle of an argument list. In context of multiple function parameters, it is always the last one on the list.

If you don't follow this rule, the following error will be generated in console:

```
Uncaught SyntaxError: Rest parameter must be last formal parameter
> 
```

```
001 function sum(a, b, ...args) {} // correct
```

Figure 9.4: Correct placement of **...rest** parameters.

```
// Quirk: if you pass an array, it will register as an array:
sum([1,2,3]); // Array(3)

// But you can flatten an array using spread syntax!
sum(...[1,2,3]); // [1,2,3]
```

In this case the three dots in **...[1,2,3]** is actually **...spread**. You can see from this example that **...spread** is like a reverse of **...rest**: it *unpacks* values from an array (or an object, as we will see later.)

Contrary to **...rest**, **...spread** is allowed to be used anywhere on a list.

But **...rest** and **...spread** can sometimes overlap:

```
// here ...args is rest
function print(a, ...args) {
  console.log(a);
  console.log(args);
}
print(...[ 1, 2, 3 ], 4, 5); // here it is ...spread
// a = 1
// args = [2, 3, 4, 5]
```

Here, first **...spread** forms a complete list of arguments: `1, 2, 3, 4, 5` and that's what's passed into the **print** function.

Inside the function, **a** equals 1, and `[2, 3, 4, 5]` is "the rest of" arguments.

Here's another example:

```
// Spreading an array of 3 items into the first 3 parameters:
function print2(a, b, c, ...args) {
  console.log(a);
  console.log(b);
  console.log(c);
  console.log(args);
}
```

Let's try it out:

```
print2(...[ 1, 2, 3 ], 4, 5);
// a = 1
// b = 2
// c = 3
// args = [4, 5]
```

Creating a sum() function with ...rest arguments

Our first **sum** function using **...rest** parameters might look like this:

```
function sum(...arguments) {
  let sum = 0;
  for (let arg of arguments)
    sum += arg;
  return sum;
}
sum(1, 2, 5); // 8
```

But because **...args** produces an array, which makes it an iterable, we can use a reducer to perform the sum operation:

```
function sum(...args) {
  return args.reduce((x, v) => x + v, 0)
}
sum(1, 2, 3, 4, 5); // 15
```

But **...rest** parameters also work in *arrow functions* and we can further shorten the function to following form:

```
let sum = (...a) => a.reduce((x, v) => x + v, 0);
sum(100, 200, 400); // 700
```

Some people think while this format is shorter, it's harder to read. But that's the trade off you get with functional programming style. People with background in math find this format elegant. Traditional programmers might not.

Flattening arrays with ...spread

Luna would be a nice name for a female cat with silvery fur that resembles the moon's surface.

```
001 let names = ["felix", "luna"];
002 const cats = [...names, "daisy"];
003 console.log(cats); // (3) ["felix", "luna", "daisy"]
```

Using ...spread outside of arrays, objects or function parameters

You can't use **...spread** syntax to assign values to variables.

```
001 let a = ...[1,2,3]; // error
```

```
Uncaught SyntaxError: Unexpected token ...
> 
```

Are you disappointed? Don't be. You will love Destructuring Assignment.

9.1 Destructuring Assignment

Destructuring assignment can be used to extract multiple items from arrays and objects and assign them to variables:

```
001 // destructure from array
002 [a, b] = [10, 20];
003 console.log(a, b); // 10 20
```

The above code is the same as:

```
001 var a = 10;
002 var b = 20;
```

When **var**, **let** or **const** are not specified, **var** is assumed:

```
001 [a] = [1];
002 console.log(window.a); // 1
```

As expected **let** definitions are not available as a property on window object:

```
001 let [a] = [2];
002 console.log(window.a); // undefined
```

It is possible to destructure into **...rest** array:

```
001 // destructure from array with rest
002 [a, b, ...rest] = [30, 40, 50, 60, 70];
003 console.log(a, b); // 30 40
004 console.log(rest); // [50, 60, 70]
```

Destructuring is often used to extract object properties to a matching name:

```
001 // Destructure to oranges from oranges property in an object
002 let { oranges } = { oranges : 1 };
003 console.log(oranges);  // 1
```

The order doesn't matter – as long as there is property **grapes** the value will be assigned to the variable with the same name on the receiving end:

```
let fruit_basket = {
  apples : 0,
  grapes : 1,
  mangos : 3,
};

let { grapes } = fruit_basket;

console.log(grapes); // 1
```

Extract from multiple values. Grab apples and oranges and count them:

```
let { apples, oranges } = { apples : 1, oranges : 2 };
console.log(apples + oranges);  // 3
```

Destructuring is not implicitly recursive, second-level objects are not scanned:

```
let { oranges } = { apples : 1, inner: {oranges : 2} };
console.log(oranges);  // undefined
```

But it's possible to extract directly from object's inner properties:

```
let deep = {
    basket: {
        fruit: {
            name: "orange",
            shape: "round",
            weight: 0.2,
        }
    }
};

let { name, shape, weight } = deep.basket.fruit;

console.log(name);   // "orange"
console.log(shape);  // "round"
console.log(weight); // 0.2
```

If variable is not found in object, you will end up with **undefined**. For example, if we attempt to destructure to property name that doesn't exist in the object:

```
001 let { apples } = { oranges : 1 };
002 console.log(apples);  // undefined
```

It is possible to destructure and rename at the same time:

```
001 let { automobile: car } = { automobile: "Tesla" };
002 console.log(car); // "Tesla"
```

Merging objects with ...spread

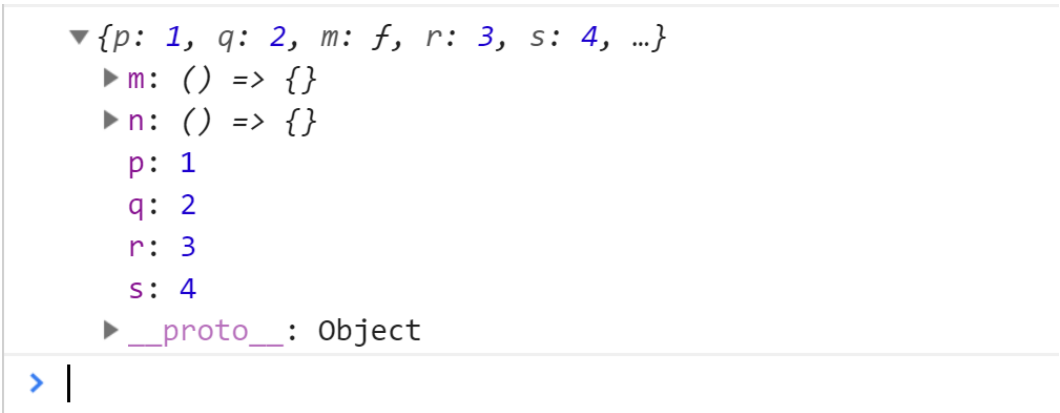
You can use **...spread** syntax to easily merge two or more objects:

```
001 let a = { p:1, q:2, m:()=>{} };
002 let b = { r:3, s:4, n:()=>{} };
003 let c = { ...a, ...b };
```

What are the contents of object **c**?

```
001 console.log(c);
```

Console output:

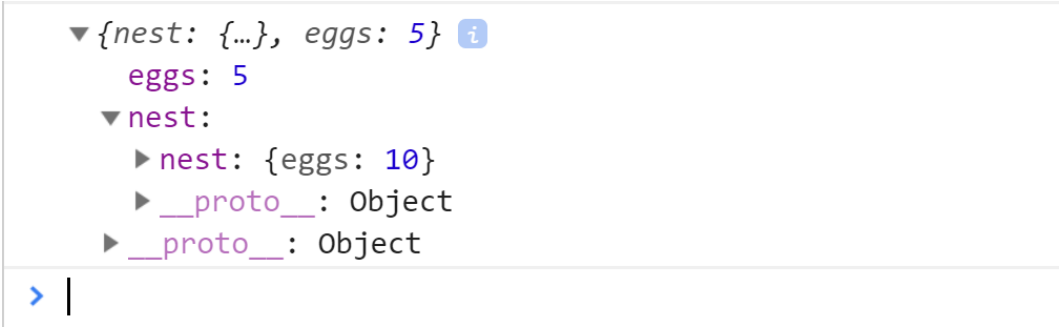


The great thing is that it's not just a shallow copy.

...spread copies nested properties too:

```
001 let a = { nest:{nest:{eggs:10}} };
002 let b = { eggs:5 };
003 let c = { ...a, ...b };
004 console.log(c);
```

Console output:



Merging arrays with ...spread

The same can be done with arrays:

```
001 let a = [1,2];
002 let b = [3,4];
003 let c = [...a, ...b];
004 console.log(c); // [1,2,3,4]
```

Chapter 10

Closure

Closure Introduction

There are many different ways to explain a closure. The explanations in this chapter should not be taken for the holy grail of closures. But it is my hope that interpretation presented in this book will be enough to deepen your understanding.

Feel free to play around with the examples shown here on `codepen.io` and see how they work. Eventually it should sink in.

In C, and many other languages, when a function call exits, memory allocated for that function is wiped out as part of automatic memory management on the stack. But in JavaScript, variables and functions defined inside that function still remain in memory, even after the function is called.

Retaining a link to variables or methods defined inside the function, after it has already been executed, is part of how a closure works.

JavaScript is an ever-evolving language.

When closures came around, there were no classes or private variables in JavaScript. It can be said that until EcmaScript 6, closures could be used to (roughly) simulate something similar to what is known as object's method privacy.

Closures are part of traditional programming style in JavaScript. They are a prime candidate for interview questions. Having said this, JavaScript Grammar cannot be complete without a discussion on closures.

What Is Closure?

A closure enables you to keep a *reference* to all local function variables, in the state they were found after the function exited.

Closures are difficult to understand, knowing nothing about scope rules and how execution context delegates control flow in JavaScript. But I think this task can be simplified if we start simple and take a look at a few practical examples.

To understand closures, we need to – at the very least – understand the following construct. Primarily it is enabled by the idea that in JavaScript you can define a function inside another function. Technically, that's what a closure is.

```
001 // The function global() is defined in an already existing
002 // execution context that was created together with window
003 function global() {
004     // At the time global() is invoked, a new execution
005     // context wil be created for this function scope.
006     // During declaration binding instantiation, internal
007     // to JavaScript interpreter, inner() will be created
008     // as a new native object with a scope that points to
009     // variable environment of global()'s execution context
010     function inner() {
011         console.log("inner");
012     }
013     inner();      // Call inner
014 }
015
016 global(); // "inner"
```

In the following example, global function `sendMail` defines an anonymous function and assigns it to variable `send`. This variable is visible only from the scope of `sendEmail` function, but not from global scope:

```
001 function sendEmail(from, sub, message) {
002   let msg = `"${sub}" > "${message}" received from ${from}.`;
003   let send  = function() { console.log(msg); }
004   send();
005 }
006
007 sendEmail('Jason', 'Re: subject', 'Good news.');
```

Figure 10.1: In JavaScript, inner functions have access to variables defined in the scope of the function in which they are defined.

When we call `sendEmail` it will create and call `send` function. It is not possible to call `send()` directly from global scope.

Console Output:

```
  "Re: subject" > "Good news." received from Jason.
> |
```

We can expose a reference to private methods (inner functions) by returning them from the function. The following example is exactly the same as one above, except here instead of calling the `send` method we *return* a refernce to it on line 004:

```
001 function sendEmail(from, sub, message) {
002   let msg = `"${sub}" > "${message}" received from ${from}.`;
003   let send = function() { console.log(msg); }
004   return send;
005 }
006
007 // Create a reference to sendMail
008 let ref = sendEmail('Jason', 'Re: subject', 'Good news.');
009
010 // Call by reference name
011 ref();
```

Figure 10.2: Instead of calling send(), let's return it. This way a reference to this private method can be created in global scope.

Now we can call `send()` by reference directly from global scope.

Even after `sendEmail` function was called, `msg` and `send` variables remained in memory. In languages like C, they would be removed from the automatic memory on the stack, and we wouldn't be able to access them. But not in JavaScript.

Let's take a look at another example. First we defined `print`, `set`, `increase` and `decrease` variables as global placeholders.

```
001 let print, set, increase, decrease;
002
003 function manager() {
004     console.log("manager();");
005     let number = 15;
006     print = function() { console.log(number) }
007     set = function(value) { number = value }
008     increase = function() { number++ }
009     decrease = function() { number-- }
010 }
```

In order to assign anonymous function names to global function variables, we need to run `manager()` at least once.

```
012 manager();   // initialize manager
013 print();     // 15
014 for (let i = 0; i < 200; i++) increase();
015 print();     // 215
016 decrease();
017 print();     // 214
018 set(755);
019 print();     // 755
020 let old_print = print; // save a reference to print()
021
022 manager();   // initialize manager (again)
023 print();     // 15
024 old_print(); // 755
```

Figure 10.3: The `set(755)` function resets the value of `number` to 755.

Console Output:

```
manager();
15
215
214
755
manager();
15
755
>
```

Explanation

After calling `manager()` for the first time. The function executed and all global references were linked to their respective anonymous functions. This created our first closure. Now, let's try to use the global methods to see what happens.

We then called some methods: `increase()`, `decrease()` and `set()` to modify the value of `number` variable defined inside manager function. At each step we printed out the value using the `print()` method, to confirm it actually changed.

Beautiful Closure

It can be assumed that closures are used in Functional Programming for similar reasons to why private methods are used in Object Oriented Programming. They provide a method API to an object in the form of a function.

What if we could advance this idea and create a closure that looked beautiful and returned several methods rather than just one?

```
001 let get = null; // placeholder for global getter function
002
003 function closure() {
004
005   this.inc = 0;
006   get = () => this.inc;  // getter
007
008   function increase() { this.inc++; }
009   function decrease() { this.inc--; }
010   function set(v) { this.inc = v; }
011   function reset() { this.inc = 0; }
012   function del() {
013     delete this.inc; // becomes undefined
014     this.inc = null; // additionally reset it to null
015     console.log("this.inc deleted");
016   }
017   function readd() {
018     // if null or undefined
019     if (!this.inc)
020         this.inc = "re-added";
021   }
022
023   // Return all methods at once
024   return [increase, decrease, set, reset, del, readd];
025 }
```

The **del** method will completely remove inc property from the object and **readd** will re-add the property back. For simplicity of the explanation there is no safeguarding against errors. But naturally, if the inc property was deleted, and an attempt to call any of the methods was detected, a reference error would be generated.

Initialize closure:

```
032 let f = closure();
```

Variable **f** now points to an array of exposed methods. We can bring them into global scope by assigning them to unique function names:

```
034 let inc = f[0];
035 let dec = f[1];
036 let set = f[2];
037 let res = f[3];
038 let del = f[4];
039 let add = f[5];
```

We can now call them to modify the hidden **inc** property:

```
041 inc(); // 1
042 inc(); // 2
043 inc(); // 3
044 dec(); // 2
045 get(); // 2
046 set(7);// 7
047 get(); // 7
048 res(0);// 0
049 get(); // 0
```

Finally we can delete the property itself using **del** method:

```
051 // Delete property
052 del(0);// null
053 get(); // null
```

Calling other functions at this point would produce a reference error, so let's re-add the **inc** property back to the object:

```
055 // Read property inc
056 add();
057 get(); // "re-added"
```

Reset the **inc** property to **0** and increment it by **1**:

```
059 res(); // 0
060 inc(); // 1
061 get(); // 1
```

Closing Words

Whenever a function is declared inside another function, a closure is created.

When a function containing another function is called, a new execution context is created, holding a fresh copy of all local variables. You can create a reference to them in global scope, by linking to variable names defined in global scope, or returning the closure from the outer function using `return` keyword.

A closure enables you to keep a *reference* to all local function variables, in the state they were found after the function exited.

Note: `new Function()` constructor does not create a closure, because objects created with `new` keyword also creates a stand-alone context.

10.0.1 Arity

Arity is the number of arguments a function takes.

You can access function's arity via Function.length property:

```
001 // Define a function with 3 parameters
002 function f(a,b,c) {}
003
004 // Get function arity
005 let arity = f.length;
006
007 console.log(arity); // 3;
```

10.0.2 Currying

In JavaScript functions are expressions. This also means a function can return another function. In the previous section we looked at the closure pattern. Currying is a pattern that immediately evaluates and returns another function expression.

A *curried function* can be constructed by chaining closures by defining and immediately returning all inner functions at the same time.

Here is an example of a curried function:

```
001 let planets = function(a) {
002   return function(b) {
003     return "Favorite planets are " + a + " and " + b;
004   };
005 };
006
007 let favoritePlanets = planets("Jupiter");
008
009 // Call the curried function with different arguments:
010 favoritePlanets("Earth");
011 favoritePlanets("Jupiter");
012 favoritePlanets("Saturn");
```

Function **planets** returns another anonymous function. So when it is assigned to **favoritePlanets** with one argument "Jupiter", it can be called again with a secondary argument.

Here is the result of the 3 curried functions from example above:

```
Favorite planets are Jupiter and Earth
Favorite planets are Jupiter and Mars
Favorite planets are Jupiter and Saturn
>
```

The inner function can be invoked immediately after the first call:

```
001 // Call the curried function with two arguments:
002 planets("Jupiter")("Mars");
```

And the result is:

```
Favorite planets are Jupiter and Mars
>
```

Currying is originally considered to be part of functional programming style.

It is not a surprise then, that this older currying syntax can be rewritten into this far more elegant arrow function format:

```
001 let planets = (a) => (b) => "Planets are " + a + " and " + b;
002
003 planets("Venus")("Mars");
```

And the outcome is:

```
Planets are Venus and Mars
>
```

Chapter 11

Loops

Loops are fundamental to working with lists. The primary purpose of a loop is to iterate over one or a set of multiple statements. Iterating is commonplace in software development – it means to *repeat* an action a multiple number of times.

Working with loops introduces the idea of **iterators**. Some built-in types are iterable. Iterables that can be passed to a **for...of** loops instead of using traditional **for-loop**. You can say that an **iterable** object abstracts away the index values of a list and helps you focus on solving the problem.

Array is an iterable. Object is not (objects are enumerable).

An iterable type guarantees the order of items in the set. This is why arrays have an index for each item. Enumerable types do not guarantee the order in which properties will appear when iterated.

11.0.1 Types of loops in JavaScript

There are different ways to iterate in JavaScript. Starting from classic **while** and **for-** loops to leaning more toward functional programming style iterators: using array's higher-order methods. Common iterators are **for**, **for...of**, **for...in**, **while** and **Array.forEach**. Some **Array** methods are assumed to be iterators: **.values**, **.keys**, **.map**, **.every**, **.some**, **.filter**, **.reduce** and a few others. They are called higher-order functions, because they take another function as an argument.

Incrementing And Reducing

Loops are often used for walking through a large list of objects and updating their properties. Loops can be used for filtering out objects and reducing the list to something more meaningful.

They can also be used for reducing a set of values to a single value:

```
let miles = [5, 12, 75, 2, 5];

// Add up all numbers using a for loop
let A = 0;
for (let i = 0; i < 5; i++)
  A += miles[i];
console.log(A); // 99
```

You can implement a reducer to the same effect:

```
// Add up all numbers using a reducer: Array.reduce method
const R = (accumulator, value) => accumulator + value;
const result = miles.reduce(R);
console.log(result); // 99
```

Generating HTML Elements Dynamically

Create a number of HTML elements dynamically to populate the UI view:

```
// Add 10 elements to the page
for (let i = 0; i < 10; i++) {
    // create a new HTML element
    let element = document.createElement("div");
    // insert inner HTML into the element
    element.innerHTML = "element " + i;
    // Add the created element to the document
    document.appendChild(element);
}
```

This code will add 10 **div** elements to the document.

It is possible to use **appendChild** method to create nested elements.

Render lists

Loops are often used together with **render lists**. Rendering is simply the act of displaying something on the screen. In software development, there are plenty of times when you need to *display* a list of items.

Dynamically sorted tables

Building an entire table dynamically can help you sort values by column using an **Array.entries** and **Array.sort** methods.

In some cases you will have to write your own sorting function, if your table columns are stored in an object as properties and not array items. That however, may or may not be a good idea, depending on the data set.

Note

You cannot easily make a decision about exactly how to deal with lists, until some sort of data layout is defined. So, choosing which type of loop to use will be often determined by other decisions made and the layout of your custom data structures.

11.1 for loops

For loop syntax comes in three syntactic flavors:

```
// For-loop with an empty body
for (initialize; condition; increment);

// Iterate over a single statement
for (initialize; condition; increment) single_statement;

// Iterate over multiple statements
for (initialize; condition; increment) {
    multiple;
    statements;
}
```

For loops require 3 statements separated by two semicolons, which can be any legal JavaScript statement, a function call, or even an empty statement.

You'll often use the following pattern in basic implementations: 1) **initialize counter** 2) **test condition** and 3) **increment** or **decrement** counter.

11.1.1 0-index based counter

Mark Duplass

Initializing the for-loop counter with a 0-index based value is a good idea, because most lists (like arrays) are 0-index based, where first item is located at `array[0]` and not `array[1]`. This might take some time to get used to.

11.1.2 The Infinite for Loop

A for loop can be defined without any of the default statements. But by doing this you will create an infinite for-loop that will freeze your program:

```
for(;;)
  console.log("hi"); // Infinite for loop - don't do it
```

You probably don't really want to do this, unless for some reason it becomes necessary. A **while** loop should probably be used in this case.

11.1.3 Multiple Statements

Multiple statements can be separated by comma. In the following example the **inc()** function is used to increment value of global variable counter. Note the combination of the two statements: **i++, inc()**:

```
001 let counter = 0;
002
003 function inc() { counter++; }
004
005 for (let i = 0; i < 10; i++, inc());
006
007 console.log(counter); // 10
```

This body-less for loop progresses the counter 10 times. The actual value is incremented inside the **inc()** function. This is just an example of executing multiple statements, we should definitely avoid using global variables in these types of cases.

Incrementing Numbers

At their basic, loops can be used to increment numbers.

```
001 let counter = 0;
002
003 for (let i = 0; i < 10; i++)
004     counter++;
005
006 counter; // 10;
```

for loops and let scope.

Bracket-less for-loop syntax is not good friends with the let keyword. The following code will generate an error:

```
for (var i = 0; i < 10; i++) let x = i;
```

A let variable cannot be defined without scope brackets. All variables defined using let keyword require their own local scope. This is fixed by:

```
for (var i = 0; i < 10; i++) { let x = i; }
```

Nested for Loops

Because a for loop is a JavaScript statement in itself it can be used as the iterable statement of another for loop. This hierarchical for loop is often used for working with 2-dimensional grids:

```
for (let y = 0; y < 2; y++)
    for (let x = 0; x < 2; x++)
        console.log(x, y);
```

Console output (all combinations between x / y):

```
0 0
1 0
0 1
1 1
```

Loop's Length

The condition statement will usually check if counter has reached a limit and if so, the loop will terminate:

```
"loop."
"loop."
"loop."
```

This simple loop will print "loop." to console 3 times:

```
"loop."
"loop."
"loop."
```

You can add brackets if you need to execute multiple statements:

```
for (let i = 0; i < 3; i++) {
    let loop = "loop.";
    console.log(loop);
};
```

The console output will be same as previous example.

Skipping Steps

You can skip an iteration step by using continue keyword;

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

Output of this for-loop: (note 1 was skipped)

```
0
2
```

The continue keyword tells code flow to go to the next step without executing any next statements in this for-loop's scope during the current iteration step.

Breaking Early

You can break out of a for loop by using break keyword:

```
for (let i = 0;; i++) {
    console.log("loop");
    break;
};
```

Note the condition statement was skipped here. The loop will break by an explicit command from the statement itself.

In this case the for loop will print "loop." to the console only once. The break keyword simply exits the loop whenever it's encountered. But it can also be conditional (see next example.)

Custom Breaking Condition

None of the 3 statements separated by ; ; in a for loop are required. It's perfectly legal to move the conditional test into the for loop's body, instead of testing for it between the parenthesis.

This example skips the middle statement, where we would usually create a conditional test for the counter, and replaces it by its own condition inside the loop where it breaks out of the loop if **i** is greater than **1**:

```
for (let i = 0;; i++) {
    console.log("loop, i = " + i);
    if (i > 1)
        break;
};
```

If not for the if statement inside the for loop, it would continue indefinitely because there are not other conditions stopping it from running.

Console output:

```
loop, i = 0
loop, i = 1
loop, i = 2
```

The word "loop." is printed 3 times. The condition **i** is greater than **1** might deceive us into thinking that the text will be printed 2 times at most. But it's

printed 3 times! That's because counting started with **0** and not **1**, and at its upper limit the condition evaluates to **2** and not **1**.

Breaking To A Label

In JavaScript, a statement can be labeled when a label_name: is prepended to a statement. Because a for loop is a statement you can label for loops.

Let's try to increment value of c inside the inner loop. By choosing whether to break the loop and jump to inner or mark label we change the pattern in which the for loops will work:

1. This example produces **11** when breaking to **mark**: label.

```
001 let c = 0;
002 mark: for (let i = 0; i < 5; i++)
003     inner: for (let j = 0; j < 5; j++) {
004         c++;
005         if (i == 2)
006             break mark;
007     }
008 console.log(c); // 11
```

2. This example produces **21** when breaking to **inner**: label.

```
001 let c = 0;
002 mark: for (let i = 0; i < 5; i++)
003     inner: for (let j = 0; j < 5; j++) {
004         c++;
005         if (i == 2)
006             break inner;
007     }
008 console.log(c); // 21
```

The two examples are logically different based on which label the execution flow of the inner for loop is transferred to.

Breaking from a labeled block scope

While we're on the subject, you can use break keyword to break out of regular non for-loop block scope as long as it's labeled:

```
001 block: {
002     console.log("before");
003     break block;
004     console.log("after");
005 }
```

Console output:

```
001 "before"
```

Execution flow never reaches "after".

11.2 for...of Loop

Using indexed iterators, such as the **for loop**, can become a hassle when dealing with arrays or object properties. Especially when their number is not known.

for...of loops to the rescue!

We'll start with a slightly advanced example where we will use a **for...of** loop together with a generator, and then discover some of the other, simpler use-cases.

11.2.1 for...of and Generators

Sometimes you might want to use a for loop with a generator – a special type of function with star * character appended to the `function*` keyword.

When a generator function is called, the multiple **yield** statements inside it do not execute at the same time, as you would normally expect. Only the first one does.

To execute **yield** statements 2 and 3, you have to call the generator function again (two more times). Internally, the **yield** statement counter is incremented automatically every time you call the generator function.

Generator executes a **yield** statement asynchronously, even though the code inside the generator function has linear appearance. This is done on purpose - it makes code more readable compared to alternatives (`XMLHttpRequest`, `Ajax`, etc).

```
001 // Generator function:
002 function* generator() {
003     yield 1;
004     yield 2;
005     yield 3;
006 }
007
008 for (let value of generator())
009     console.log(value);
```

The code above is equivalent to calling generator manually 3 times (*When you want to increment generator manually just make sure that you first assign it to another variable.*):

```
001 let gen = generator();
002
003 console.log(gen.next().value);
004 console.log(gen.next().value);
005 console.log(gen.next().value);
```

Here's the console output in either case:

```
001 1
002 2
003 3
```

Generators are one-time use functions. You should not attempt to reuse a generator function more than once as if it were a regular function (after the last **yield** statement has been executed.)

11.2.2 for...of and Strings

Strings are *iterable*.

You can walk each character of a string using a for...of loop:

```
001 let string = 'text';
002
003 for (let value of string)
004     console.log(value);
```

Console:

```
001 't'
002 'e'
003 'x'
004 't'
```

11.2.3 for...of and Arrays

Let's say we have an array:

```
001 let array = [0, 1, 2];
```

We can iterate through it without having to create index variables. Once the end of the array is reached the loop will end automatically:

```
001 for (let value of array)
002     console.log(value);
```

Console:

```
001 0
002 1
003 2
```

11.2.4 for...of and Objects

It would be nice to have the ability to iterate over an object's *properties* using **for...of** loop, right?

```
001 let object = { a: 1, b: 2, c: 3 };
002
003 for (let value of object) // Error: object is not iterable
004    console.log(value);
```

But **for...of** loops work only with *iterable* values. An object is *not* an iterable. (It has *enumerable* properties.) One solution is to convert the object to an iterable first before using it in a **for...of** loop.

11.2.5 for...of loops and objects converted to iterables

As a remedy you can first convert an object to an iterable using some of the built-in Object methods: **.keys**, **.values** or **.entries**:

```
let enumerable = { property : 1, method : () => {} };

for (let key of Object.keys( enumerable ))
    console.log(key);

Console Output:
> property
> method

for (let value of Object.values( enumerable ))
    console.log(value);

Console Output:
> 1
> () => {}

for (let entry of Object.entries( enumerable ))
    console.log(entry);

Console Output:
> (2) ["prop", 1]
> (2) ["meth", ƒ()]
```

This can also be achievable by using a **for...in** loop instead, without having to use any of the Object conversion methods.

We'll take a look at that in the following section.

11.3 for...in Loops

The **for...of** loops (in previous section) only accept *iterables*. Unless the object is first converted to an iterable, a **for...of** loop won't be of any help.

for...in loops work with *enumerable* object properties. It's a much more elegant solution for iterating through Object properties.

When you have an object at hand you should probably use a **for...in** loop.

```
001 let object = {
002     a: 1,
003     b: 2,
004     c: 3,
005     method: () => { }
006 };
007
008 for (let value in object)
009     console.log(value, object[value]);
010
011 Console Output: (Note that methods are enumerable too.)
012 1
013 2
014 3
015 () => { }
```

A **for...in** loop will iterate only *enumerable* object properties. Not all object properties are enumerable, even if they exist on the object. All non-enumerable properties will be skipped by **for...in** iterator.

You won't see **constructor** and **prototype** properties in an output from the **for...in** loop. Although they exist on an object they are not considered to be enumerable.

11.4 While Loops

A while loop will iterate for an indefinite number of times until the specified condition (there is only one) evaluates to false. At which point your loop will stop and execution flow will resume.

```
while (condition) {
    /* do something until condition is false */
}
```

Once condition evaluates to false the while loop will break automatically:

```
let c = 0;
while (c++ < 5)
    console.log(c);
```

Console output:

```
1
2
3
4
5
```

A secondary condition can be tested within the loop. This makes it possible to break from the loop earlier if needed:

```
while (condition_1) {
    if (condition_2)
        break;
}
```

11.4.1 While and continue

The continue keyword can be used to skip steps:

```
let c = 0;

while (c++ < 1000) {
    if (c > 1)
        continue;
    console.log(c);
}
```

Console output:

1

Just keep in mind this is just an example. In reality, this would be considered bad code, because the `if` statement is still executed 1000 times. The console prints 1 only when `c == 0`. If you need an early exist, use **break** instead.

Chapter 12

Arrays

Many of the `Array.*` methods are iterators. Instead of passing your array into a **for** or a **while** loop you should use built-in Array methods instead. Arrays usually already have methods offering cleaner syntax for anything you would write yourself to solve the same problem. So why re-invent the wheel?

Array methods are attached to Array.prototype property. This means you can execute them directly from array object like `array.forEach()` or directly from array's literal value like: `[1,2,3].forEach();`

12.0.1 ES10 Array.prototype.sort()

Previous (pre-ES10) implementation of V8 used an unstable quick sort algorithm for arrays containing more than 10 items.

A stable sorting algorithm is when two objects with equal keys appear in the same order in the sorted output as they appear in the unsorted input. But this is no longer the case. ES10 offers a stable array sort:

Fruit array definition:

```
var fruit = [
    { name: "Apple",      count: 13, },
    { name: "Pear",       count: 12, },
    { name: "Banana",     count: 12, },
    { name: "Strawberry", count: 11, },
    { name: "Cherry",     count: 11, },
    { name: "Blackberry", count: 10, },
    { name: "Pineapple",  count: 10, }
];
```

Perform sort:

```
// Create our own sort criteria function:
let my_sort = (a, b) => a.count - b.count;

// Perform stable ES10 sort:
let sorted = fruit.sort(my_sort);
console.log(sorted);
```

Console Output:

```
▼ (7) [{…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▶ 0: {name: "Blackberry", count: 10}
  ▶ 1: {name: "Pineapple", count: 10}
  ▶ 2: {name: "Strawberry", count: 11}
  ▶ 3: {name: "Cherry", count: 11}
  ▶ 4: {name: "Pear", count: 12}
  ▶ 5: {name: "Banana", count: 12}
  ▶ 6: {name: "Apple", count: 13}
    length: 7
  ▶ __proto__: Array(0)
> |
```

12.0.2 Array.forEach

The **forEach** method will execute a function for every item in the array.

Each iteration step receives 3 arguments **value**, **index**, **object**.

It's similar to a **for-loop** but it looks cleaner:

```
let fruit = ['pear',
             'banana',
             'orange',
             'apple',
             'pineapple'];

let print = function(item, index, object) {
    console.log(item);
}

fruit.forEach( print );
```

Starting from ES6 it can be suggested to use arrow functions together with Array methods. The code will be easier to read and maintain when building large scale applications. Let's take a look at how we can make syntax cleaner:

Because in JavaScript functions are also expressions, you can pass the function directly into the forEach method:

```
fruit.forEach(function(item, index, object) {
    console.log(item, index, object);
});
```

But you might want to use an arrow function here: `() => {}`

```
fruit.forEach((item, index, object) => {
    console.log(item, index, object);
});
```

The console output from both of the cases above:

```
"pear",      0, (5)["pear","banana","orange","apple","p...]
"banana",    1, (5)["pear","banana","orange","apple","p...]
"orange",    2, (5)["pear","banana","orange","apple","p...]
"apple",     3, (5)["pear","banana","orange","apple","p...]
"pineapple", 4, (5)["pear","banana","orange","apple","p...]
```

If you can get away with one argument and return statement you can use this slim form:

```
001 fruit.forEach(item => console.log(item));
```

As long as you have only one single statement you can remove brackets.

```
"pear"
"banana"
"orange"
"apple"
"pineapple"
```

12.0.3 Array.every

Return value: boolean

Not to be confused with "execute for every item" logic of forEach. In many cases method every will actually not run on every item in the array when at least one item doesn't evaluate to true based on specified condition.

The method every will return true if the value of every single item in the array satisfies the condition specified in its function argument:

The result is true because none of the numbers in the array are greater than or equal to 10. Let's take at the same function with a different value set. If 10 or a greater number was present in the array the result would be false:

Here one of the numbers is 256. Which can be translated to "not every value in the array is ¡ 10". Hence, false is returned. It's important to note that once Array.every method encounters 256 the condition function will not execute on the remaining items. Just a single failed test will cause false.

Array.every does not modify the original array. The value inside the function is a copy, not a reference to the value in the original array:

12.0.4 Array.some

Return value: boolean

Similar to **Array.every** except it stops looping whenever it encounters a value that evaluates to true (and not false like in **Array.every**) Let's compare:

```
let numbers = [0, 10, 2, 3, 4, 5, 6, 7];
let condition = value => value < 10; // value is less than 10
let some = numbers.some(condition); // true
let every = numbers.every(condition); // false
```

Here some returns true because it checks first value: 0 for ¡ 10 and immediately returns true without having to check the rest of the values.

The every method returns false on the same data set. That's because when it reaches the second item whose value is 10, the ¡ 10 test fails.

Note: Try not to think of some as an "opposite" of every. In some cases they return the same result on the same data set.

12.0.5 Array.filter

Return value: new array consisting only of items that passed a condition.

```
let numbers = [0,10,2,3,4,5,6,7];
let condition = value => value < 10;
let filtered = numbers.filter(condition);
console.log(filtered); // [0,2,3,4,5,6,7]
console.log(numbers); // Original array unchanged

```

The new filtered array contains all original items except 10. Because it did not pass

the ¡ 10 test. In a real-world scenario the condition can be much more complex and involve larger objects sets.

12.0.6 Array.map

Return value: a copy of the original array with modified values (if any.)

```
let numbers = [0,1,256,3,4,5,6,7];

let result = numbers.map(value => value = value + 1 );

console.log(result); // [1,2,257,4,5,6,7,8]
console.log(numbers); // Original array is unchanged
```

Array.map is like Array.forEach but it returns a copy of the modified array. Note that original array is still unchanged.

12.0.7 Array.reduce

Return value: accumulator

Reducers are similar to other methods. Yet they are unique because they have an accumulator value. The accumulator value must be initialized. There are different types of reducers. In this first example we'll take at a simple case.

As values are iterated this accumulator adds all numbers into a single value:

Like any other Array method that works with iterables, a reducer has access to the value it is currently iterating (currentValue).

This reducer added up all the numbers into the single accumulator value and returned it: $1 + 2 + 4 = 7$.

How to understand reducers in more complex, practical situations?

When developing software in the real world you won't be using reducers to count numbers. This can be done with a simple for loop. You will encounter plenty of situations where a set of data should be "reduced" only to the set of important values based on some criteria.

Array.reduce or Array.filter?

There is a reason why `Array` object has a number of different methods that at first sight appear to do the same things.

When it comes to Array methods, always try to to choose a proper tool for the task. Don't use reduce just because you want to use reduce – figure out if a particular method was *purposed* to produces a more efficient logic.

It is written, somewhere, that **reduce** is the father of **filter** and **map**. Anything you can do with filter and map can be done with reduce.

However, **reduce** provides a more *elegant* solution to adding up numbers than a for-loop or other Array methods.

Reducers And Updating Object Properties In A Database

After an API action – **update** or **delete** an item, for example – you may want to update the application view. But why update all objects everywhere, when you can "reduce" which object properties should be affected, without having to copy entire lists of object – even the ones that weren't affected by the API call?

12.0.8 Practical Reducer Ideas

Narrowing down on object properties

Let's say your car listing management application has a button that updates the price of a particular vehicle. The user sets a new price and clicks on the button. An action is dispatched to update the vehicle in the database.

Then the callback function returns containing the object with all properties for that vehicle ID (price, make, model, year). But, we only need to update the price.

A reducer can make sure to narrow down on (or "hand pick") only the vehicle price property, not the entire set of properties on the object. The object is then sent back to the database and application view is updated.

Counting weekends

Imagine a task where you had to implement a function that, given month would return number of weekends, working days and holidays there are in that month. A month could be represented by the number of days in it.

It's important to keep the input values the same type as the return value of a reducer. This is one of the main characteristics of a reducer: to reduce a set (not necessarily by filtering, although it is a possibility.)

Function Purity

Reducers are often used in code written following Functional Programming style principles. One of which is **function purity**. The following Dos and Dont's describe some of properties of a pure function.

12.0.9 Dos and Dont's

Even though these are not absolute requirements, these ideas might be helpful for avoiding anti-pattern code. Only use `reduce()` when the result has the same type as the items and the reducer is associative:

```
001 [1,2,3,4,5].reduce((a, b) => a + b, 0);
```

Figure 12.1: Array of *numbers* is "reduced" to a *number* - value of the same type.

Do use it for summing up some numbers.

Do use it for multiplying some numbers.

Do use it for **updating** state in React.

Do not use it for **building** new lists or objects from scratch.

Do not use it for just about anything else (use a loop).

Do not use it to mutate (change original values of) its arguments.

Do not use it perform side effects, like API calls and routing transitions.

Do not use it to call non-pure functions, e.g. Date.now() or Math.random().

12.0.10 ES10 Array.flat()

Flattening a multi-dimensional array:

```
let multi = [1,2,3,[4,5,6,[7,8,9,[10,11,12]]]];
multi.flat();               // [1,2,3,4,5,6,Array(4)]
multi.flat().flat();        // [1,2,3,4,5,6,7,8,9,Array(3)]
multi.flat().flat().flat();// [1,2,3,4,5,6,7,8,9,10,11,12]
multi.flat(Infinity);       // [1,2,3,4,5,6,7,8,9,10,11,12]
```

12.0.11 ES10 Array.flatMap()

```
let array = [1, 2, 3, 4, 5];
array.map(x => [x, x * 2]);
```

Becomes:

```
▶ 0: (2) [1, 2]
▶ 1: (2) [2, 4]
▶ 2: (2) [3, 6]
▶ 3: (2) [4, 8]
▶ 4: (2) [5, 10]
  length: 5
```

Now flatten the map:

```
array.flatMap(v => [v, v * 2]);
```

Becomes:

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

12.0.12 ES10 String.prototype.matchAll()

Matching multiple patterns in a string is a common problem when writing software. Use cases include extracting name and email addresses at the same time from an email header, scanning for presence of unique patterns, etc.

In the past, to match multiple items we used `String.match` with a regular expression and `/g` ("global") flag or `RegExp.exec` and/or `RegExp.test` with `/g`.

First, let's take a look at how the older spec worked.

String.match with string argument only returns the first match:

```
001 let string = "Hello";
002 let matches = string.match("l");
003 console.log(matches[0]); // "l"
```

The result is a single `"l"` (**note:** the match is stored in `matches[0]`, not `matches`.)

Only `"l"` is returned from a search for `"l"` in the word `"hello"`.

The same goes for using string.match with a regex argument. Let's locate the `"l"` character in the string `"hello"` using the regular expression `/l/`:

```
001 let string = "Hello";
002 let matches = string.match(/l/);
003 console.log(matches[0]); // "l"
```

Adding /g to the mix

`String.match` with a regex and the `/g` flag *does* return multiple matches:

```
001 let string = "Hello";
002 let ret = string.match(/l/g); // (2) ["l", "l"];
```

Great... we've got our multiple matches using < `ES10`. It worked all along. So why bother with a completely new `matchAll` method? Well, before we can answer this question in more detail, let's take a look at capture groups. If nothing else, you might learn something new about *regular expressions*.

Regular Expression Capture Groups

Capturing groups in regex is simply extracting a pattern from `()` parenthesis.

You can capture groups with `/regex/.exec(string)` and with `string.match`.

Regular capture group is created by wrapping a pattern in `(pattern)`.

But to create groups property on resulting object it is: `(?<name>pattern)`.

To create a group name: prepend ?<**name**> *inside* brackets. The resulting object will have a new property `groups.name` attached (see code below.)

This is the string we will take as the specimen for our experiment:

```
black*raven lime*parrot white*seagull
```

Figure 12.2: String specimen to match.

Here's a code example:

```
001 const string = 'black*raven lime*parrot white*seagull';
002 const regex = /(?<color>.*?)\*(?<bird>[a-z0-9]+)/g;
003 while (match = regex.exec(string))
004 {
005     let value = match[0];
006     let index = match.index;
007     let input = match.input;
008     console.log(`${value} at ${index} with '${input}'`);
009     console.log(match.groups.color);
010     console.log(match.groups.bird);
011 }
```

Figure 12.3: Note: `match.groups.color` & `match.groups.bird` are created from adding ?<**color**> and ?<**bird**> to the `()` match in the regex string.

regex.exec method needs to be called multiple times to walk the entire set of the search results. During each iteration when **.exec** is called, the next result is revealed (exec *doesn't return all matches right away.*) Hence, while loop.

Console Output:

```
black*raven at 0 with 'black*raven lime*parrot white*seagull'
black
raven
lime*parrot at 11 with 'black*raven lime*parrot white*seagull'
lime
parrot
white*seagull at 23 with 'black*raven lime*parrot white*seagull'
white
seagull
> 
```

But there is the quirk:

If you remove /g from this regex, you will create an infinite loop cycling on the first result forever. This has been a huge pain in the past. Imagine receiving a regex from some database where you are unsure of whether it has /g at the end or not. You'd have to check for it first, which would require additional code.

And now, we have enough background to answer the question:

Good reasons to use `.matchAll()`

1. It can be more elegant when using with capture groups. A capture group is simply the part of regular expression with `( )` that extracts a pattern.

2. It returns an iterator instead of array. Iterators on their own are useful.

3. An iterator can be converted to an array using spread operator `(...)`

4. It avoids regular expressions with /g flag... useful when an unknown regular expression is retrieved from database or outside source and used together with the archaic RegEx object.

5. Regular expressions created using `RegEx` object cannot be chained using the dot `(.)` operator.

6. Advanced: `RegEx` object changes internal `.lastIndex` property that tracks

last matching position. This can wreck havoc in complex cases.

How does `.matchAll()` work?

Let's try to match all instances of letter e and l in the word hello. Because an iterator is returned we can walk it with a `for...of` loop:

```
001 // Match all occurrences of the letters: "e" or "l"
002 let iterator = "hello".matchAll(/[el]/);
003 for (const match of iterator)
004     console.log(match);
```

Note that `.matchAll` method does not require /g flag.

```
001 [ 'e', index: 1, input: 'hello' ] // Iteration 1
002 [ 'l', index: 2, input: 'hello' ] // Iteration 2
003 [ 'l', index: 3, input: 'hello' ] // Iteration 3
```

Capture Groups example with `.matchAll()`

`.matchAll` returns an *iterator* so we can walk it with `for...of` loop.

```
001 const string = 'black*raven lime*parrot white*seagull';
002 const regex = /(?<color>.*?)\*(?<bird>[a-z0-9]+)/;
003 for (const match of string.matchAll(regex)) {
004     let value = match[0];
005     let index = match.index;
006     let input = match.input;
007     console.log(`${value} at ${index} with '${input}'`);
008     console.log(match.groups.color);
009     console.log(match.groups.bird);
010 }
```

Console Output:

```
black*raven at 0 with 'black*raven lime*parrot white*seagull'
black
raven
lime*parrot at 11 with 'black*raven lime*parrot white*seagull'
lime
parrot
white*seagull at 23 with 'black*raven lime*parrot white*seagull'
white
seagull
> 
```

Perhaps aesthetically it is very similar to the original `regex.exec` while loop implementation. But as stated earlier this is the better way for the reasons mentioned above. And removing /g won't cause an infinite loop.

12.0.13 Dos and Dont's

Do use `string.matchAll` instead of `regex.exec` & `string.match` with /g flag.

12.0.14 Comparing Two Objects

It makes sense to compare two literal numeric values such as `1 === 1` or literal boolean values `true === false`, but what does it mean to compare two objects? Furthermore == and === operators won't help because they compare by *reference* and not by value: `[]` and `[]` may be equal by value but they are different arrays:

```
[] === [];            // false by value
let x = []; x === x; // true only by reference
```

We'll have to write our own function! One way of thinking about comparing objects is comparing the number, type and value of all of their properties.

Following naming convention of **strcmp**: a function that compares strings in many languages, we can write a custom function **objcmp** that compares two objects:

```
// Compare objects & return true if all properties are equal
export default function objcmp(a, b) {

    // Copy properties into A and B
    let A = Object.getOwnPropertyNames(a);
    let B = Object.getOwnPropertyNames(b);

    // Return early if number of properties is not equal
    if (A.length != B.length)
        return false;

    // Walk and compare all properties on both objects
    for (let i = 0; i < aProps.length; i++) {
        let propName = A[i];

        // properties must equal by value *and type
        if (a[propName] !== b[propName])
            return false;
    }

    // objects are equal
    return true;
}
```

Figure 12.4: A shallow copy object property comparison algorithm.

The function **objcmp** takes two arguments: **a** and **b**, representing the two objects we want to compare.

This is a non-recursive, shallow copy algorithm. In other words, we are only comparing first-class properties attached directly to the object. Properties attached to properties are not compared. In many cases this is enough.

But there is another, much greater problem. In its current form the function assumes that properties cannot point to either an array or object. Two common data structures that are very likely to be part of an arbitrary object.

Even if we're not doing a deep comparison, the function will fail if any of the properties point to either an object or an array, regardless of whether their length and actual values are the same:

```
001 // Create two identical objects
002 let a = { prop: [1,2], obj: {} };
003 let b = { prop: [1,2], obj: {} };
004
005 objcmp(a, b); // false
```

Our algorithm failed on `[1,2] === [1,2]` comparison. So how do we deal with this situation? First, we can write our own **is_array** function. Because array is the only object in JavaScript with `length` property and at least 3 higher-order functions: **filter**, **reduce** and **map**, we can say that if these methods exist on an object, then it must be an array, with roughly 99% certainty:

```
001 function is_array(value) {
002     return typeof value.reduce == "function" &&
003            typeof value.filter == "function" &&
004            typeof value.map == "function" &&
005            typeof value.length == "number";
006 }
007
008 // Test the function
009 console.log(is_array(1));            // false
010 console.log(is_array("string"));     // false
011 console.log(is_array({a: 1}));       // false
012 console.log(is_array(true));         // false
013 console.log(is_array([]));           // true
014 console.log(is_array([1,2,3,4,5])); // true
```

12.0.15 Writing `arrcmp`

Let's write our own **arrcmp** function based on the assumption the array equality means that each value at each corresponding index in both arrays match:

```
001 let a = [1,2];
002 let b = [1,2];
003 let c = [5,5];
004
005 function arrcmp(a, b) {
006
007   // One or more values are not arrays:
008   if (!(is_array(a) && is_array(b)))
009       return false;
010
011   // Not equal by length
012   if (a.length != b.length)
013       return false;
014
015   // Compare by value
016   for (let i = 0; i < a.length; i++)
017       if (a[i] !== b[i])
018           return false;
019
020   // All tests passed: arrays a and b are equal
021   return true;
022 }
023
024 console.log(arrcmp(a,b)); // true
025 console.log(arrcmp(b,b)); // true
026 console.log(arrcmp(b,c)); // false
```

There is no built in function for comparing arrays in JavaScript. There is probably a good reason for it. How data is mapped to an array largely depends on overall design of the data in your application.

After all, what exactly does it mean for two arrays to be equal? The data layout can be different from project to project. So is the nature of what you're trying to accomplish, by storing data in arrays. For this reason an array does not always guarantee integrity between its values and indexes they are stored at.

Without a concrete project, we can only assume that it does.

12.0.16 Improving `objcmp`

Now that we have **is_array** and **arr_cmp** let's add two special case comparisons to the **objcmp** function: one for arrays using our new function, and one for objects using recursion. This deepens our algorithm and makes it less prone to bugs.

We will call **objcmp** from itself (`line 025`) if one of the object properties checks out to be an object literal itself:

```
001 // Compare objects & return true if all properties are equal
002 function objcmp(a, b) {
003
004     // Copy properties into A and B
005     let A = Object.getOwnPropertyNames(a);
006     let B = Object.getOwnPropertyNames(b);
007
008     // Return early if number of properties is not equal
009     if (A.length != B.length)
010         return false;
011
012     // Walk and compare all properties on both object
013     for (let i = 0; i < A.length; i++) {
014         let propName = A[i];
015         let p1 = a[propName];
016         let p2 = b[propName];
017         // Property points to an array
018         if (is_array(p1) && is_array(p2)) {
019           if (!arrcmp(p1, p2))
020               return false;
021         } else
022         // Property points to an object
023         if (p1.constructor === Object &&
024             p2.constructor === Object) {
025             if (!objcmp(p1, p2))
026                 return false;
027         // Compare by primitive value
028         } else if (p1 !== p2)
029             return false;
030     }
031     return true;
032 }
```

Lines that were changed from previous version are highlighted. A test for whether

property points to an array or an object literal was added. If the property is neither, primitive value is tested as usual.

12.0.17 Testing `objcmp` on a more complex object

Let's try it out in action!

```
let M = {
    a: 1,
    b: 100n,
    c: {},
    d: [1,2],
    e: "abc",
    f: true,
    g: () => {}
};
let N = {
    a: 1,
    b: 100n,
    c: {},
    d: [1,2],
    e: "abc",
    f: true,
    g: () => {}
};
let O = {
    a: 1,
    b: 100n,
    c: {},
    d: [5,7],
    e: "abc",
    f: true,
    g: () => {}
};
```

Here we have 3 objects **M**, **N** and **O**.

Objects **M** and **N** are the same, whereas **O** only differs in array value `[5,7]`.

```
objcmp(M, M);                   // true
objcmp(M, N);                   // true
objcmp(M, O);                   // false
```

Our first attempt to run the older version of **objcmp** failed on the following case.

But now it works:

```
// Create two identical objects
let a = { prop: [1,2], obj: {} };
let b = { prop: [1,2], obj: {} };

objcmp(a, b); // true
```

Another test, this time using a few different object literals:

```
objcmp({a:[1,2]}, {a:[1,2]}); // true
objcmp({a:{b:1}}, {a:{b:1}}); // true
objcmp({a:[1,2]}, {a:[1,3]}); // false
objcmp({a:{b:1}}, {a:{b:2}}); // false
```

Our object comparison function now works as expected. It's not perfect but at least it won't get stuck in large majority of cases. In fact, it even checks object properties recursively, which is a bit more of a deeper search than before.

Possible Improvements: You can further improve this function by checking for arrays of objects, instead of just arrays of values.

As you can see, this is exactly why a native function for a deep search doesn't exist. It really depends on your data implementation. Who says your arrays will contain objects or that any object property will ever point to another object? Without this knowledge, it's difficult to create a common-case algorithm without risking creating an anti-pattern (*when your code does more than it needs to.*)

Chapter Review

Trying to solve one problem (compare two objects) led to discovery of another problem we also needed to solve first (compare two arrays.) This is not something we predicted at the time we thought of writing **objcmp**.

JavaScript already provides **Array.isArray** method – so why reinvent the wheel?

Taking initiative to solve *any* problem is what makes the distinction between a hobbyist coder and a software developer. The skill of thinking for yourself to solve problems, instead of using existing libraries, helps you train yourself for.

Writing your own code is always a good idea. If you can't write an **is_array** function yourself, chances are you won't have the practice and training to write an important function later, one that doesn't exist but is crucial to success of a real-life project, facing much more complex problems.

This might cause you not only reputation among your peers, but the job itself. Most interviews at software companies don't test only for knowledge, they want to see your problem-solving approach. Perhaps, it's a good idea to develop it!

Chapter 13

Functions

13.1 Functions

In JavaScript there are two types of functions: the standard function that can be defined using **function** keyword and an arrow function `()=>{}` added later in ES6.

Regular **functions** can be called. But they can also act as object constructors, when used together with **new** operator in order to create an *instance* of an object. Note that since EcmaScript >= 6 you can use **class** keyword to the same effect.

Inside a function, the **this** keyword can point either to the context from which the function was called. But it can also point to an instance of the created object, if that function was used as an object constructor.

Functions have an array-like **arguments** object inside their scope, which holds the length of parameters and values that were passed to the function, even if parameter names were not present in function definition.

An **arrow function** can be called. But it cannot be used to instantiate objects.

Arrow functions are often used with functional programming style. Like regular functions, they can be used to define object methods. They are also often used as event callback functions. Inside the scope of an arrow function **this** keyword points to whatever **this** equals to outside of its own scope.

Arrow functions do not have the array-like **arguments** inside their scope.

13.1.1 Function Anatomy

The function definition consists of the **function** keyword followed by its name (shown as **update**, in the following example) parenthesis containing a list of parameter names (**a,b,c**) and the function body enclosed in brackets:

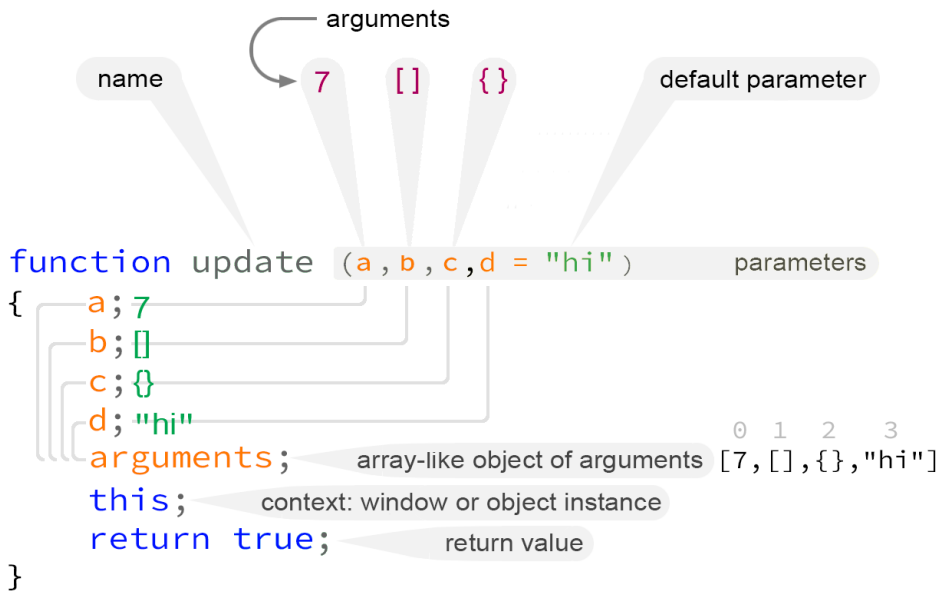


Figure 13.1: Function anatomy.

The **return** keyword is optional. But function will return anyway once all statements in its body are done executing, even if **return** keyword is not specified.

The **this** keyword inside an ES5 style function points to the context from which the function was executed. Very often it is the global `window` object. If the function is used to instantiate an object using **new** keyword then **this** keyword will point to object instance that was instantiated using function.

The **arguments** is an array-like object that contains 0-index list of arguments that were passed into the function, even if parameter names were not specified in the function's definition.

13.1.2 Anonymous Functions

Nameless or *anonymous* functions can be defined by using the same syntax but skipping the function name.

Anonymous functions are often used as event callbacks, where we usually don't need to know what the names are – we simply want to execute them at a specific time after an event has finished doing its work:

```
setTimeout(function() {
  console.log("Print something in 1 second.");
  console.log(arguments);
}, 1000);
```

Figure 13.2: Anonymous function used as a **setTimeout** event callback.

```
document.addEventListener("click", function() {
  console.log("Document was clicked.");
  console.log(arguments);
});
```

Figure 13.3: Anonymous function used to intercept a mouse click event.

13.1.3 Assigning Functions To Variables

Anonymous functions can be assigned to a variable, making them named functions again. By doing this you can separate the function definition from its use in an event-based method:

```
let print = function() {
  console.log("Print something in 1 second.");
  console.log(arguments);
}
```

Figure 13.4: Assign an anonymous function to variable **print**.

```
let clicked = function() {
  console.log("Document was clicked.");
  console.log(arguments);
}
```

Figure 13.5: Assign an anonymous function to variable **clicked**.

We can now call these functions by their name:

```
// call anonymous named functions
print();
clicked();
```

Figure 13.6: "anonymous" functions that were assigned to a variable name become named functions.

You can also pass them to the event functions just by their name:

```
// cleaner code
setTimeout(print, 1000);
document.addEventListener("click", clicked);
```

Figure 13.7: Cleaner code.

This often makes your code look cleaner. Note that different event functions generate their own arguments, regardless of whether your anonymous function defines parameters to catch them – they *will* be passed into the function:

```
let clicked = function(event) {
  console.log(event, event.target);
}
```

Function Parameters

Function parameters are optional. But in many cases you will find yourself defining some. You can use default function parameters to pass things like primitives, arrays and objects into a function. Anything that evaluates to a single value will do it:

```
001 function Fun(text, number, array, object, func, misc) {
002
003     // Output the arguments values:
004     console.log(text);
005     console.log(number);
006     console.log(array);
007     console.log(object);
008     console.log(func);
009     console.log(misc);
010
011     // Call the function by its parameter name
012     func();
013 }
```

You can pass the name of another function. This way you can call that function at some point later inside another function.

Let's pass some arguments into function's parameters:

```
001 function Volleyball() { return "Volleyball"; }
002
003 Fun("Text", 125, [1,2,3], {count:1}, Volleyball, Volleyball());
```

Note that we're passing the function name `Volleyball` and the result: `Volleyball()` (*which will evaluate to string value "Volleyball."*) into the last two parameters of the function `Fun();` to get the following console output:

```
Text                                          console.log(text);
125                                           console.log(number);
▶ (3) [1, 2, 3]                               console.log(array);
▶ {count: 1}                                  console.log(object);
ƒ Volleyball() {return "Volleyball";}         console.log(func);
Volleyball                                    console.log(misc);
Volleyball                                    console.log( func() );
>
```

That's a useful variety of primitives and objects!

The last value is generated by calling function `Volleyball`, which was passed into parameter `func`. This means we can simply call it by calling `func()` instead of `Volleyball()`, since `func` is its name inside the function.

Checking For Types

JavaScript is a dynamically-typed language. The type of a variable is determined by its value. The variable definition simply assumes the type. This sometimes can be the cause of subtle bugs.

For example, even though JavaScript provides a number of different object types to work with, it doesn't really provide an *automatic* safeguard that *makes sure* the arguments passed into the function were what you expect them to be.

What if in the previous example, we passed an array or an object into `func` parameter? The function expects it to be a function. We wouldn't be able to call `func()` if `func` wasn't a function.

Let's narrow down on the problem:

```
001 function Fun(func) {
002   console.log(func()); // Call the function by its parameter name
003 }
004
005 var array = [];
006 var f = function() {}
007
008 Fun(array); // pass array instead of function
```

But we cannot call an array. Hence, console output:

```
Uncaught TypeError: func is not a function
    at Fun (pen.js:4)
    at pen.js:9
>
```

This is a problem if it happens in production code.

Safeguarding Function Parameters

The solution is to safeguard the value by checking its type. JavaScript has a built-in directive `typeof` that we can use before calling the function:

```
001 function Fun(func) {
002   // Call the function but only if it *is a function:
003   if (typeof func == "function")
004       console.log(func());
005 }
006
007 var array = [];
008 var f = function() {}
009
010 Fun(array); // pass array instead of function
```

The function `Fun(array)` was called. The function expects a function name as an argument but an array was sent. The **typeof** test failed and nothing happened but at least our program didn't break.

You can do the same with other types if it becomes imperative that a particular value must be in absolute compliance with a particular type.

13.2 Origin of `this` keyword

The **this** keyword was borrowed from C++. In its original design `this` keyword was meant to point to an *instance* of an object in `class` definitions.

That's it! There shouldn't be much more to it.

But it seems like original designer of JavaScript language decided to use **this** keyword to provide one extra feature: carrying a link to *execution context*, which shouldn't really be part of computer language feature design, but rather kept away to its internal implementation:

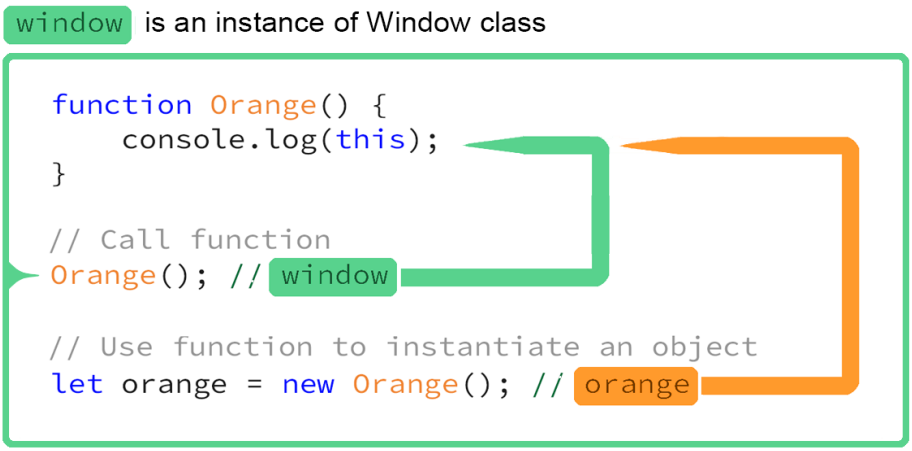


Figure 13.8: This duality of **this** keyword often causes a headache that you may need to take two Ibuprofens for.

You can't avoid dealing with context in any programming language. But wiring context into **this** keyword was a mistake and created duality and much confusion.

Later arrow functions were invented to deal with some of the side effects of this odd-ball use-case for **this** keyword. And that is our next subject!

Chapter 14

Higher-order Functions

14.0.1 Theory

Higher-order functions may sound complicated but they are actually simpler than their regular (first-order) functions that you've already been dealing with all along.

Higher-order functions, as the name suggests, is something that exists at a higher level of thinking. That's what *abstraction* is. If you want to become even better at software development, abstraction is a very important concept to understand.

You can think of abstraction as the quality of dealing with ideas rather than gritty details. Once you get a good grasp on it, abstraction will become your best friend.

Abstraction

When you are driving a car and you push the break pedal, you don't think about weight distribution, the brake caliper containing pistons that push the brake pads against the disc, or how power assistance mode augments the pressure you are placing on the pedal. You simply want the car to stop. That's what you expect to happen. You see? You've already been using abstract thinking – it's natural.

How does this hold up in the context of writing software applications? In the following section, we will write our own higher-order function **map** that will walk through each item in an array, and apply a function to it. If our **map** function was the car brake system from previous example, it would be the brake pedal. It will be responsible for controlling the brake calipers and pistons – a mechanism designed to take care of the low-level details by making them abstract. So when we call

map function, we don't have to think of all the gritty details. We simply want something to happen and have the expected result returned from the function.

That's great, but before using the function we actually have to design its content and determine how we want those gritty details to work.

JavaScript already supports several higher-order functions that do just that. But before using them, we will write our own. If anything, this will help us deepen our understanding not only of high-order functions but of abstraction in general.

Writing your first higher-order function

Not all problems can be solved with built-in JavaScript methods. Working on custom software, you will be faced with situations where you would have to write your own functions of this kind, or functions that require thinking in abstract terms in order to produce the most efficient solution.

The first-order function that applies an action

The actual action happens not in **map**, but in the *function passed to* **map** as one of its parameters. This means the **map** function doesn't have one single purpose. It is whatever the function passed into it does. And that can be anything.

We'll create the **map** function and pass a first-order function into it, whose single purpose will be to increment a numeric value by 1. Just like in the car brakes example, the user of our **map** function should not be concerned about *how* the internal `for-loop` iterates through all items. We just want to give it a task. In order to do that, we will pass *another* function into the higher-order **map** function.

Important: What actually makes the function abstract is the fact that the higher-order function itself does not need to know *exactly* what it's doing. It is simply a logical scaffold to perform an action on a set of values. Much like a for-loop. In fact, a for-loop *is* at its core. But when the function is actually being used, it is of no concern. You can think of this as abstracting the for-loop (it becomes assumed.)

They are actually often used *together* with first-order functions. Try not to think of a higher-order function as a specific feature. They can behave as a for-loop. But they can also be used as a way to instantiate an event – in which case a first-order function would be used together with it as a callback function. They are not limited to a single purpose. They enable a few different logical patterns

that cannot be created using first-order functions alone.

For example **Array.map** iterates through a set of values and applies a modification.

The **Array.reduce** "reduces" a set of values to a single value.

Event-based **setTimeout** is a higher-order function, and so is **addEventListener**.

14.0.2 Definition

A **higher-order function** is a function that either takes a function as one of its parameters or returns a function (or both.)

14.0.3 Abstract

Here is one way of visually thinking about the pattern of a high-order function. It exists as a higher level thinking.

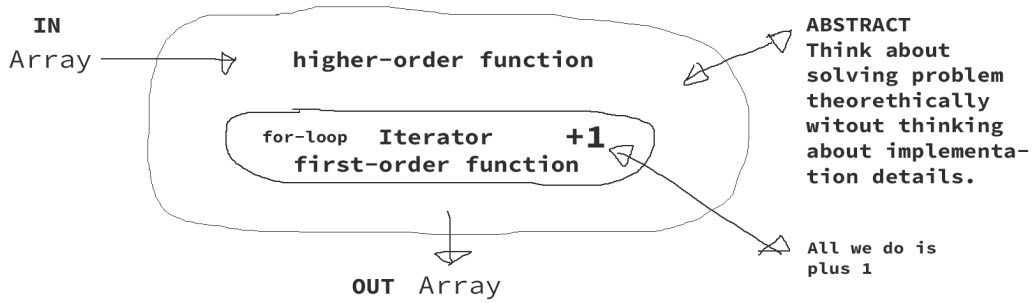
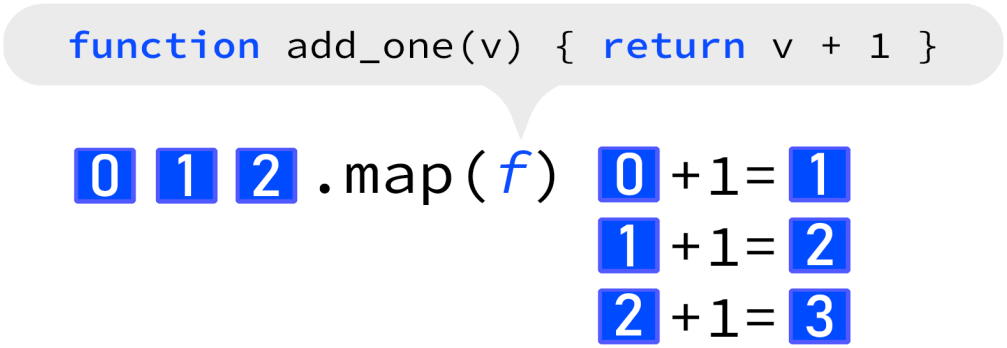


Figure 14.1: Visualizing a higher-order function.

But this still doesn't tell us much about how they're used or what they can actually do. Let's take a look at few different examples.

14.0.4 Iterators

The **Array.map** method is one of the most common higher-order functions. It takes a function to run on every item in the array. Then it returns a modified *copy* of the original array:



Here is a rather abstract way of thinking about the problem: "Add 1 to each item in the array." Simple logic which is defined in **add_one** function.

The **Array.map** method does not expose its loop implementation. The idea is not to make the iterator merely more efficient either (although, that would help) but hide it completely. We are only concerned with supplying a first-order function to the map method. Internally, it will run the function on each value in the array.

This is a very powerful technique that can apply to many problems. But the greatest advantage of using a higher-order function is that it abstracts problem solving. It helps us focus on the key: running the function on each individual item in the array, while abstracting away the for-loop (or while loop).

Let's create the function that will modify the values. The body of the function depends on what type of modifications you're looking to apply to each array item.

We will create a *first-order* function called **add_one**, which simply adds 1 to the value. This is just a helper function that will work together with a higher-order function (first-order and higher-order functions are often used together.)

```
// Function that adds 1 to any numeric value
function add_one(value) { return value + 1; }
```

Figure 14.2: A first-order function **add_one**.

For a function to qualify as a higher-order function it either needs to *take* a function as one of its parameters, or *return* a function. As long as one of those conditions is met we are creating a higher-order function.

The **map** function will take an array to work on. It will return a copy of that array with each item modified by the **add_one** function we have written earlier, which will be passed as the *second* parameter.

Let's write our own version of **map** which will be similar to what **Array.map** does:

```
// Higher-order map function source code:
function map(array, f ) {
  let copy = []; // return value placeholder array
  for (let index = 0; index < array.length; index++) {
    let original = array[index];
    let modified = f (original);// return original + 1
    copy[index] = modified; building return value one index at a time
  }
  return copy;
}
```

Figure 14.3: Complete source code of a higher-order function **map**. It is assumed that the array will contain only numeric values.

Line-by-line explanation of map() function

The function **map** takes two parameters: an arbitrary **array** of values and the **function** which we want to apply to each item in the array.

```
function map(array, f ) {
```

First, **copy** array was created and assigned to an empty array literal: []. This array will store a *modified* copy of the original array that was passed into the function.

```
let copy = []; // return value placeholder array
```

Then a `for-loop` iterates through the array we received as first parameter. This is the part we want to make abstract. When using the function we won't even have to think about the for-loop:

```
for (let index = 0; index < array.length; index++) {
```

Inside the loop, let's copy the value at current index in the original array to a temporary variable **original**.

```
let original = array[index];
```

Now let's pass the value in **original** variable to the first-order function that was passed to this function:

```
let modified = f(original);
```

The **f** function will do the magic (in our example, add 1 to the original value, but it can be anything) and return the modified value. So let's copy the modified value into our **copy** array which is a placeholder for the entire modified array.

```
copy[index] = modified;
```

Finally, a copy of the modified array is returned:

```
return copy;
```

Once all items have been copied and processed by **add_one** function, they will be stored in **copy** array which will then be used as the return value of the function.

Side note: The **reduce** method – which is also a higher-order function – uses something known as an accumulator. The accumulator in **Array.reduce** serves a similar purpose as the **copy** array in this example. In **reduce**, however, the accumulator is not an array – it is a single value that cumulatively gathers together all items from the array and combines them into a single return value. That's why reducers are a better solution when you need to *combine* values.

Calling our custom map function

To see how it all works, first, let's define an initial set of values to work with:

```
// Original array object
let array = [0,1,2];
```

Let's try out our **map** function in action:

```
// Let's try it out!
array = map(array, add_one); // [1,2,3]
```

Here **add_one** is the function from earlier in this chapter. It simply adds 1 to the value that was passed to it and returns it.

The result? The original array `[0,1,2]` is now `[1,2,3]`. All items in the array were incremented by 1.

We've just written our own map method that internally does exactly the same thing as the built-in method **Array.map**. This operation is so common that it was added as a native method on the Array object.

Calling Array.map function

Yes, we can do exactly the same thing using a built-in Array method map. It does exactly (or relatively) the same thing:

```
// Same thing using native Array.map method:
array.map(add_one); // [2,3,4]
```

Sounds easy. We could have used this method from the very start. But by writing our own map method we now actually understand how it works internally.

This will help us understand many other higher-order functions that implement iteration over a list of items, such as **filter**, **every**, **reduce**, etc. They all use similar internal code, with just a few slight differences.

What happened to the for-loop?

As you can see **Array.map** implements a for-loop internally. This isn't the issue of providing a more efficient for-loop, but rather, hiding it from our sight completely.

All we have to do is supply a function to the **Array.map** method. By hiding the iteration steps, we are left only with writing the actual function that compares, adds or filters each value individually.

This helps you to focus on solving the problem, instead of writing and re-writing a lot of repetitive code. But it also makes your code look cleaner.

14.0.5 Dos and Dont's

Often beginners use one method instead of another to accomplish relatively the same thing. While it "still works," this choice shouldn't be taken lightly.

Do use a high-order method for solving the problems it was *intended* to solve. Understanding the differences between **map**, **filter**, **reduce** matters. This isn't about just the syntax differences, but writing efficient code and avoiding anti-patterns. Try to find a proper method for the given task.

Do not use **filter** if you can get away with using **reduce** to accomplish the same action with *more efficiency*. Different high-order functions are designed to deal with problems specific to their implementation.

Chapter 15

Arrow Functions

ES6 Arrow Functions

Arrow functions were introduced in ES6 and provide a slim syntax for creating *function expressions* in JavaScript. Instead of defining the function using the `function` keyword, arrow functions have following syntax

```
001 // Arrow function syntax
002 () => {};
```

In many ways arrow functions behave in the same way as standard functions. You can assign them to a variable name:

```
001 // Create an arrow function and assign it to a variable
002 let fun_1 = () => {};
```

You call them by name:

```
001 // Call arrow function by its name
002 fun_1();
```

And have values returned from them using `return` keyword:

```
001 // Return a value from an arrow function
002 let fun_2 = () => { return 1; }
```

Now we can call function `fun_2` by its name:

```
001 fun_2(); // 1
```

The function returns a value of 1, just as expected.

`return`-less return

Arrow functions add one other unique feature. You can return values without using `return` keyword.

Even after removing {} brackets and `return` keyword in the example below `1` is still treated as a valid return value. This makes your code a lot more clean than using the old and redundant ES5 function syntax:

```
001 // Return value from arrow function without return keyword
002 let fun_3 = () => 1;
```

And now you can simply call this function as usual:

```
001 fun_3(); // 1
```

Remember that in JavaScript functions are expressions. An expression is anything that returns a single value. Similar to how a math equation returns a value.

Arrow functions help us expand on the idea by providing an even slimmer syntax (by removing parenthesis and the need to explicitly specify `return` keyword:)

```
001 let expression = e => e;
```

They turn functions into something that looks a lot like a math equation. (or a math function.) That's why they are often used in Functional Programming style.

```
001 expression(1); // 1
```

If you have a background in math you will feel at home using them!

Arrow Functions As Events

Some believe using arrow functions is a more elegant solution for events:

```
let clicked = (event) => { console.log(event.target) }
```

Or an even slimmer syntax:

```
let clicked = event => console.log(event.target);
```

15.0.1 Arrow Function Anatomy

Arrow functions do not have array-like **arguments** object. They also cannot be used as constructors. The **this** keyword points to the same value this points to in the scope just outside of the arrow function.

Arrow functions are *expressions* – they do not have a named syntax, like regular functions defined with **function** keyword. But just like regular functions, you can assign them to variable names.

```
001 // Create an arrow function and assign it to a variable
002 let fun_1 = () => {};
```

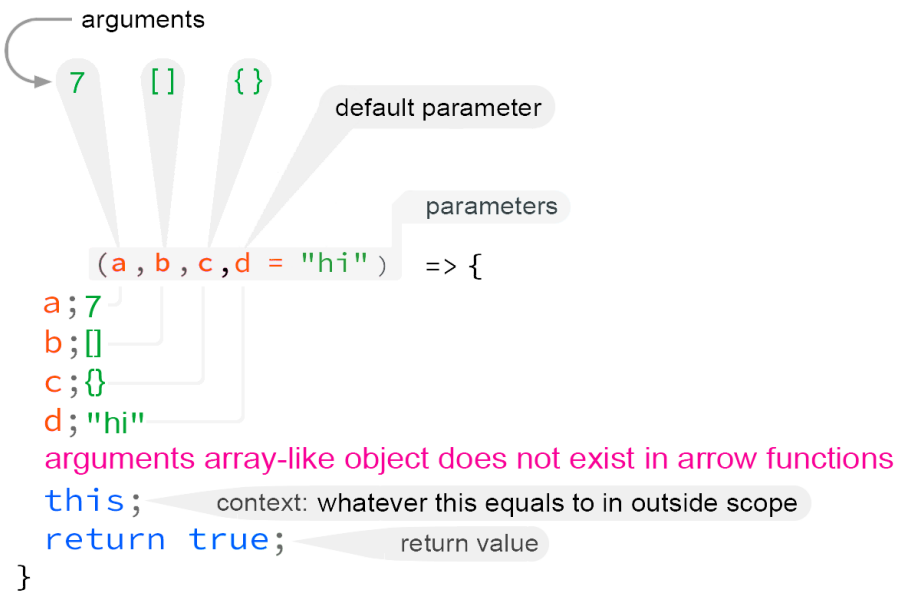


Figure 15.1: Arrow function anatomy.

Arguments

You can pass arguments to an arrow function via parameters.

```
// Arrow function syntax with arguments
(arg1, arg2) => { console.log(arg1, arg2); };
```

```
// Arrow function that prints out its arguments
let x = (arg1, arg2) => { console.log(arg1, arg2); };

x(1, 2); // Output: 1, 2
```

Returning From An Arrow Function

Arrow functions are primarily designed as expressions, so you might want to spend extra time on learning how they return values:

```
let boomerang = a => "returns";
let karma = a => { return "returns"; }
let prayer = a => { return random() >= 0.5 }
let time = a => { "won't return"; }

boomerang(1);      // "returns"
karma(1);          // "returns"
time(1);           // [undefined]

// Be careful when using with higher-order functions
let a = [1];
a.map(boomerang); // "returns"
a.map(karma);      // "returns"
a.map(time);       // [undefined]

console.log(x);
console.log(y);
console.log(z);

// Sometimes true.
prayer("Make me understand JavaScript.");
```

Takeaway: **time** arrow function is the only function that does not return a value at all. Be careful not to use this syntax with higher-order functions.

Similarities Between ES-style Functions

There are some similarities between arrows and classic functions. Most of the time you can use arrow functions as a replacement to standard ES5 functions without a hitch!

Let's try this example to get started with our discussion about arrow functions. I defined two classic ES5 functions **classic_one** and **classic_two**, followed by definition of **arrow** – an ES6-style arrow function:

```
001 function classic_one() {
002     console.log("classic function one.");
003     console.log(this);
004 }
005
006 var classic_two = function() {
007     console.log("classic function two.");
008     console.log(this);
009 }
010
011 let arrow = () => {
012     console.log("arrow function.");
013     console.log(this);
014 }
```

I added `console.log(this);` statement to the scope of each function, simply to see the outcome.

Let's call all 3 functions in a row:

```
001 // Call first function
002 classic_one();
003
004 // Call second function
005 classic_two();
006
007 // Call third function
008 arrow();
```

Console output:

```
classic function one.
▶ [Window]
classic function two.
▶ [Window]
arrow function.
▶ [Window]
>
```

When defined in global scope, it only *seems* like there is no difference between classic and arrow functions, in terms of `this` binding.

No `this` binding

Arrow functions do not bind **this** keyword. They look it up from whatever **this** equals in the outer scope, just like any other variable. Hence, you can say arrow functions have a "transparent" scope.

No `arguments` object

The `arguments` object does not exist in arrow function scope, you will get a reference error if you try to access it:

```
001 let a = () => {
002     console.log( arguments ); // arguments is not defined
003 }
```

Just as a reminder here, the `arguments` object does exist on classic ES5 functions:

```
001 function f() {
002     console.log( arguments ); // [object Arguments]
003 }
```

No Constructor

ES5-style functions are object constructors. You can create and call a function but you can also use same function as an object constructor – together with `new` operator – to instantiate an object. The function itself becomes class definition.

For this reason you would often hear it said that in JavaScript all functions are objects. After ES6 specification introduced arrow functions to the language this statement is no longer true. Arrow functions *cannot* be used as object constructors.

Therefore, arrow functions cannot be used to instantiate objects. They work best as event callbacks or function expressions in methods such as `Array.filter`, `Array.map`, `Array.reduce` and so on... In other words, they are more proper in context of Functional Programming style.

Thankfully, modern JavaScript is rarely written using ES5-style functions masquerading as object constructors. It's probably a good idea to start defining your classes using `class` keyword anyway, instead of using function constructors.

When classic and arrow functions are used as event callbacks

There is a difference between classic ES5 functions and arrows, when they are used as event callbacks.

Here is an example of an arrow function that outputs a string and `this` property to console in the event of a click on the document:

```
001 // Create an arrow function that outputs some info
002 let arrow = (event) => {
003     console.log("Hello. I am an arrow function.");
004     console.log(this);
005 }
006
007 // Attach arrow function to an event listener on document
008 document.addEventListener("click", arrow);
```

And here is the same exact event using classic ES5 function syntax:

```
001 function classic(event) {
002     console.log("Hello. I am a classic ES5 function.");
003     console.log(this);
004 }
005
006 // Attach an ES5 function to an event listener on document
007 document.addEventListener("click", classic);
```

So what's the difference?

Here is console output after clicking on document in each case:

```
    "Hello. I am an arrow function."
    [object Window]

    "Hello. I am a classic ES5 function."
    [object HTMLDocument]
```

Inside the arrow function's scope `this` property points to `Window` object.

In classic ES5 function `this` property points to the *target element* that was clicked.

What happens here is that arrow function took the Window context with it, instead of giving you the object that refers to the clicked element.

Inherited `this` Context

But wait, how did the arrow function inherit `Window` context in the first place? Is that because it was defined in global scope (same as the object of type `Window`)?

Not exactly.

The arrow function inherits the lexical scope based on where it was *used*, not where it was defined. Here it so happens that the arrow function was both defined and called in global scope context (`Window` object.)

To truly understand this, let's draw another example, where I will attach arrow function `B()` to a click event.

But this time, I will execute `addEventListener` function from another class called Classic instead of `Window` like in the previous example.

Remember that when you use `new` operator, you execute the function as though it was an object constructor. This means that every statement inside it will be executed from the context of its own instantiated object: `object`, not `window`.

```
001 // Define new class Classic using an ES5 constructor
002 function Classic() {
003     let B = () => {
004         console.log("Hello. I am arrow function B()");
005         console.log(this);
006     }
007     document.addEventListener("click", B);
008 }
009
010 let object = new Classic(); // Instantiate the object
```

A new context is created when using **new** operator to instantiate an object. Anything called from within that object will have its own context.

After running this code and clicking on document, we get following console output:

```
    "Hello. I am arrow function B()."
    [object Classic]
```

The object instance delimited by `[]` brackets is of type Classic. And that's exactly what happens here.

Event was attached from context of `Classic` constructor and not from the context of global scope object `Window` like in the previous examples.

The event has literally "taken the context it was executed from" with it into its own scope via the `this` property.

This type of context chaining is common to JavaScript programming. We'll see it once again when we explore *prototype-based inheritance* in more depth later in the book. The idea of execution context may start to become more clear by now.

Chapter 16

Creating HTML Elements Dynamically

JavaScript creates a unique object for each HTML tag currently present in your `*.html` document. They are *automatically* included to DOM (Document Object Model) in your application once the page is loaded into browser. But what if we want to add new elements without having to touch the HTML file?

Creating and appending another element to an existing element will dynamically insert it into the DOM and instantly display it on the screen as if it were directly typed into the HTML source code.

However, this element is not typed directly into your HTML document using HTML tag syntax. Instead, it is created dynamically by your application.

The method `createElement` natively exists on the document object. It can be used to create a new element:

```
001 // Dynamically create an HTML element
002 let E = document.createElement("div");     // Create ,<div>
```

```
001 // create a few element dynamically
002 let E1 = document.createElement("div");     // <div>
003 let E2 = document.createElement("span");    // <span>
004 let E3 = document.createElement("p");       // <p>
005 let E4 = document.createElement("img");     // <img>
006 let E5 = document.createElement("input");   // <input>
```

At this point none of the created elements are attached to DOM yet.

When adding a new HTML element dynamically, it is usually inserted into another element that already exists in the DOM. But before we can accomplish that, let's set some CSS styles on the newly created element.

16.0.1 Setting CSS Style

So far we created an empty element without dimensions, background color or border. At this point all of its CSS properties are set to defaults. We can assign a value to any standard CSS property via `style` property.

```
001 // create a <div> element dynamically
002 let div = document.createElement("div");
003
004 // Set ID of the element
005 div.setAttribute("id", "element");
006
007 // Set class attribute of the element
008 div.setAttribute("class", "box");
009
010 // Set element's CSS style via style property
011 div.style.position    = "absolute";
012 div.style.display     = "block";
013 div.style.width       = "100px";// px is required
014 div.style.height      = "100px";// px is required
015 div.style.top         = 0;      // px is not required on 0
016 div.style.left        = 0;      // px is not required on 0
017 div.style.zIndex      = 1000;   // z-index > zIndex
018 div.style.borderStyle = "solid";// border-style > borderStyle
019 div.style.borderColor = "gray"; // border-color > borderColor
020 div.style.borderWidth = "1px";  // border-width > borderWidth
021 div.style.backgroundColor = "white";
022 div.style.color       = "black";
```

In CSS dash (-) is a legal property name character. But in JavaScript it is always interpreted as the minus sign. Using it as part of a JavaScript identifier name will cause an error. For this reason, single-word CSS property names remain the same – **style.position** and **style.display** for example. Multi-word property names are changed to *camel-case* format, where the second word is capitalized. For example **z-index** becomes **.zIndex**, and **border-style** becomes **.borderStyle**.

16.0.2 Adding Elements To DOM with `.appendChild` method

Method *`element.`*`appendChild(`*`object`*`)` inserts an element object into DOM.

Here *`element`* can be any other element that currently exists in the DOM.

This method exists on all DOM element objects, including `document.body`.

document.body

Insert the element into the body tag using `appendChild` method:

```
001 // Add element to the DOM by inserting it into <body> tag
002 document.body.appendChild( div );
```

Although very common the body tag is not the only place you can add the newly created element.

getElementById

Insert element into another element by id:

```
001 // Insert element into another element with id = "id-1"
002 document.getElementById("id-1").appendChild( div );
```

querySelector

Insert element to any element selected using a valid CSS selector:

```
001 // Insert element into selected parent element
002 let selector = "#parent .inner .target";
003 document.querySelector( selector ).appendChild( div );
```

16.0.3 Writing A Function To Create Elements

Writing your own functions is fun. And sometimes necessary. In this section we will write our own function that makes it easy to create HTML elements dynamically. Before writing the function body, let's take a closer look at its parameters.

Function Parameters

To accommodate for most cases, we don't need to include all CSS properties, just ones that have most impact on element's visual appearance.

```
001 let element = (id,     // id attribute
002                type,   // "static", "relative" or "absolute"
003                l,      // left     (optional)
004                t,      // top      (optional)
005                w,      // width    (optional)
006                h,      // height   (optional)
007                z,      // z-index  (optional)
008                r,      // right    (optional)
009                b) => { // bottom   (optional)
010          /* function body */
011 }
```

Most of the parameters are optional. If you skip them, either default values will be used – defined using `const` keyword inside the function body (*see next page*) – or not assigned (if you pass `null`, for example.)

Last parameters on the argument list: `r` and `b` (`right` and `bottom`) will override standard placement in `top` and `left` corner of the parent element.

Using this function we can create basic HTML elements with one line of code:

```
001 // Static 100px x 25px at 0 x 0 with "unset" z-index
002 let A = element("id-1", "static", 0, 0, 100, 25, "unset");
003
004 // Relative 50px x 25px at 0 x 0 with z-index = 1
005 let B = element("id-2", "relative", 0, 0, 50, 25, 1);
006
007 // Absolute 50px x 25px at 10px x 10px z-index = 20
008 let C = element("id-3", "absolute", 10, 10, 50, 25, 20);
```

Function Body

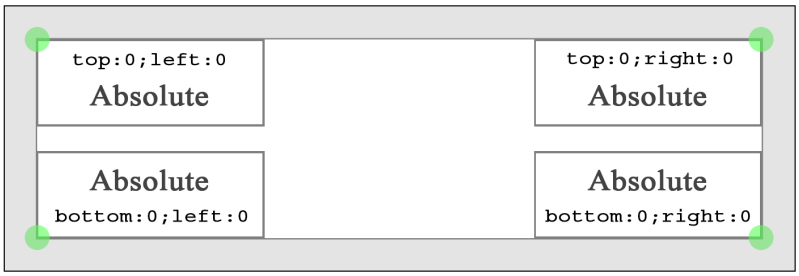
Let's take a look at the body of the element-creation function:

```
001 // Creates a generic HTML element
002 export let element = (id, type, l, t, w, h, z, r, b) => {
003
004   // defaults - used to replace missing arguments
005   const POSITION = 0;
006   const SIZE = 10;
007   const Z = 1;
008
009   // create a <div> element dynamically
010   let div = document.createElement("div");
011
012   // Set ID of the element
013   div.setAttribute("id", id);
014
015   // Set absolute behavior
016   div.style.position = type;
017   div.style.display = "block";
018
019   if (right) // If right is provided, reposition element
020       div.style.right = r ? r : POSITION + "px";
021   else
022       div.style.left = l ? l : POSITION + "px";
023
024   if (bottom) // If bottom is provided, reposition element
025       div.style.bottom = b ? b : POSITION + "px";
026   else
027       div.style.left = t ? t   : POSITION + "px";
028
029   // If w,h and z were supplied, use them
030   div.style.width  = w ? w : SIZE + "px";
031   div.style.height = h ? h : SIZE + "px";
032   div.style.zIndex = z ? z : Z;
033
034   // Return the element object we just created
035   return div;
036 };
```

Figure 16.1: Place this function into separate file `common-styles.js`

Creating UI elements often requires pixel-perfect precision. This function will create a `position:absolute` element (*unless otherwise specified*) with default size of `10px` in each direction, unless replacement values are supplied via its arguments: `w` and `h` parameters.

As a quick reminder here is the behavior of HTML elements with `position` set to `absolute` based on point of attachment.



Note that the element can be attached to the logical coordinate position within the parent. For example `top:0; right:0` attaches the element to the upper right corner of the parent.

The direction in which the attached element will move, when its coordinates are provided using negative values are displayed in the following diagram:

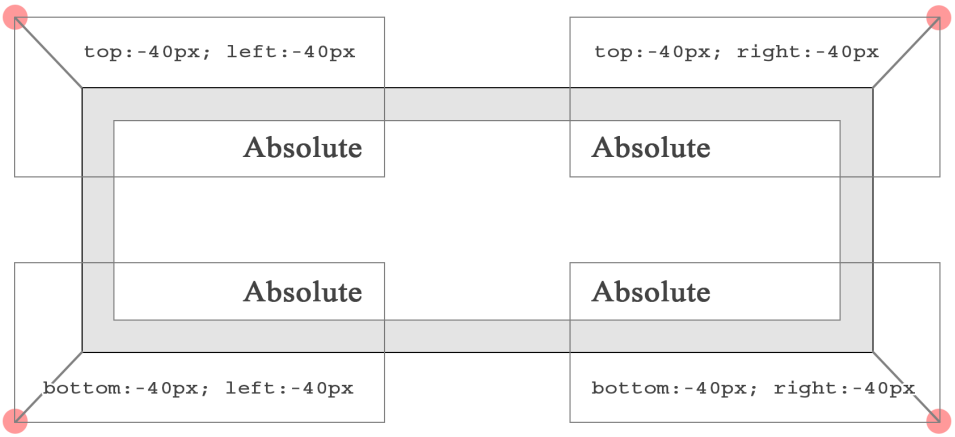


Figure 16.2: This depends on corner the element was attached to using a combination of properties `left`, `top`, `right` and `bottom`.

Importing And Using `element()` Function

Let's import the function we created above into our project.

To import a module `type` attribute on `script` tag must be set to `"module"`.

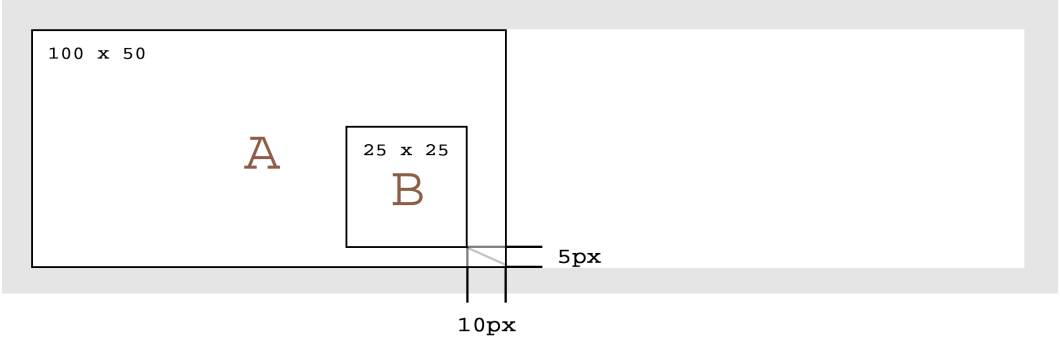
```
<script type = "module">

  // import function absolute()
  import { absolute } from "./common-styles.js";

  // create a new element located at top / left
  let A = absolute("id-1", 0, 0, 100, 50, 1);

  // create a new element located at right / bottom
  let B = absolute("id-2", null, null, 25, 25, 1, 10, 5);

  // nest B inside A
  B.addElement(A);

  // Add A (together with element B nested in it) to <body>
  document.body.addElement(A);

</script>
```

We can now create an HTML element using just one line of code.

In this example we created two elements `A` and `B`. Then we nested element `B` in element `A`, and attached element A to the body container.

The result of this code is displayed below:

<body>



16.0.4 Creating objects using function constructors

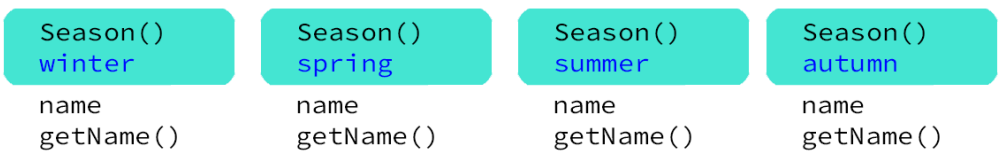
Let's create a function called **Season**:

```
001 function Season(name) {
002     this.name = name;
003     this.getName = function() {
004         return this.name;
005     }
006 }
```

To instantiate four seasons:

```
001 let winter = new Season("Winter");
002 let spring = new Season("Spring");
003 let summer = new Season("Summer");
004 let autumn = new Season("Autumn");
```

We just created 4 instances of the same type:



This creates a problem, because function **getName** is copied 4 times in memory, but its body contains exactly the same code.

In JavaScript programs Objects and Arrays are created all the time. Imagine if you instantiated 10000 or even 100000 objects of a particular type, each storing a copy of the same exact method.

This is rather wasteful. Could we somehow have a single **getName** function?

The answer is yes. For example, native function **.toString()** you may have used before as **Array.toString()** or **Number.toString()** exists in memory at a single location, but it can be called on all built-in objects! How does JavaScript do it?

Chapter 17

Prototype

17.0.1 Prototype

When a function is defined two things happen: the function object is created, because functions are objects. Then, a completely separate prototype object is created. The prototype property of the defined function will point to it.

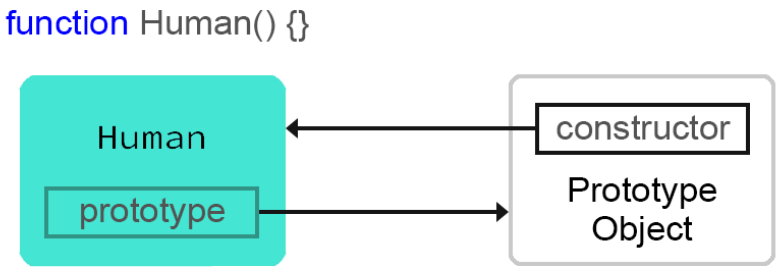
Let's say we defined a new function **Human**:

```
001 function Human(name) { }
```

You can verify that prototype object is created at the same time:

```
001 typeof Human.prototype; // "object"
```

Human.prototype will point to the prototype object. This object has another property called **constructor**, which points back to the **Human** function:



Human is a constructor function, used to create objects of type **Human**. Its prototype property points to a separate entity in memory: **prototype object**. There is one separate prototype object per each unique object type (class).

Some will argue that there are no classes in JavaScript. But technically, Human is a unique object type, and basically that's what a class is.

If you come from C++ background, you can probably refer to **Human** as a class. A class is an abstract representation of an object. It's what determines its type.

Note, **prototype** property is not available on an instance of an object, only on the constructor function. On an instance, you can still access prototype via **__proto__**, but should probably use static method **Object.getPrototypeOf(instance)** which returns the same prototype object as **__proto__** (in fact**__proto__()** is a getter.)

17.0.2 Prototype on Object Literal

To draw a simple example, let's create an object literal:

```
001 let literal = {
002     prop: 123,
003     meth: function() {}
004 };
```

Internally it is wired into prototype as an object of type **Object**, even though it wasn't created using the **new** operator.

```
001 literal.__proto__;              // Object {}
002 literal.__proto__.constructor; // f Object { [native code] }
003 literal.constructor;            // f Object { [native code] }
```

When **literal** was created, **literal.__proto__** was wired to point to **Object.prototype**:

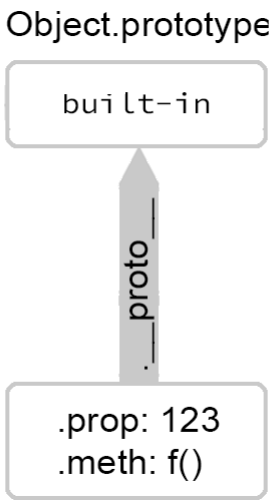


Figure 17.1: Here object literal's __proto__ points to Object.prototype

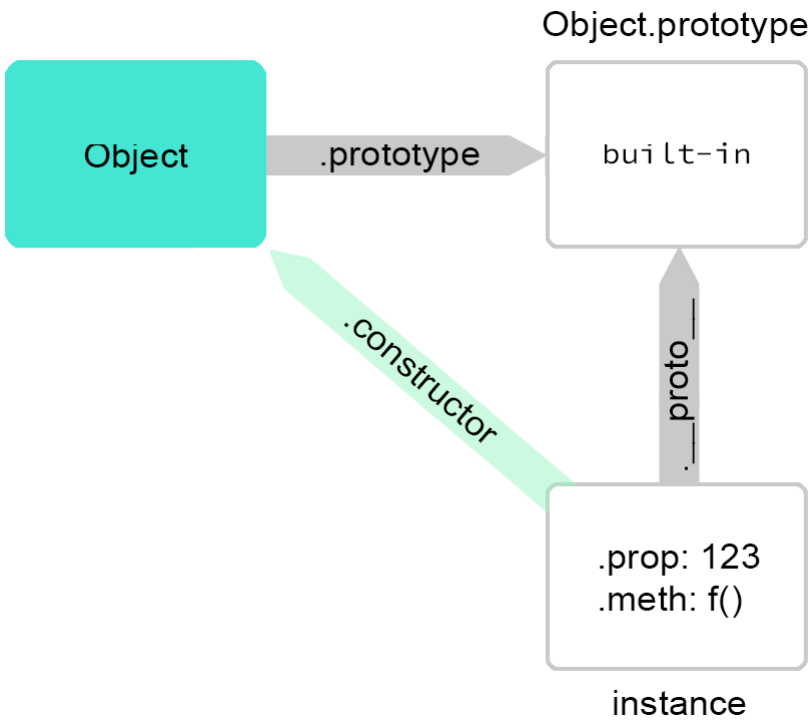
Object.prototype was already created internally by JavaScript. Whenever a new object type is defined, a secondary object to serve as its prototype is created.

17.0.3 Prototype Link

When an object is instantiated using **new** keyword, the constructor function executes to build the instance of that object.

```
001 let instance = new Object();
002 instance.prop = 123;
003 instance.meth = function(){}
```

In this case **Object** constructor function is executed and we get a constructed link, that looks like this:



The **.prototype** property points to a separate object: the built-in prototype object.

In this case it's **Object.prototype**. It is similar to **Human.prototype** from the earlier example: in this case we just don't control how **Object** was created, because Object is a preexisting built-in type.

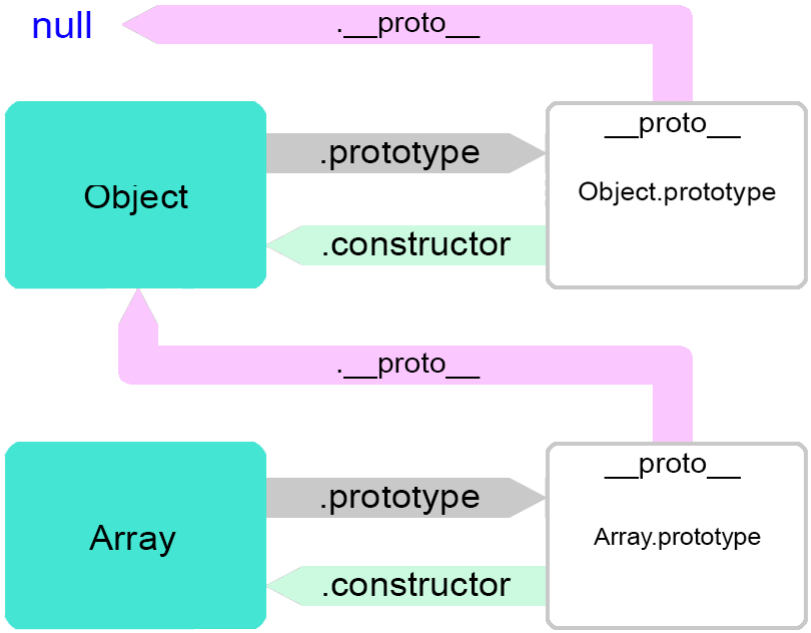
The **instance** of an object of type Object has __**proto**__ property which points to the prototype object of the constructor from which it was instantiated.

This three-way relationship between the **Object**, secondary **prototype** object that was created, and the **instance** of that object with __**proto**__ pointing to Object's prototype object is a peculiar structure.

This pattern represents just one link in a prototype chain of objects.

17.0.4 Prototype Chain

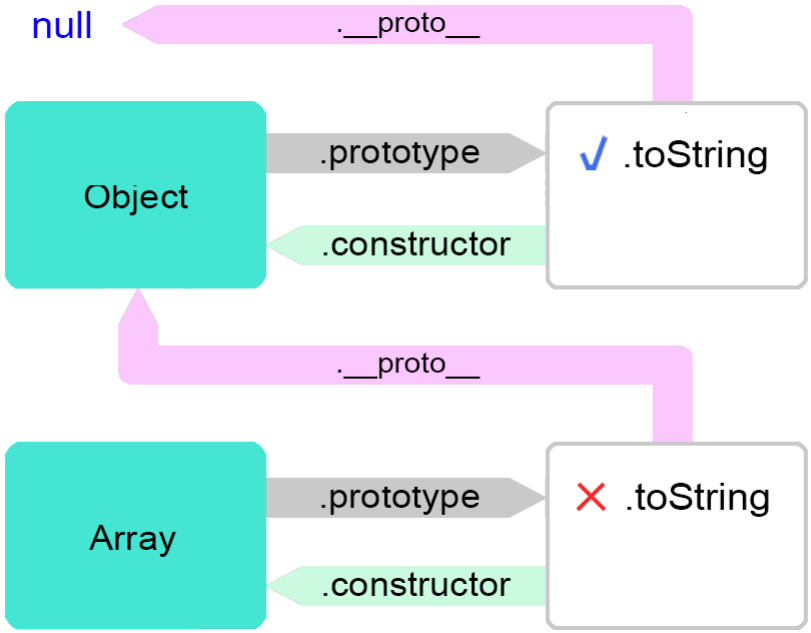
It can be argued that Array is a child of its parent type Object.



Object.prototype is Object, but not because Object is inherited from Object. This is simply because the prototype itself is just an object: they co-exist.

You can think of this as Object's prototype being **null**, because it's the top-level object on the prototype chain. In other words Object does not have an abstract prototype. Object does have a "ghost" prototype object just like any other type.

17.0.5 Method look-up



When you call **Array.toString()** what actually happens is, JavaScript will first look for method **toString** on the prototype of **Array** object. But it does not find it there. Next, JavaScript decides to look for **toString** method on the prototype property of Array's parent class: Object.

It finally finds **Object.prototype.toString()** and executes it.

17.0.6 Array methods

In previous section we've taken a look at prototype chain and how **.toString** method is found by traversing the prototype chain. But **.toString** is available on all objects that stem from the **Object** type (which is most built-in types.)

The same is true for **Number**, **String** and **Boolean** built-in types. You can call **toString** method on each one of them, and yet it exists only in one place in memory on **Object.prototype** property.

Methods native to Array type should exist on **Array.prototype** object.

It is obvious that there isn't much use in having higher-order functions like **.map**, **.filter** and **.reduce** attached to Number or Boolean types.

The fact is, every single Array method already exists on **Array.prototype**:

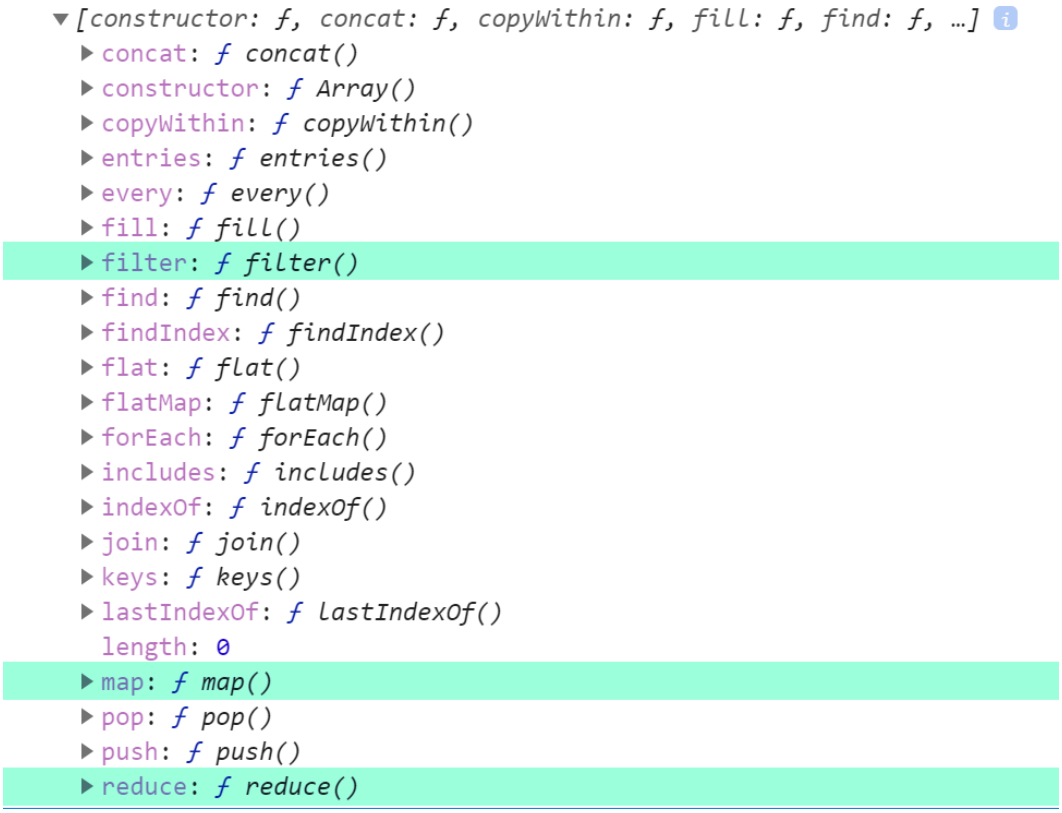


Figure 17.2: Looks like our favorite methods **.map**, **.filter** and **.reduce** live on **Array.prototype** object.

If you want to extend functionality of the **Array** method specifically, attach a method to **Array.prototype.my_method**.

17.1 Parenting

But how does **Array**, **Number**, etc. know that **Object** is its parent?

That's exactly what prototype inheritance is about: creating links between children and parent objects. Often this is referred to as **prototype chain**.

Furthermore, it can be said that prototype structure provides some sort of imitation of "traditional" (or "classic") class inheritance as seen in C++, which is one of the few truly object-oriented programming languages. (It actually supports all features that make a language to be considered object-oriented.)

17.1.1 Extending Your Own Objects

Number's and Array's parent is Object. This is great, but what if we want to extend our own object from another object?

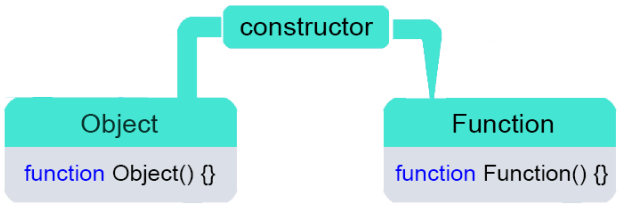
As we saw from the preceding diagrams __**proto**__ getter is part of internal prototype implementation. It's crucial for establishing the link, but we shouldn't mess with it directly. For the same reason.

JavaScript is a dynamically typed language, so you can try to create some object constructors and rewire the __**proto**__ property on their prototype object to the "parent", but this is often considered to be a hack. In practical coding situation, you will actually never need to do anything like this. There is literally no software you would possibly be writing that modifies the internal function of prototype.

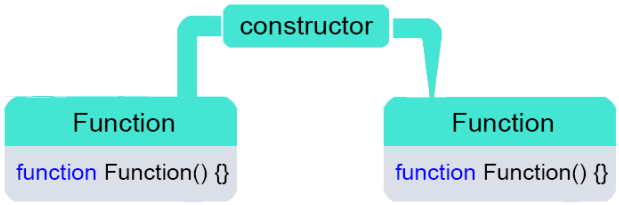
After EcmaScript 6 you are encouraged to create and extend classes using **class** and **extends** keywords and let JavaScript worry about prototype links.

17.1.2 constructor property

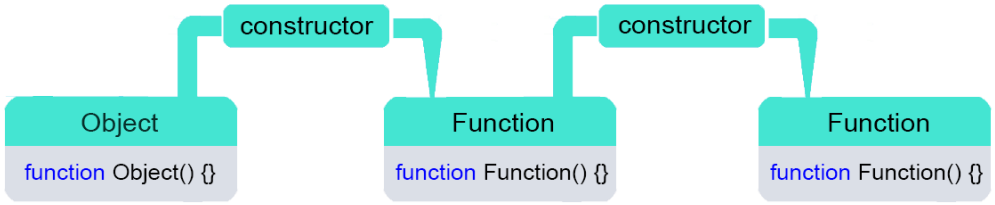
The **constructor** property of **Object** class points to **Function**:



The **constructor** property of **Function** class points to **Function**:



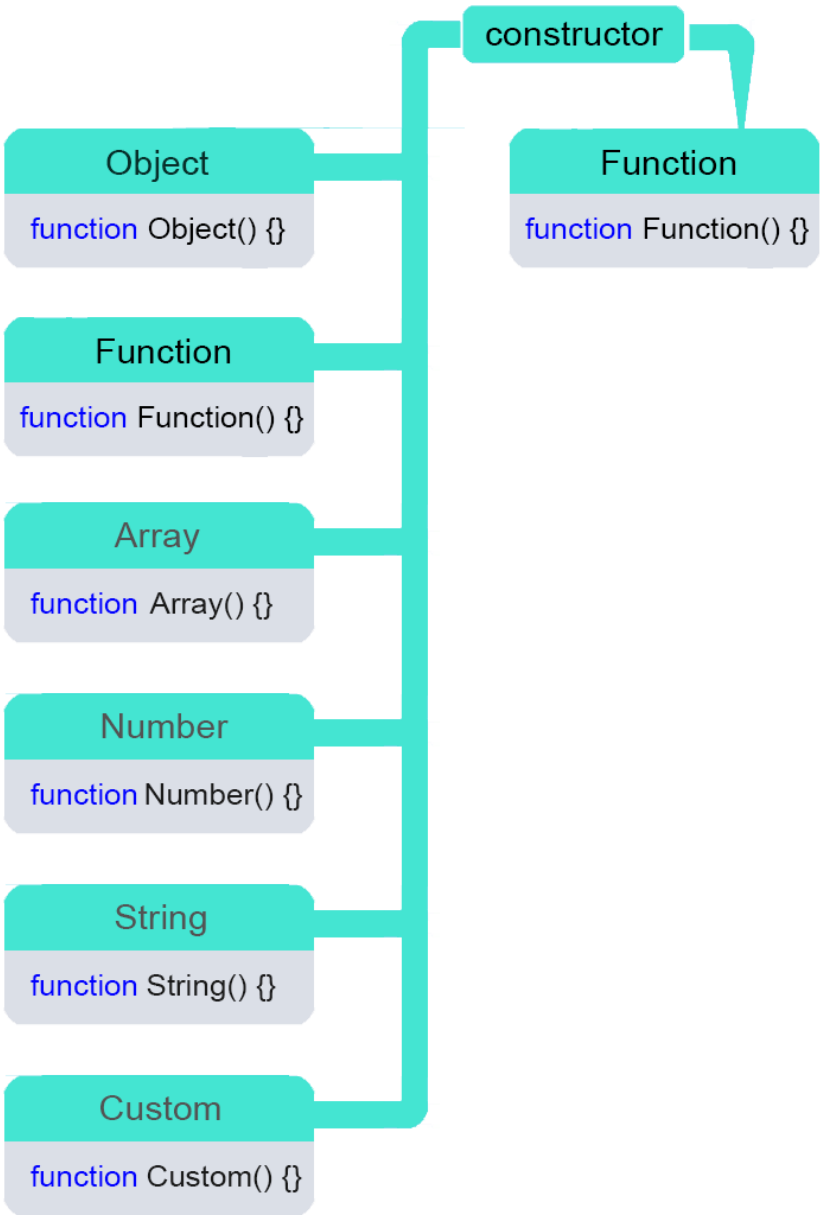
This creates a circular dependency around the Function class:



Function.constructor is Function (circular.) But Object.constructor is also Function. This can imply that a class is constructed using a function. Yet, Function itself is a class. This is circular dependency.

17.1.3 Function

Function is the constructor of all object types.



17.2 Prototype In Practice

Understanding how prototype works is a gradual process. It might be a difficult task, considering JavaScript language has evolved over the years. To get a better idea of how it all works, we'll start from the very beginning.

We've already covered the theory behind prototype in the previous section of this chapter. In this section we will finally arrive at prototype in terms of how fits into the big picture when it comes to actually writing code.

This section is a thorough walk-through that demonstrate different ways of working with objects. And what's a better place to start than the object literal?

17.2.1 Object Literal

In this example **cat** object was defined using a simple **object literal** syntax. In some ways, under the hood JavaScript wires up all the prototype linking.

Throughout the following sections of this chapter we will gradually update this example and build on it to finally arrive at how prototype can be useful to you as a JavaScript programmer.

```
001 let cat = {}
002
003 cat.name = "Felix";
004 cat.hunger = 0;
005 cat.energy = 1;
006 cat.state = "idle";
```

Figure 17.3: Meet Felix the cat, represented by an object literal.

We named our cat specimen Felix, gave him 0 level of hunger and 1 unit of energy. Currently Felix is in idle state. Just where we want him to be!

```
001 // Sleep to restore energy
002 cat.sleep = function(amount) {
003   this.state = "sleeping";
004   console.log(`{$this.name} is ${this.state}.`);
005   this.energy += 1;
006   this.hunger += 1;
007 }
008
009 // Wake up
010 cat.wakeup = function() {
011   this.state = "idle";
012   console.log(`{$this.name} woke up.`);
013 }
014
015 // Eat until hunger is quenched
016 cat.eat = function(amount) {
017   this.state = "eating";
018   console.log(`${this.name} is ${this.state}
019                ${amount} unit(s) of food.`);
020   if (this.hunger -= amount <= 0)
021     this.energy += amount;
022   else
023     this.wakeup();
024 }
025
026 // Wandering depletes energy,
027 // If necessary, sleep for 5 hours to restore energy
028 cat.wander = function() {
029   this.state = "wandering";
030   console.log(`{$this.name} is ${this.state}.`);
031   if (--this.energy < 1)
032     this.sleep(5);
033 }
```

Methods **sleep**, **wakeup**, **eat** and **wander** were added directly to the instance of the **cat** object. Each method has basic implementation that either restores or depletes cats energy.

```
035 cat.sleep(); // "Felix is sleeping."
```

Figure 17.4: To restore cat's energy, we can invoke the **sleep** method.

17.2.2 Using Function Constructor

Even after some sleep Felix feels a bit of melancholy. He needs a friend.

But instead of creating a new object literal, we can place the same code inside a function called **Cat** to represent a global **Cat** class:

```
001 function Cat(name, hunger, energy, state) {
002
003   let cat = {}
004
005   cat.name = name;
006   cat.hunger = hunger;
007   cat.energy = energy;
008   cat.state = state;
009
010   cat.sleep = function(amount) { /* implement */ }
011   cat.wakeup = function(amount) { /* implement */ }
012   cat.eat = function(amount) { /* implement */ }
013   cat.wander = function(amount) { /* implement */ }
014
015   return cat;
016 }
```

Note that all that was done here is we moved the exact same code we wrote in the previous section into a function and returned the object using the **return** keyword.

The methods implementation remained the same too. I simply used the comment markers `/* implement */` to avoid repeating the same code again.

```
018 let felix = Cat("Felix", 10, 5, "idle");
019 felix.sleep();  // "Felix is sleeping."
020 felix.eat(5);   // "Felix is eating 5 unit(s) of food."
021 felix.wander(); // "Felix is wandering."
022
023 let luna = Cat("Luna", 8, 3, "idle");
024 luna.sleep();  // "Luna is sleeping."
025 luna.wander(); // "Luna is wandering."
026 luna.eat(1);   // "Luna is eating 1 unit(s) of food."
```

Now, saving space, we can create two separate cats: Felix and Luna.

17.2.3 Prototype

Branching out from previous example, we see a problem.

All of the methods of **Felix** and **Luna** take twice as much space in memory. This is because we are still creating two object literals for each cat.

And this is the problem prototype tries to solve.

Why don't we take all of our methods and place them at a single location in memory instead?

```
001 const prototype = {
002   sleep(amount) { /* implement */ },
003   wakeup(amount) { /* implement */ },
004   eat(amount) { /* implement */ },
005   wander(amount) { /* implement */ }
006 }
```

Now all of our neatly packaged methods share a single place in memory.

Let's go back to our **Cat** class implementation, and wire the methods from the above **prototype** object directly into each method on the object:

```
001 function Cat(name, hunger, energy, state) {
002
003   let cat = {};
004
005   cat.name = name;
006   cat.hunger = hunger;
007   cat.energy = energy;
008   cat.state = state;
009
010   cat.sleep = prototype.sleep;
011   cat.wakeup = prototype.wakeup;
012   cat.eat = prototype.eatsleep;
013   cat.wander = prototype.wander;
014
015   return cat;
016 }
```

Before we see how JavaScript does this, let's take a look at something else.

17.2.4 Creating objects using Object.create

In JavaScript we can also create objects using **Object.create** method, which takes a clean slate object as one of its arguments:

```
001 const cat = {
002   name: "Felix",
003   state: "idle",
004   hunger: 1,
005 }
006
007 const kitten = Object.create(cat);
008 kitten.name = "Luna";
009 kitten.state = "sleeping";
```

Now let's take a look at **kitten**:

```
001 console.log(kitten);
```

Mysteriously, object **kitten** has only two methods – it is missing **hunger** property.

```
  ▶ {name: "Luna", state: "sleeping"}
> |
```

To explain this, let's see what will happen if we actually try to output it:

```
001 console.log(kitten.hunger);
```

Console output:

```
  1
> |
```

But wait, the console is telling us **hunger** actually exists. What's going on here?

This behavior is unique to objects created via **Object.create** method. When we try to get kitten.hunger, JavaScript will look at **kitten.hunger**, but will not find it there (because it wasn't created directly on the *instance* of the **kitten** object.)

Then what happens is JavaScript will look at **.hunger** property in **cat** object. Because **kitten** was created via **Object.create(cat)**, **kitten** considers **cat** to be its parent so it looks there.

Finally it finds it on **cat.hunger** and returns **1** in console. Again, property **hunger** is stored only once in memory.

17.2.5 Back To The Future

Let's rewind a bit and go back to the earlier example from section called **Using Function Constructor** fully equipped with new knowledge about **Object.create**.

```
001 const prototype = {
002   sleep(amount) { /* implement */ },
003   wakeup(amount) { /* implement */ },
004   eat(amount) { /* implement */ },
005   wander(amount) { /* implement */ }
006 }
007
008 function Cat(name, hunger, energy, state) {
009
010   let cat = Object.create(prototype);
011
012   cat.name = name;
013   cat.hunger = hunger;
014   cat.energy = energy;
015   cat.state = state;
016
017   // We no longer need them here
018   // cat.sleep = prototype.sleep;
019   // cat.wakeup = prototype.wakeup;
020   // cat.eat = prototype.eatsleep;
021   // cat.wander = prototype.wander;
022
023   return cat;
024 }
```

Let's delete the part where we wired our own prototype object into the methods of **Cat** class, and instead pass them into the native **Object.create** method which will now reside inside our **Cat** function (source code listing above.)

Now we can create **felix** and **luna** via this new **Cat** function as follows:

```
001 let felix = Cat("Felix", 10, 5, "idle");
002 felix.sleep();  // "Felix is sleeping."
003
004 let luna = Cat("Luna", 8, 3, "idle");
005 luna.sleep();  // "Luna is sleeping."
```

Now we get the ideal syntax, and **sleep()** is defined only once in memory. No matter how many **felixes** or **lunas** you create, we're no longer wasting memory on their methods, because they are defined only once.

17.2.6 Function Constructor

Let's recall that each Object has a prototype property pointing to its ghost prototype object:

```
001 console.log(typeof Object.prototype); // "object"
```

So now what we can do is attach all **Cat** methods directly to its built-in **prototype** property instead of our own "prototype" object we created earlier:

```
001 function Cat(name, hunger, energy, state) {
002
003   let cat = Object.create(Cat.prototype);
004
005   cat.name = name;
006   cat.hunger = hunger;
007   cat.energy = energy;
008   cat.state = state;
009
010   return cat;
011 }
```

```
013 // Now define methods on actual Cat.prototype object:
014 Cat.prototype.sleep = function { /* implement */ };
015 Cat.prototype.wakeup = function { /* implement */ };
016 Cat.prototype.eat = function { /* implement */ };
017 Cat.prototype.wander = function { /* implement */ };
```

```
001 let luna = Cat("Luna", 5, 1, "sleeping");
002 felix.wakeup();  // "Luna is sleeping."
```

In this scenario, JavaScript will look for **.sleep** on **luna** object, and will not find it there. It will then look for **.sleep** method on **Cat.prototype**. It finds it there and the method is invoked.

```
001 let felix = Cat("Felix", 5, 1, "sleeping");
002 felix.wakeup();  // "Felix woke up."
```

The same happens here, **.wakeup** method is executed on **Cat.prototype.wakeup**, not on the instance itself.

Therefore the main purpose of prototype is to serve as a special look up object, which will be shared across all instances of objects instantiated with its constructor function while preserving memory.

17.2.7 Along came `new` operator

We can wipe out everything we learned up to this point and replace it all with **new** operator – which will *automatically* do every single thing we've just explored in the previous sections of this chapter!

```
001 function Cat(name, hunger, energy, state) {
002   let cat = Object.create(Cat.prototype);
003   cat.name = name;
004   cat.hunger = hunger;
005   cat.energy = energy;
006   cat.state = state;
007   return cat;
008 }
```

Let's remove **Object.create** and **return cat** from our class definition.

In JavaScript functions defined with **function** keyword are hoisted. This means we can add methods to **Cat.prototype** before **Cat** is defined:

```
001 // Define some methods on Cat.prototype object:
002 Cat.prototype.sleep = function { /* implement */ };
003 Cat.prototype.wakeup = function { /* implement */ };
004 Cat.prototype.eat = function { /* implement */ };
005 Cat.prototype.wander = function { /* implement */ };
```

Followed by **Cat** definition:

```
007 function Cat(name, hunger, energy, state) {
008   cat.name = name;
009   cat.hunger = hunger;
010   cat.energy = energy;
011   cat.state = state;
012 }
```

Now we can instantiate **luna** and **felix** as follows:

```
let luna = new Cat("Luna", 8, 3, "idle");
luna.sleep();  // "Luna is sleeping."

let felix = new Cat("Felix", 5, 1, "sleeping");
felix.wakeup();  // "Felix woke up."
```

17.2.8 ES6 The `class` keyword

Everything we've just explored about prototype was converging toward the **class** keyword added in EcmaScript 6.

How prototype works is a common subject during JavaScript interviews. But in production environment, you will never (spelling is correct) have to touch it at all in your entire career as a front-end software engineer.

```
class Cat {
  constructor(name,hunger, energy, state) {
    this.name = name;
    this.hunger = hunger;
    this.energy = energy;
    this.state = state;
  }
  sleep()  { /* implement */ };
  wakeup() { /* implement */ };
  eat()    { /* implement */ };
  wander   { /* implement */ };
}
```

Use **class** and **new** keywords. Let JavaScript worry about prototype:

```
let luna = new Cat("Luna", 8, 3, "idle");
luna.sleep();  // "Luna is sleeping."

let felix = new Cat("Felix", 5, 1, "sleeping");
felix.wakeup();  // "Felix woke up."
```

In the next section we will take this concept further to design an entire application using **OOP**: **Polymorphism** with examples via **Inheritance** and **Object Composition** and just a bit of **Functional Programming** style.

Chapter 18

Object Oriented Programming

In this chapter we will exemplify OOP by building a cooking range with stove.

The best example of Object Oriented Programming would be based on many different types of objects. We will define a class for each abstract type: **Fridge**, **Ingredient**, **Vessel**, **Range**, **Burner**, and **Oven**.

We will take a look at two key OOP principles: **inheritance** and **polymorphism** in the sense of how it actually relates to JavaScript code.

18.1 Ingredient

The **Ingredient** class will be our generic class for instantiating ingredients from. It will have properties: **name**, **type** and **calories** which will be enough to describe pretty much any ingredient we might need.

Here's a list of the common ingredients we will create: *water*, *olive_oil*, *broth*, *red_wine*, *bay_leaf*, *peppercorn*, *beef*, *chicken*, *bacon*, *pineapple*, *apple*, *blueberry*, *mushroom*, *carrot*, *potato*, *egg*, *cheese*, *sauce*, *oatmeal*, *rice*, *brown_rice*, *cheese*.

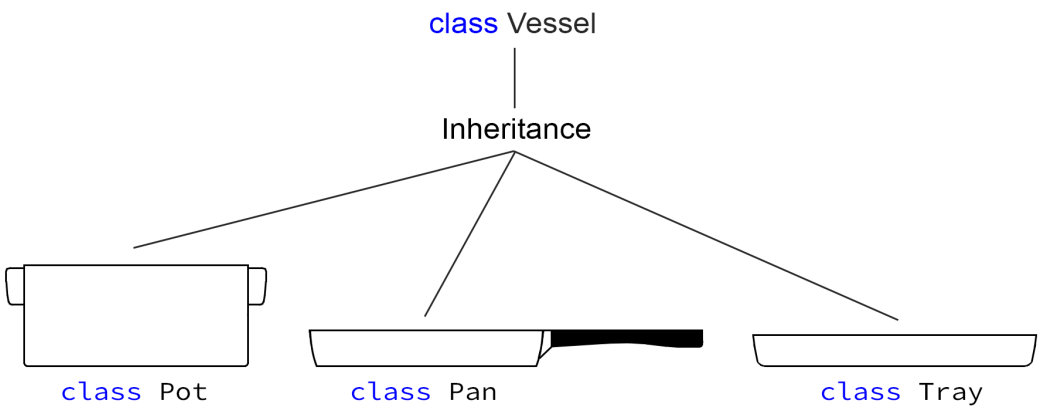
I think it's enough to make several interesting meals! Ingredient class itself will have **static** properties to describe the type of the ingredient. We'll take a look at how this works when we get to source code. (eg: **Ingredient.vegetable**)

18.2 FoodFactory

FoodFactory will be a class that will create a completely new instance of an ingredient. This way we don't end up reusing the same **Ingredient** object instance in different cooking vessels at the same time! Because, as you may know, in JavaScript variable names are only references (links) to the same object instance.

18.3 Vessel

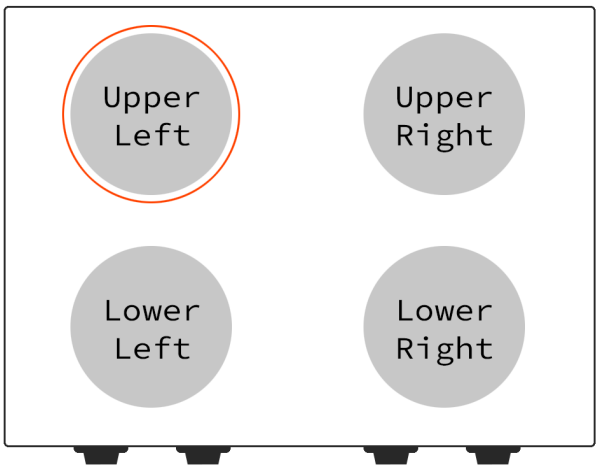
Classes **Pot**, **Pan** and **Tray** demonstrate inheritance because they are derived from the same abstract class **Vessel** (as in cooking vessel.)



Vessel class will have an **add()** method for adding ingredients. This way we can choose which ingredients to place inside each created vessel for baking, boiling or baking (depending on which vessel type was chosen.)

18.4 Burner

There will be four burners on the range surface, each represented by a unique instance of the class **Burner**. You can turn burners **on()** or **off()**.



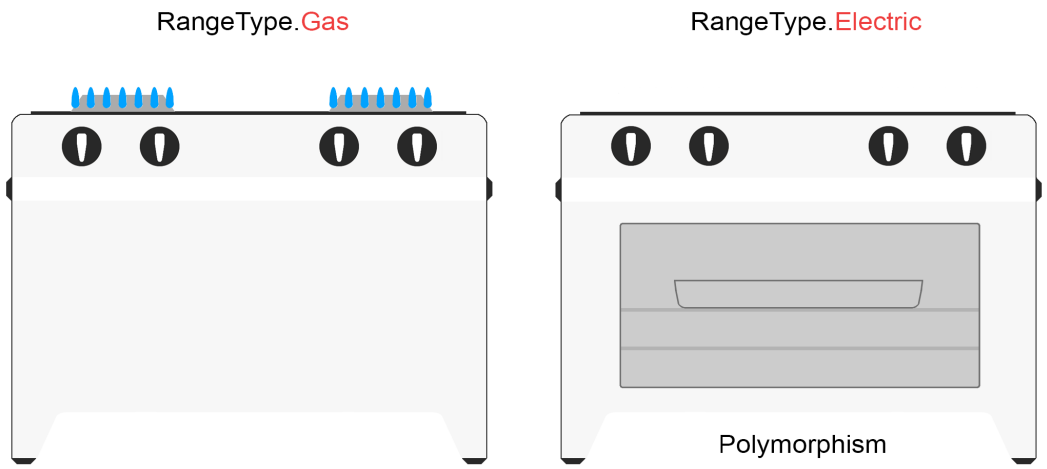

```
let burner = new Burner(BurnerIndex.UpperLeft);
let skillet = new Pan("cast iron");
burner.place(skillet); // place skillet on burner
burner.on();           // switch burner on
range.run(1);          // run range for 1 minute
```

Vessels loaded with ingredients can be placed on each of the four burners.

Once a burner is turned on, whatever is in the vessel placed on a burner whose state is "on", will be considered to be cooking, but only when the range is running for a period of time by calling: **Range.run(minutes)**.

18.5 Range Type and The Polymorphic Oven

RangeType.Gas and **RangeType.Electric** represent a different internal implementation for generating heat.



We won't go into deep detail on implementing gas or electric range, we'll simply write a heat generator function that converts an energy source into a heat unit. The main idea is that, even though **RangeType** implementation can change from **Gas** to **Electric** our range API code will still work without additional modification.

Polymorphism

Here **polymorphism** is demonstrated via **object composition** by integrating an **Oven** object directly into the **Range** object. Object Composition is when you combine two or more objects together to achieve polymorphism, instead of using **inheritance** like we've seen in **Vessel** example.

In code, object composition can manifest itself as new object instantiated as one of the properties of another object, usually in its constructor function.

For example in Range constructor: **this.oven = new Oven()** (instead of inheriting **Oven** from **Range**.)

18.6 Class Definitions

Programming in general can be split into two parts: defining things and using them. In this section, let's define every single class we need to demonstrate OOP principles. Comments will be provided where explanations become necessary.

Each class will be placed into a separate JavaScript file. For example **Range** will live in **./range.js** and **Ingredient** in **./ingredient.js**. The **import** and **export** keywords will be used to include them in your main application file.

After going over all class definitions, we'll implement them.

But first...

18.6.1 print.js

First, let me just say that console.log can be a bit cumbersome of a name. I renamed the function to simply print. This will be more aesthetically pleasant to read throughout remaining code.

```
001 // Overwrite console.log to print
002 const print = (message) => console.log(message);
003
004 export default print;
```

Yes, we are overwriting native **window.print** method, but in this particular application we don't have a use for it and it makes our code look more understandable.

18.6.2 Ingredient

```
001 export default class Ingredient {
002   constructor(name, type, calories) {
003     this.name = name;
004     this.type = type;
005     this.calories = calories;
006     this.minutes = {
007       fried: 0,
008       boiled: 0,
009       baked: 0
010     }
011   }
012   // Static types (use them like: Ingredient.fruit)
013   static meat = 0;
014   static vegetable = 1;
015   static fruit = 2;
016   static egg = 3;
017   static sauce = 4;
018   static grain = 5;
019   static cheese = 6;
020   static spice = 7;
021 };
```

18.6.3 FoodFactory

FoodFactory class will help us spawn a unique instance of the **Ingredient** object.

```
001 import print from "./print.js";
002 import Ingredient from "./ingredient.js";
003
004 export default class FoodFactory {
005   // Empty constructor;
006   // we won't be instantiating any objects here.
007   constructor() {}
008 };
009
010 // This function will produce a new instance of an object
011 FoodFactory.make = function(what) {
012   return new Ingredient(what.name, what.type, what.calories);
013 };
```

18.6.4 Fridge

Fridge is a pretty simple class.

It utilizes higher-order Array method **filter**:

```
001 export default class Fridge {
002   constructor(ingredients) {
003     this.items = ingredients;
004   }
005   // get all ingredients of type
006   get(type) {
007     return this.items.filter(i => i.type == type, 0);
008   }
009 };
```

You can load the refrigerator with an array of objects of type **Ingredient**. Want to grab just the vegetables? No problem! Just make the following call:

```
001 // Store all available ingredients in an array
002 const ingredients = [ water, olive_oil, broth, red_wine,
003 bay_leaf, peppercorn, beef, chicken, bacon, pineapple,
004 apple, blueberry, mushroom, carrot, potato, egg, cheese,
005 sauce, oatmeal, rice, brown_rice, cheese, sauce ];
006
007 // Create the refrigerator and store ingredients in it
008 let Frigidaire = new Fridge(ingredients);
009
010 // Get only vegetables from the fridge
011 let vegetables = Frigidaire.get(Ingredient.vegetable);
012
013 console.log(vegetables);
```

In console output, we get an array of only vegetables!

```
▼ (3) [Ingredient, Ingredient, Ingredient]
  ▶ 0: Ingredient {name: "mushroom", type: 2, calories: 4, minutes: {…}}
  ▶ 1: Ingredient {name: "carrot", type: 2, calories: 41.35, minutes: {…}}
  ▶ 2: Ingredient {name: "potato", type: 2, calories: 163, minutes: {…}}
    length: 3
> |
```

18.6.5 convert_energy_to_heat

Let's define our core energy to heat transfer function:

```
001 import print from "./print.js";
002 import { RangeType } from "./range.js";
003 // return one unit of energy based on stove implementation
004 const convert_energy_to_heat = function(source_type, minutes)
005 {
006   let energy = 0;
007   // Produces less energy due to longer heating up stage
008   if (source_type == RangeType.Electric) energy = 1;
009   // Procues more energy (more efficient)
010   if (source_type == RangeType.Gas) energy = 2;
011   let E = energy * minutes;
012   print(`Range generated ${E} unit(s) of energy.`);
013   // Return energy generated
014   return energy * minutes;
015 };
016
017 // short-hand
018 const convert = convert_energy_to_heat;
```

Guesswork

Here, energy is naively calculated on the premise that electric range is less efficient, so it will produce roughly 50% less percent energy than a gas-based range. This is all of course only speculative. If we had the actual hardware with us, we would modify this function to work with accurate numbers based on an actual model.

Separating Implementation

However, this function is important because it's at the core of range implementation. If you change the internals of how this function does what it does, our main class API will not be affected.

This means that in order to upgrade from **RangeType.Gas** to **RangeType.Electric** we won't have to rewrite *any* part of our code anywhere else!

This ability is offered to us if we follow Object Oriented Programming principles. Of course, it can be imitated using any other similar pattern or logical construct but, it is often encountered in the context of OOP.

18.6.6 Vessel

Vessel is the only class we have that demonstrates object inheritance.

We can start by defining its constructor:

```
003 // Define cooking vessel
004 export default class Vessel {
005   constructor(material, type) {
006     this.type = type;
007     this.material = material;
008     this.ingredients = [];
009     print(`Created ${this.material} ${this.type}.`);
010   }
```

The **calories** method will use a **reducer** to calculate the total number of calories currently in the vessel:

```
011   calories() {
012     const reducer = (acc, ing) => acc + ing.calories;
013     let sum = this.ingredients.reduce(reducer, 0);
014     print(`There are ${sum} calories in
015             ${this.material} ${this.type}.`);
016   }
```

The **add** method will add one ingredient to the vessel:

```
017   add(ingredient) {
018     this.ingredients.push(ingredient);
019     print(`Added ${ingredient.name} to
020             ${this.material} ${this.type}.`);
021   }
```

The **cook** method will cook contents of this vessel:

```
022   cook() {
023     console.log(`Cooking in ${this.material} ${this.type}`);
024   }
025 };// close export default class Vessel
```

Let's extend **Pan**, **Pot** and **Tray** from **Vessel**. Note that every time you extend an object from its parent, you must call **super** to invoke the super constructor of the parent object.

```
027 class Pan extends Vessel{
028   constructor(material) {
029     super(material, "frying pan");
030   }
031 };
032
033 class Pot extends Vessel {
034   constructor(material) {
035     super(material, "cooking pot");
036   }
037 };
038
039 class Tray extends Vessel {
040   constructor(material) {
041     super(material, "baking tray");
042   }
043 };
```

Inherited classes **Pan**, **Pot** and **Tray** inherit all default functionality from **Vessel** class. And we didn't add a single line of code to any of their own implementation!

Finally, export **Pan**, **Pot** and **Tray**. Note, we don't really have to export **Vessel** itself because this is our main abstract class that provides default implementation but it is never used directly anywhere.

```
045 export { Vessel, Pan, Pot, Tray };
```

18.6.7 Burner

And now the **Burner** class itself:

```
001 export const BurnerIndex = { UpperLeft: 0, UpperRight: 1,
002                              LowerLeft: 2, LowerRight: 3 };
003
004 export default class Burner {
005   constructor(id) {
006     this.id = id;
007     this.state = Burner.Off;
008     this.vessel = null; // link to what's on this burner.
009     print(`Created burner ${this.id} in Off state.`);
010   }
011   on() {
012     this.state = Burner.On;
013     print(`Burner ${this.id} turned on!`);
014   }
015   off() {
016     this.state = Burner.Off;
017     print(`Burner ${this.id} turned off!`);
018   }
019 };
```

We could have defined static properties **On** and **Off** inside the class itself using **static** keyword (like we did earlier with **Ingredients** class) but just to show that alternatives are possible (and you might see this in someone else's code) the static properties were defined directly on the **Burner** class, just outside of its body:

```
001 // Static flags
002 Burner.Off = false;
003 Burner.On = true;
```

18.6.8 Range

Polymorphic Oven

The Oven can be polymorphically added to Range. We'll see how in just a moment!

```
001 // Adds a new feature bake() (requires aluminum tray vessel)
002 class Oven {
003   constructor() {
004     this.tray = null;
005   }
006   add(tray) {
007     this.tray = tray;
008     print(`Tray with ${this.tray.ingredients} added!`);
009   }
010   on() {
011     print("Oven turned on!");
012   }
013   off() {
014     print("Oven turned off!");
015   }
016   bake() {
017     print("Oven bakes...");
018   }
019 };
```

Our Range class has some dependencies: **print**, **Burner**, **BurnerIndex** and **convert**, so let's import them from their respective source files.

```
001 import print from "./print.js";
002 import { Burner, BurnerIndex, convert } from "./burner.js";
003
004 const RangeType = {
005   Electric: "Electric",
006   Gas: "Gas"
007 }
```

On the next page we will start defining our **Range** class.

The **Range** class is the core to the whole cooking system. It ties all classes together into one cookware machine!

First, we'll start a new class **Range** and prepare it for exporting:

```
001 export default class Range {
```

Let's take a look at the constructor of **Range**:

```
003   constructor(type, stove) {
004
005     // By default oven is not installed
006     this.oven = null;
007
008     // RangeType.Electric, or RangeType.Gas
009     this.type = type;
010
011     // Placeholders for placing cookware on four burners
012     this.burners = [new Burner(BurnerIndex.UpperLeft),
013                     new Burner(BurnerIndex.UpperRight),
014                     new Burner(BurnerIndex.LowerLeft),
015                     new Burner(BurnerIndex.LowerRight)];
016
017     if (type == RangeType.Electric) {
018       // RangeType.ELECTRIC - takes longer to heat up, and
019       // cool down making it less efficient, and more
020       // difficult to control, more expensive to operate,
021       // with cheaper installation
022     }
023     if (type == RangeType.Gas) {
024       // RangeType.GAS - quicker to heat, with no cool down
025       // time making it more efficient and easier to control,
026       // cheaper to operate, with more expensive installation
027     }
028
029     console.log(`Created ${type} range!`);
030   }
```

In the constructor we add four new burners to our **Range** object. This is polymorphism (composing one object from other objects.) In this case our **Range** is composed of 4 separate **Burner** objects. Using inheritance would make less sense here. How can you "inherit" a range from burners, or burners from range? It makes little logical sense. Here a **Burner** is part of the **Range** object, which seems to more accurately relate to how things are composed in reality.

Range.install_oven()

We can install our polymorphic oven at a later time by calling **install_oven**:

```
032   install_oven() {
033     // Polymorphism in action: object composition
034     // We just added another key object to Range
035     this.oven = new Oven();
036     print("Installed oven!");
037   }
```

Range.place(index, vessel)

The **Range.place** method is used to place an existing **Vessel** object onto one of the burners at any of the four available locations.

```
039   // Place cookware vessel onto one of the 4 burners
040   place(index, vessel) {
041     this.burners[index].vessel = vessel;
042     print(`Placed ${vessel.material} ${vessel.type}
043             on range index ${index}.`);
044   }
```

Range.run(minutes)

Assuming our cookware vessels were placed onto one or more burners using **Range.place** method described above, we can now operate the range for a number of specified minutes by calling **Range.run(1);**

```
001   // Run the range converting energy to heat on each burner
002   run(minutes) {
003     // 1. convert energy to heat
004     this.burners.forEach(item => convert(this.type, minutes))
005     // 2. call cook() on the vessel
006     this.burners.forEach(item =>
007       // first check if a vessel exists on this burner
008       // if vessel is not null, so cook() in it
009       // otherwise do nothing
010       item.vessel ? item.cook() : null);
011   }
```

18.6.9 Putting It All Together

All of our classes are defined and we are ready to do some cooking!

First, let's add all dependencies to our main JavaScript file:

```
001 import Ingredient from "./ingredient.js";
002 import Fridge from "./fridge.js";
003 import { Burner, BurnerIndex } from "./burner.js";
004 import { Range, RangeType } from "./range.js";
005 import { Vessel, Pan, Pot, Tray } from "./cookware.js";
006 import FoodFactory from "./foodfactory.js";
```

One neat thing to notice here is we never have to import **print** function into our main program from **./print.js** file – it is completely internal to our range implementation.

But before we continue...

```
001 let Ing = Ingredient;
```

The word **Ingredient** is long. The statement for instantiating an object of this type didn't fit on one line of code in this book. You will see this in the source code on the next page. So I simply created a reference to it and called it **Ing**.

Remember how variable names are *references* to the original object. This means our **Ing** variable is a reference to exactly the same constructor as **Ingredient**. No copies were made.

Let's head to the next page where we will create actual ingredients as instances of the class **Ingredient**.

The **Ingredient** constructor takes: **name**, **type** and **calories**. I looked up calories per serving for each ingredient on Google.

Define ingredients

```
// water
const water       = new Ing("water", Ing.liquid, 0);
const olive_oil   = new Ing("olive oil", Ing.liquid, 0);
const broth       = new Ing("broth", Ing.liquid, 11);
const red_wine    = new Ing("red wine", Ing.liquid, 125);
// spices
const bay_leaf    = new Ing("bay leaf", Ing.spice, 6);
const peppercorn  = new Ing("peppercorn", Ing.spice, 17);
// meats
const beef        = new Ing("beef", Ing.meat, 213);
const chicken     = new Ing("chicken", Ing.meat, 335);
const bacon       = new Ing("bacon", Ing.meat, 43);
// fruits
const pineapple   = new Ing("pineapple", Ing.fruit, 452);
const apple       = new Ing("apple", Ing.fruit, 95);
const blueberry   = new Ing("blueberry", Ing.fruit, 85);
// vegetables
const mushroom    = new Ing("mushroom", Ing.vegetable, 4);
const carrot      = new Ing("carrot", Ing.vegetable, 41.35);
const potato      = new Ing("potato", Ing.vegetable, 163);
const pepper      = new Ing("bell pepper", Ing.vegetable, 24);
const onion       = new Ing("onion", Ing.vegetable, 44);
// eggs
const egg         = new Ing("egg", Ing.egg, 78);
// grain, other
const oatmeal     = new Ing("oatmeal", Ing.grain, 158);
const rice        = new Ing("rice", Ing.grain, 206);
const brown_rice  = new Ing("brown rice", Ing.grain, 216);
const cheese      = new Ing("cheese", Ing.cheese, 113);
const sauce       = new Ing("tomato sauce", Ing.sauce, 70);
```

Instantiate the range object

```
// Create the electric range
let range = new Range(RangeType.Electric);
```

Next time we run **range.run(1)** we can cook everything that was placed on the stove for 1 minute. But until that can happen we first need to create some pans and pots, and put some ingredients in them.

Now that we created the range, let's created our cooking vessels!

```
// Create cooking vessels
let pan = new Pan("cast iron");
let skillet = new Pan("cast iron");
let pot = new Pot("stainless steel");
let tray = new Tray("aluminum");
```

Let's drop some ingredients into our **pot** in preparation for making a beef stew:

```
// Make beef stew
pot.add(water);
pot.add(broth);
pot.add(red_wine);
pot.add(beef);
pot.add(potato);
pot.add(carrot);
pot.add(bay_leaf);
pot.add(peppercorn);
pot.calories(); // 576.35 calories
```

But there is a problem, JavaScript treats variables as references to the same object. So if we add water, broth, beef, etc... to another skillet, essentially we are adding the same ingredients we already placed in the pot object above.

This means if we turned on the range, we might be cooking *the same* ingredients in different cookware vessels! This is where a factory class comes in.

The **FoodFactory** class will return a new instance of the ingredient, instead of a reference to the existing ingredient object instance: (see FoodFactory class implementation to understand how it does it.)

```
skillet.add(FoodFactory.make(mushroom));
skillet.add(FoodFactory.make(carrot));
skillet.add(FoodFactory.make(onion));
skillet.add(FoodFactory.make(potato));
skillet.add(FoodFactory.make(pepper));
skillet.calories(); // 2260
```

Using **FoodFactory** to make sure all ingredients are unique object instances:

```
// Fry 3 eggs
pan.add(FoodFactory.make(egg));
pan.add(FoodFactory.make(egg));
pan.add(FoodFactory.make(egg));
pan.calories(); // 234 calories
```

Now, let's place all pots and pans on the burners:

```
// Place the pot with beef stew on LowerLeft burner
range.place(BurnerIndex.LowerLeft, pot);

// Place skillet with vegetable stew onto LowerRight burner
range.place(BurnerIndex.LowerRight, skillet);

// Place frying pan with 3 eggs onto UpperRight burner
range.place(BurnerIndex.UpperRight, pan);
```

And turn them on:

```
// Turn burners on
range.burners[BurnerIndex.LowerLeft].on();
range.burners[BurnerIndex.LowerRight].on();
range.burners[BurnerIndex.UpperRight].on();
```

Finally, let the range run for 1 minute:

```
// Cook everything on all turned on burners for 1 minute
range.run(1);
```

Chapter 19

Event Loop

As a JavaScript programmer, you don't need to understand actual implementation of the event loop. But understanding how the Event Loop works is important for at least two reasons:

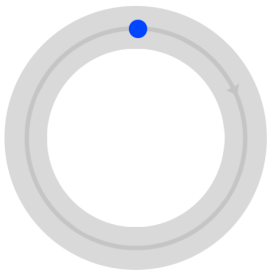
First, questions related to the event loop are often asked at job interviews.

The second reason is a bit more practical. By developing an awareness of how it works, you will be able to understand the order of events as they occur, for example, when working with callbacks and timers such as **setTimeout** function.

In the previous section we've seen how when functions are executed they are placed on the Call Stack. We can even use the stack trace to track which function was originally called to produce an error. This makes sense when working with a deterministic set of events (one statement will proceed to be executed immediately after the previous one is finished executing.)

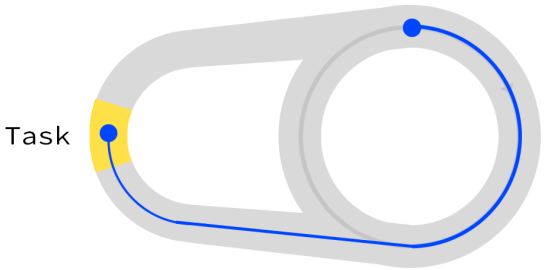
But writing JavaScript code is often based on listening to events: timers, mouse clicks, HTTP requests, etc. Events assume there will be a period of waiting time until they return after accomplishing whatever task they were set to perform. But while events are doing their work, we don't want to halt our main program.

For this reason, whenever an event occurs, it is handed over to the Event Loop.

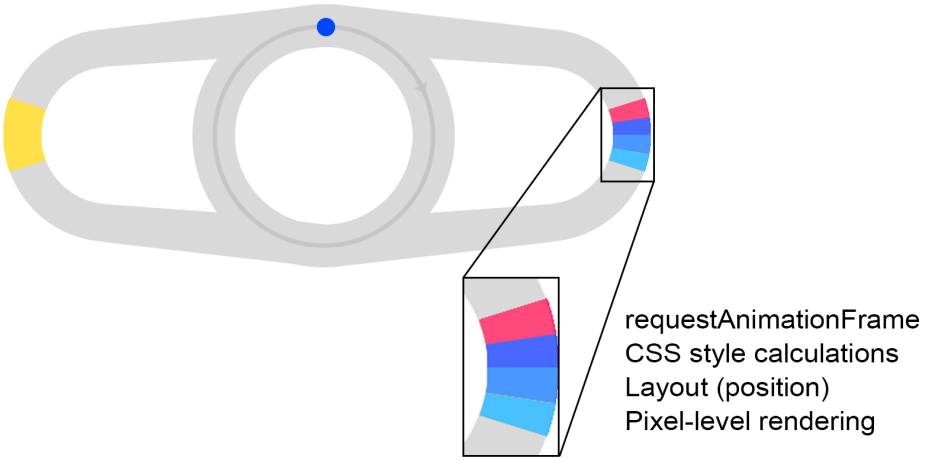


In abstract terms, the Event Loop can be thought of as just what it sounds like. It's a loop, with a process that keeps running in circles.

Whenever an event occurs, which can be thought of as a task, it is delegated to the Event Loop, which "goes out of its way" to pick up the task.



But it's not that simple. The event loop handles events such as mouse clicks, and timeouts. But it also needs to take care of updating the browser view:



The process in the event loop will continue to make rounds, sometimes spending time processing tasks or updating the view.

The whole experience of how users interact with your front-end application will often depend on how optimized your code is for the event loop.

To create smooth user experience, your code should be written in such way, that balances task processing with screen updates.

In modern browsers updating the view usually consists of 4 steps: checking for **requestAnimationFrame**, CSS style calculations, determining layout position, and rendering the view (actually drawing the pixels.)

Choosing the right time to update the browser is tricky. After all **setTimeout** or **setInterval** were never meant to be used together with rendering the browser view. In fact if you've tried to use them to animate elements, you may have experienced choppy performance.

This is because **setInterval** hijacks the event loop, by executing the callback (in which many place their animation code) as fast as possible. For this reason many have moved animations that can be done in CSS to their respective CSS style definitions, instead of performing them in JavaScript.

However, choppy performance can be fixed with **requestAnimationFrame**. What happens is that event loop will actually sync to your monitor's refresh rate, rather than execute each time **setInterval** fires its callback function.

Chapter 20

Call Stack

The **call stack** is a place to keep track of currently executing functions. As your code executes, each call is placed on the call stack in order in which it appears in your program. Once the function returns it is removed from the call stack.

Placing a function call onto the stack is called **pushing** and removing it from the call stack is called **popping**. Same idea behind `Array.push` and `.pop` methods.

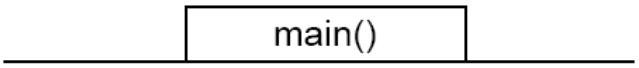


Figure 20.1: The main entry point is **pushed** on the stack.

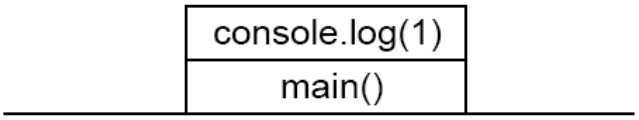


Figure 20.2: Every time **console.log** is called, it's **pushed** to the call stack.

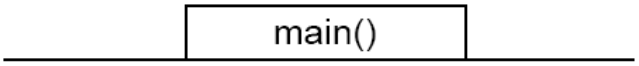


Figure 20.3: The console.log prints 1 to the console and returns. It is then **popped** from the stack. Main function continues to run until it returns. This usually will happen when browser is closed.

How does this apply to writing code?

The **call stack** is a fundamental building block of computer language design. Most languages implement a call stack in one way or another. But how does this apply to those who are simply writing code and not designing computer languages?

Call Stack Example

Complex tasks have priorities. Many things require to be done in a logical order.

When writing software you will often call one function from the body of another function. You can't mop the floor until you fill the bucket with water. You have no reason to fill the bucket with water until you *decide* to clean the house first:

```
001 function mop_floor() {
002   console.log("Mop the floor.");
003   throw new Error("Ran out of water!");
004 }
005
006 function fill_bucket(what) {
007   console.log("Filling bucket with " + what);
008   mop_floor();
009 }
010
011 function clean_house() {
012   console.log("Cleaning house.");
013   fill_bucket("water");
014 }
015
016 clean_house();
```

Figure 20.4: Calling **clean_house** triggers a chain of function calls.

Our last function **mop_floor** throws an error. When this happens a stack trace is shown in the console.

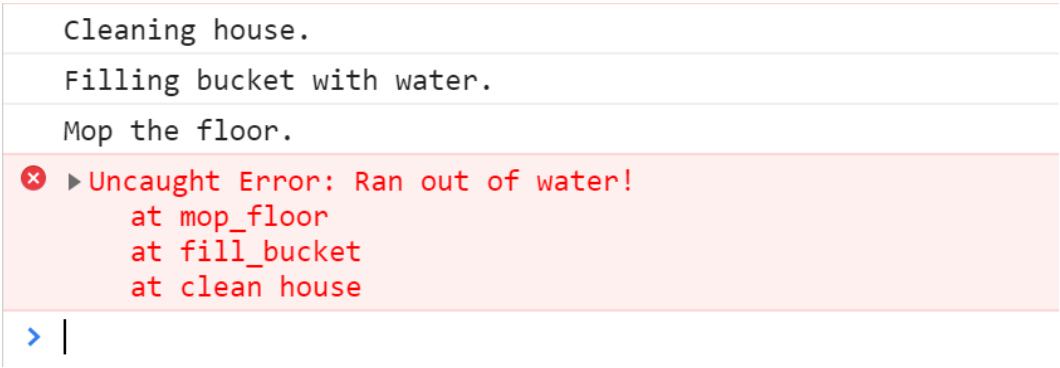


Figure 20.5: Call stack helps us identify the root source of the error.

Error displays the trace of the call stack history, starting with the most recently called function **mop_floor**, in which the error occurred. When debugging this help us trace the error all the way back to the original function **clean_house**.

Most of the time you won't be concerned with thinking about them when writing code. But you might need to understand them when debugging complex large scale software.

20.1 Execution Context

The **call stack** *is* a stack of execution contexts. When discussing one we will inevitably run into the other.

You don't need to understand **execution context** or the **call stack** in great detail to write JavaScript code. But it might help understand the language better.

What Is Execution Context?

As your program continues to run, the statements being executed exist alongside something called an **execution context**. Note that the execution context is pointed to by **this** keyword in each scope. Not only function-scope either. Block-scope also carries with it a link to execution context via **this** keyword.

This often creates confusion, because in JavaScript the **this** keyword is also used as a reference to an *instance* of an object in class definitions, so that we can access

its member properties and members.

But things become clear if we understand that **execution context** *is* represented by an instance of an object. It is just not used to access its properties or methods. Instead, it establishes a link between sections of code flow across multiple scopes.

Root Execution Context

When your program opens in a browser, an instance of a **window** object is created automatically. This **window** object becomes the root execution context, because it's the first object instantiated by the browser's JavaScript engine itself. The **window** object is the execution context of **global scope** – they refer to the same thing. The `window` object is an instance of `Window` class.

So how does it work?

If you call a function from global scope, the **this** keyword *inside* the function's scope will point to **window** object – the context from which the function was called. The context was carried over into the function's scope.

It's like a link was established from current execution context to the previous one.

The execution context is something that is carried over from one scope to another, during code execution flow throughout lifetime of your program. You can think of it as a tree branch that extends into another scope from the root **window** object.

In the remaining sections of this chapter we will take a look at one possible interpretation of **execution context** and the **call stack**.

20.2 Execution Context In Code

There is a difference between the logic of call stack and execution contexts and how it manifests itself to the programmer. Obviously, being aware of the call stack isn't required when writing code. In JavaScript, the closest you will get to working with execution contexts is via the **this** keyword.

Execution context is held by **this** keyword in each scope. The name *context* suggests that it can change. This is true. The **this** keyword in each scope may change or point to another new object in various situations.

But where does it all begin?

20.2.1 Window / Global Scope

When the window object is created, we get a handful of things happening under the hood. A new **lexical environment** is created: it contains variable environment for that scope – a place in memory for storing your local variables. Around this time the first ever **this** binding takes place.

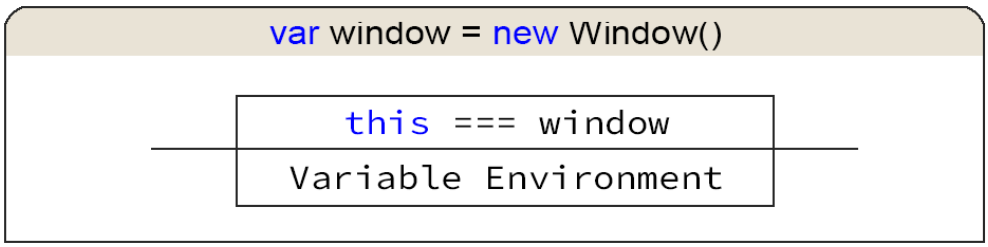


Figure 20.6: A new execution context is created when main window object is instantiated. The **this** keyword points to the window object.

In global scope **this** keyword points to **window** object.

20.2.2 The Call Stack

The **call stack** keeps track of function calls. If you call a function from context of the global scope, a new entry will be placed "on top" of the current context. The newly created stack will inherit execution context from the previous environment.

To visualize this, let's take a look at this diagram:

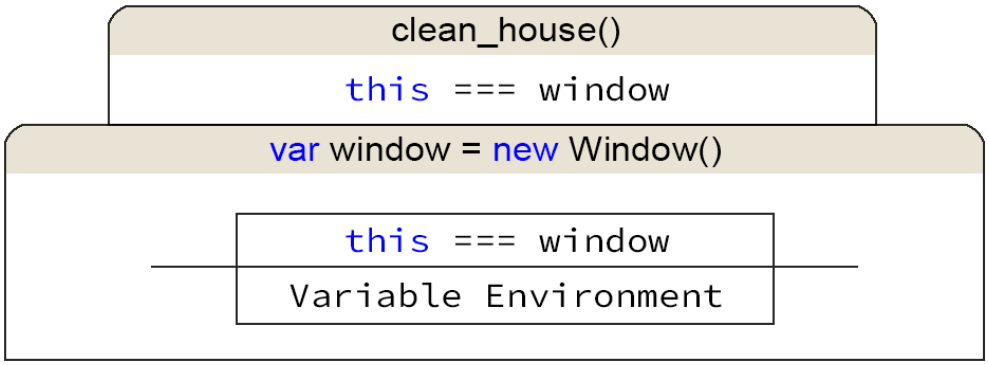


Figure 20.7: Binding of **this** object across **execution contexts** on the **call stack**. A new stack is created when function is called. This new context is logically placed "on top" of the previous object on the call stack.

Call Stack & Execution Context Chain

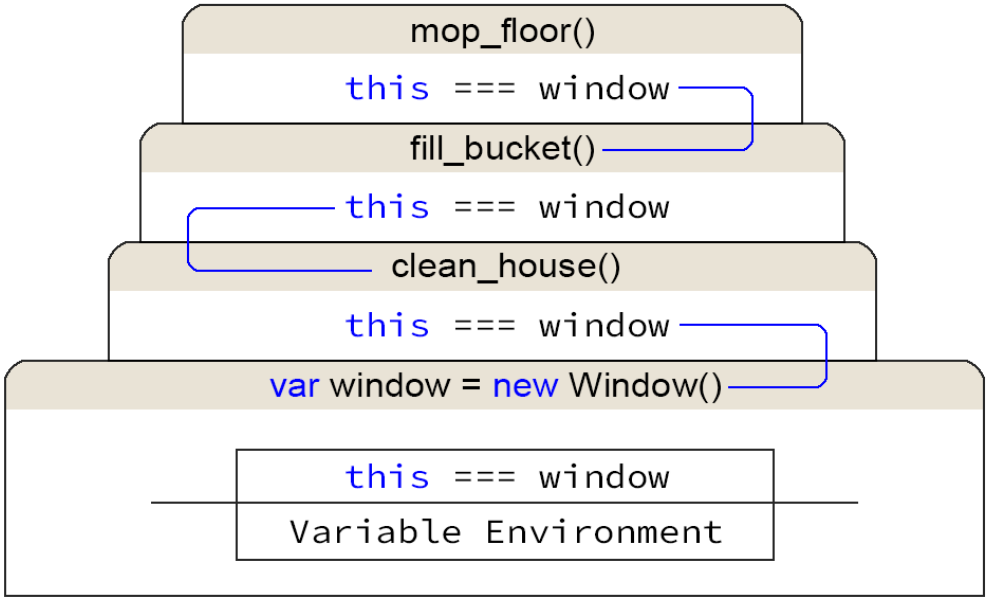


Figure 20.8: As more functions that depend on each other are called from within other functions, the stack grows.

As you can see the context carries over to the newly created stack and remains accessible via the **this** keyword. This process repeats while maintaining a chain of execution contexts all the way up to the currently executing context:

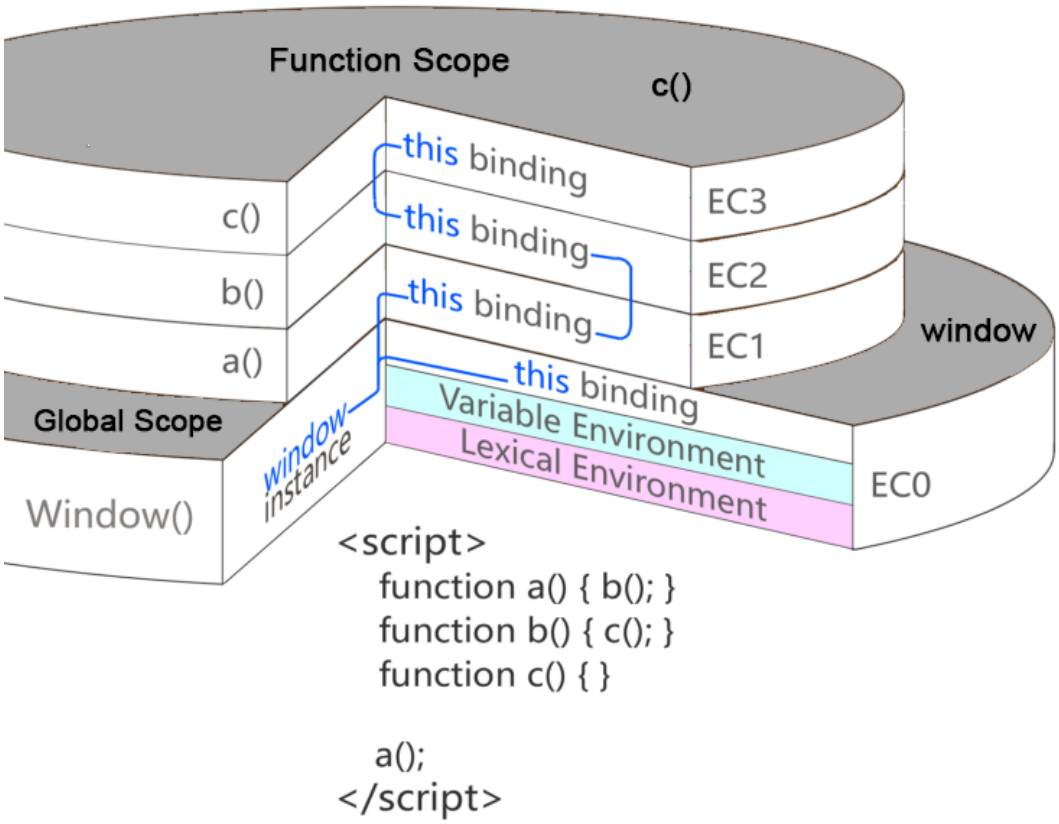


Figure 20.9: Calling multiple functions from within each other's scope will build a tower of function calls on the call stack. Note that this would happen only one at a time for each function if all functions were called from global scope context.

Note that each function carries with it its own execution context EC0 – EC3.

There is always one *current executing context*. This is the context on the very top of the stack. While all of the previous stacks remain below, until execution returns from the current context (*function ran it's course and returned, so JavaScript removes the context from memory and we no longer need it...*)

After the function is finished executing the stack is removed from the top and the code flow returns to the remaining previous / uppermost execution context. Contexts are constantly pushed and popped from the call stack.

Stacking only occurs if you call a function from another function. It won't happen if all functions are executed consequently from the *same* execution context.

In which case... you will have one function pushed to the stack, popped from the stack, and then the next function will be pushed onto the empty stack again... and so on.

20.2.3 .call(), .bind(), .apply()

These three functions can be used to call a function and choose what **this** keyword should point to within the scope of that function, overriding its default behavior.

20.2.4 Stack Overflow

When you pour your favorite soda into a tall glass, it will lose its carbonation and given enough **amount** and **pace** it will fizz over the rim.

You can think of **stack overflow** in a similar way. The glass is the call stack's memory address space. The foam above rim is memory that could not be allocated.

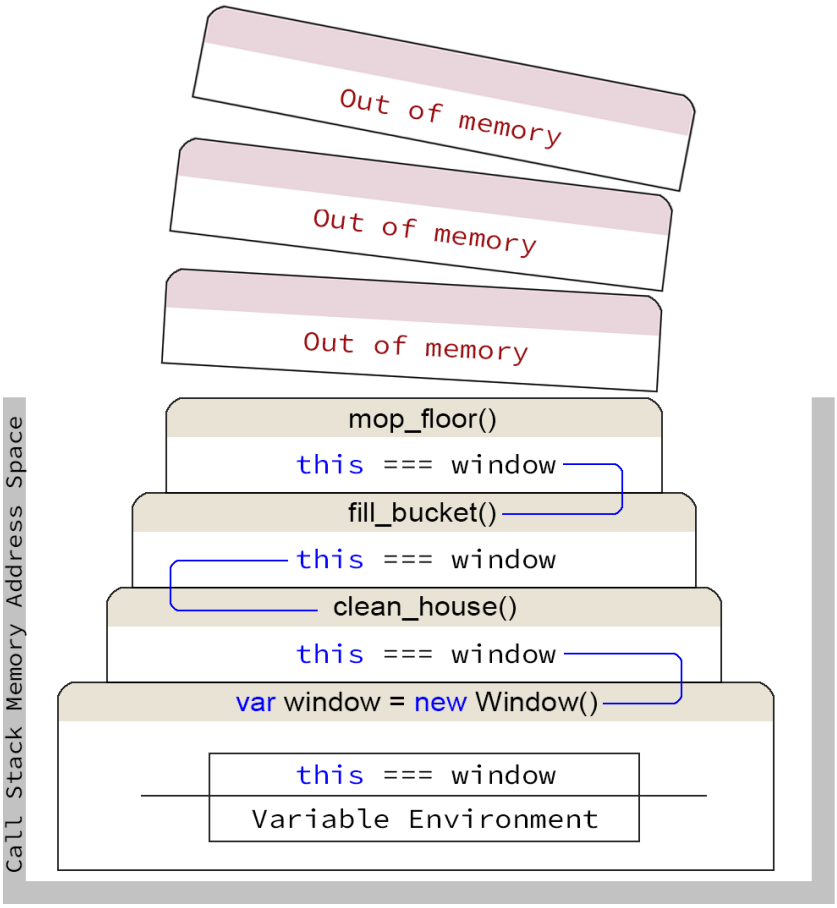


Figure 20.10: Every time a function is called, new context is created in memory. But memory is not infinite. **Stack Overflow** occurs when the memory required to build the call stack exceeds the address space allocated for the stack. This amount is determined and managed internally by the browser.