

Advanced JavaScript

A reference guide for Advanced JavaScript Concepts

*This guide is incomplete! Please **contact me** to add knowledge and ask questions. Sorry, but comments have been disabled, a few too many accidental edits/comments.*

[A Map for the Advanced JavaScript Concepts Course](#)

[Acknowledgements](#)

Compiled by [Bryan Windsor](#).   

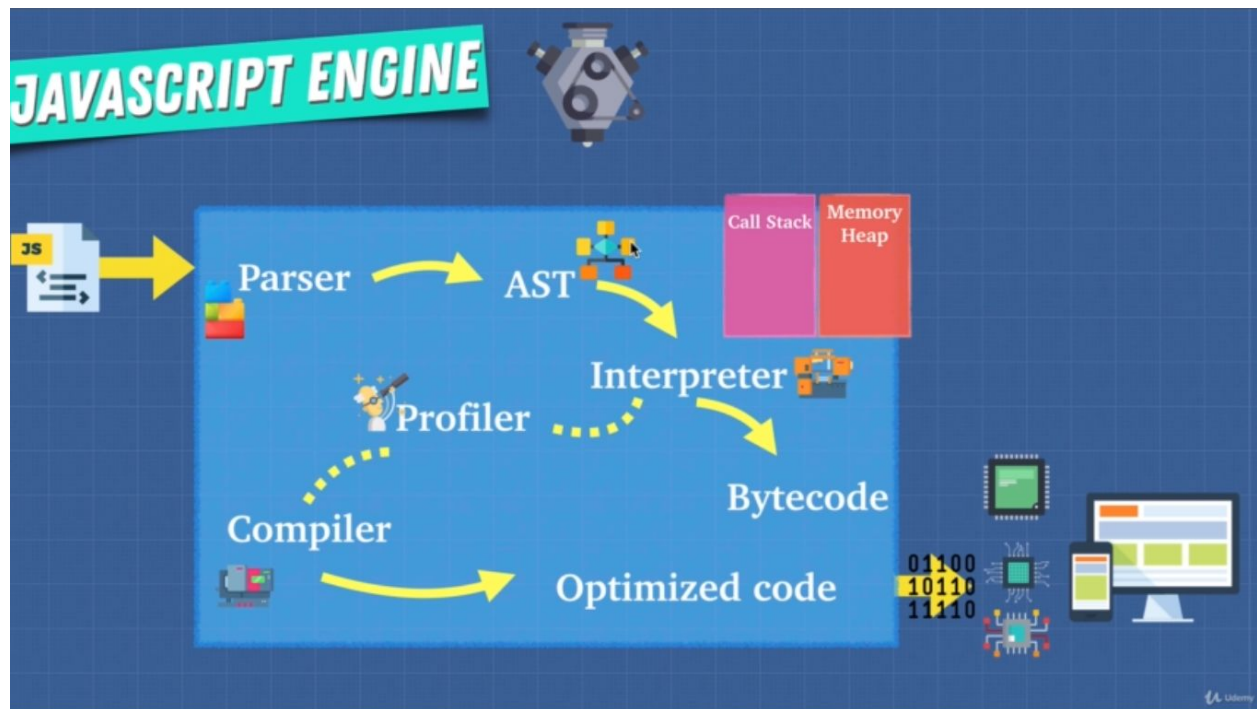
JavaScript Foundation

Section 1

JavaScript Engine

- An **engine** is responsible for providing the mechanics of parsing and JIT compilation, i.e. producing machine-executable operations from a script written in JavaScript.
- Translates JS into something a computer can understand
- Many Engines:
 - All JS Engines should comply with ECMAScript standards
 - Examples:
 - V8 (Chrome)
 - Chakra (MS Edge)
 - Nitro (Safari)
 - SpiderMonkey (Firefox)
- JS Creator and first JS Engine author
 - Brendan Eich

How the JS engine works



- **Parser**
 - Lexical Analysis
 - Code is parsed into “tokens”
- **AST**
 - Tokens form an “Abstract Syntax Tree” (AST)
 - [Practical AST Examples](#)
- **Bytecode**
 - Not as low-level as machine code, but a lower level than JS
- **Profiler/Monitor**
 - Watches code and optimizes
 - If it sees an opportunity to optimize, e.g. a loop with same input/output, it passes the code to the JIT compiler
- **Compiler**
 - Creates optimized machine code
- **Single-threaded**
 - The call stack only executes one function at a time
 - We circumvent the limitations of ‘single-threadedness’ with the Web API
- **Gotchas**
 - Hidden classes
 - When instantiating new objects, the compiler will try to create a common ‘hidden class’. By defining properties in different orders, e.g.

```
function Animal(x, y) {
  this.x = x;
  this.y = y;
}
```

```

}
const obj1 = new Animal(1,2)
const obj2 = new Animal(3,4)
// Here the objects are given new properties, but in different orders
obj1.a = 1
obj1.b = 2
obj2.b = 2
obj2.a = 1

```

the compiler will de-optimize the code

- When compiling, the compiler tries to optimize the code, for example, by creating "**hidden classes**":
<https://engineering.linecorp.com/en/blog/v8-hidden-class/>
- There are cases where the code can be optimized by the compiler by *sharing "hidden classes"*, but for some reason, such as a *different order of object property creation*, the compiler mistakenly thinks that two objects, which should be able to share "**hidden classes**", cannot. The compiler is thus creating an unnecessary inefficiency which we call "*de-optimizing the code*".
- This means that we should either set all possible properties in the constructor of a class
- Or we can be careful to always define new properties in the same order

JavaScript Runtime

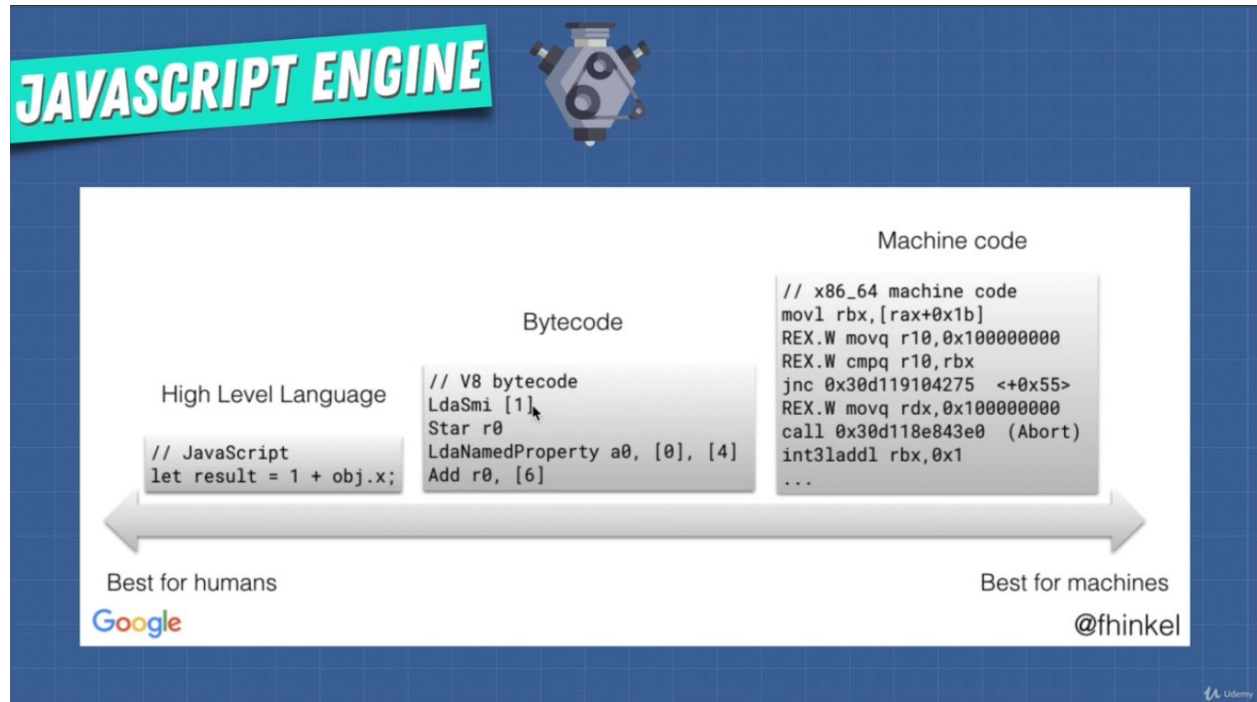
- The **runtime** environment *provides the built-in libraries that are available to the program at runtime* (during execution). So, if you're going to use the Window object or the DOM API in the browser, those would be included in the browser's JS runtime environment. A Node.js runtime includes different libraries, say, the Cluster and FileSystem APIs. Both runtimes include the built-in data types and common facilities such as the Console object.

JavaScript Engine vs. Runtime

- The distinction between the two is not always clear and you'll find that the terms are commonly used interchangeably.
 - Chrome and Node.js share the *same engine* (Google's V8), but they have *different runtime* (execution) environments.
 - In a way, the **runtime** is to the **engine** what the linker is to the compiler in a traditional compiled language.
- [<https://www.quora.com/What-is-the-difference-between-javascript-engine-and-javascript-runtime>]

Interpreter/Compiler/JIT Compiler

Shows how JavaScript is translated into a lower-level language



Interpreter vs Compiler

- Interpreter
 - Line by line
 - Faster to start
- Compiler
 - Reads entire file and translates to a lower level language
 - Once begun, is faster than interpreted by optimizing into a lower level language
 - Not perfect, can accidentally de-optimize code

JIT Compiler

"Just-In-Time" Compiler

As a way of getting rid of the interpreter's inefficiency—where the interpreter has to keep retranslating the code every time they go through the loop—browsers started mixing compilers in.

Different browsers do this in slightly different ways, but the basic idea is the same. They added a new part to the JavaScript engine, called a monitor (aka a profiler). That monitor watches the code as it runs, and makes a note of how many times it is run and what types are used.

At first, the monitor just runs everything through the interpreter.

If the same lines of code are run a few times, that segment of code is called warm. If it's run a lot, then it's called hot.

Baseline compiler

When a function starts getting warm, the JIT will send it off to be compiled. Then it will store that compilation.

Each line of the function is compiled to a “stub”. The stubs are indexed by line number and variable type (I'll explain why that's important later). If the monitor sees that execution is hitting the same code again with the same variable types, it will just pull out its compiled version.

That helps speed things up. But like I said, there's more a compiler can do. It can take some time to figure out the most efficient way to do things... to make optimizations.

The baseline compiler will make some of these optimizations (I give an example of one below).

It doesn't want to take too much time, though, because it doesn't want to hold up execution too long.

However, if the code is really hot—if it's being run a whole bunch of times—then it's worth taking the extra time to make more optimizations.

Optimizing compiler

When a part of the code is very hot, the monitor will send it off to the optimizing compiler. This will create another, even faster, version of the function that will also be stored.

In order to make a faster version of the code, the optimizing compiler has to make some assumptions.

For example, if it can assume that all objects created by a particular constructor have the same shape—that is, that they always have the same property names, and that those properties were added in the same order— then it can cut some corners based on that.

The optimizing compiler uses the information the monitor has gathered by watching code execution to make these judgments. If something has been true for all previous passes through a loop, it assumes it will continue to be true.

But of course with JavaScript, there are never any guarantees. You could have 99 objects that all have the same shape, but then the 100th might be missing a property.

So the compiled code needs to check before it runs to see whether the assumptions are valid. If they are, then the compiled code runs. But if not, the JIT assumes that it made the wrong assumptions and trashes the optimized code.

Then execution goes back to the interpreter or baseline compiled version. This process is called deoptimization (or bailing out).

Usually optimizing compilers make code faster, but sometimes they can cause unexpected performance problems. If you have code that keeps getting optimized and then deoptimized, it ends up being slower than just executing the baseline compiled version.

Most browsers have added limits to break out of these optimization/deoptimization cycles when they happen. If the JIT has made more than, say, 10 attempts at optimizing and keeps having to throw it out, it will just stop trying.

An example optimization: Type specialization

There are a lot of different kinds of optimizations, but I want to take a look at one type so you can get a feel for how optimization happens. One of the biggest wins in optimizing compilers comes from something called type specialization.

The dynamic type system that JavaScript uses requires a little bit of extra work at runtime. For example, consider this code:

```
function arraySum(arr) {  
  var sum = 0;  
  for (var i = 0; i < arr.length; i++) {  
    sum += arr[i];  
  }  
}
```

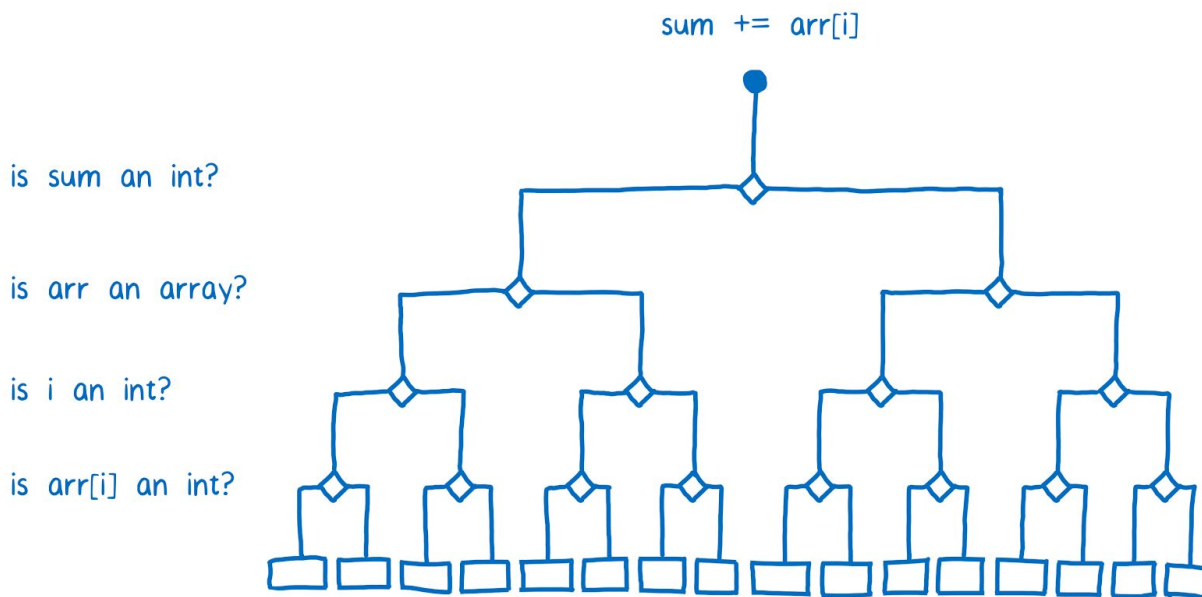
The `+=` step in the loop may seem simple. It may seem like you can compute this in one step, but because of dynamic typing, it takes more steps than you would expect.

Let's assume that `arr` is an array of 100 integers. Once the code warms up, the baseline compiler will create a stub for each operation in the function. So there will be a stub for `sum += arr[i]`, which will handle the `+=` operation as integer addition.

However, `sum` and `arr[i]` aren't guaranteed to be integers. Because types are dynamic in JavaScript, there's a chance that in a later iteration of the loop, `arr[i]` will be a string. Integer addition and string concatenation are two very different operations, so they would compile to very different machine code.

The way the JIT handles this is by compiling multiple baseline stubs. If a piece of code is monomorphic (that is, always called with the same types) it will get one stub. If it is polymorphic (called with different types from one pass through the code to another), then it will get a stub for each combination of types that has come through that operation.

This means that the JIT has to ask a lot of questions before it chooses a stub.



Because each line of code has its own set of stubs in the baseline compiler, the JIT needs to keep checking the types each time the line of code is executed. So for each iteration through the loop, it will have to ask the same questions.

The code would execute a lot faster if the JIT didn't need to repeat those checks. And that's one of the things the optimizing compiler does.

In the optimizing compiler, the whole function is compiled together. The type checks are moved so that they happen before the loop.

Some JITs optimize this even further. For example, in Firefox there's a special classification for arrays that only contain integers. If `arr` is one of these arrays, then the JIT doesn't need to check if `arr[i]` is an integer. This means that the JIT can do all of the type checks before it enters the loop.

Conclusion

That is the JIT in a nutshell. It makes JavaScript run faster by monitoring the code as it's running it and sending hot code paths to be optimized. This has resulted in many-fold performance improvements for most JavaScript applications.

Even with these improvements, though, the performance of JavaScript can be unpredictable. And to make things faster, the JIT has added some overhead during runtime, including:

- optimization and deoptimization
- memory used for the monitor's bookkeeping and recovery information for when bailouts happen
- memory used to store baseline and optimized versions of a function

There's room for improvement here: that overhead could be removed, making performance more predictable. And that's one of the things that WebAssembly does.

In the [next article](#), I'll explain more about assembly and how compilers work with it.

[by [Lin Clark](#) at <https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/>]

Writing Optimized Code

- Initialize objects of the same class in the same order
-

Call Stack + Memory Heap

- Call Stack
 - The call stack stores function calls
 - Ensures the program runs in order
 - The first stack frame (on top) is program's current 'location'
 - First In, Last Out
 - **Global Execution Context** is called and is at bottom of **Call Stack**
 - First function is called and is added to top of **Call Stack**
 - First function 'returns' and it is popped off of the **Call Stack**
 - Repeat until program completes and Global Execution Context pops off the Call Stack
 - Stack Overflow is when there are too many stack frames, e.g.

```
function inception() {  
    inception()  
}
```


- Memory Heap
 - Stores values and references

```
const number = 6; //allocate memory for number
const string = 'hello'; //allocate memory for a string
const human = { //allocate memory for an object and its values
  first: 'Bryan',
  last: 'James'
}
```

Garbage Collection

- Garbage collection refers to a process that automatically frees up memory as it is able
- Garbage collection helps us avoid memory leaks
- Memory leaks are a failure in a program to release discarded memory, causing impaired performance or failure.
 - Global variables
 - Event listeners
 - For the case of observers, it is important to make explicit calls to remove them once they are not needed anymore (or the associated object is about to be made unreachable). In the past, this used to be particularly important as certain browsers (Internet Explorer 6) were not able to manage cyclic references well (see below for more info on that). Nowadays, most browsers can and will collect observer handlers once the observed object becomes unreachable, even if the listener is not explicitly removed. It remains good practice, however, to explicitly remove these observers before the object is disposed
(<https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/>)

```
var element = document.getElementById('button');

function onClick(event) {
  element.innerHTML = 'text';
}

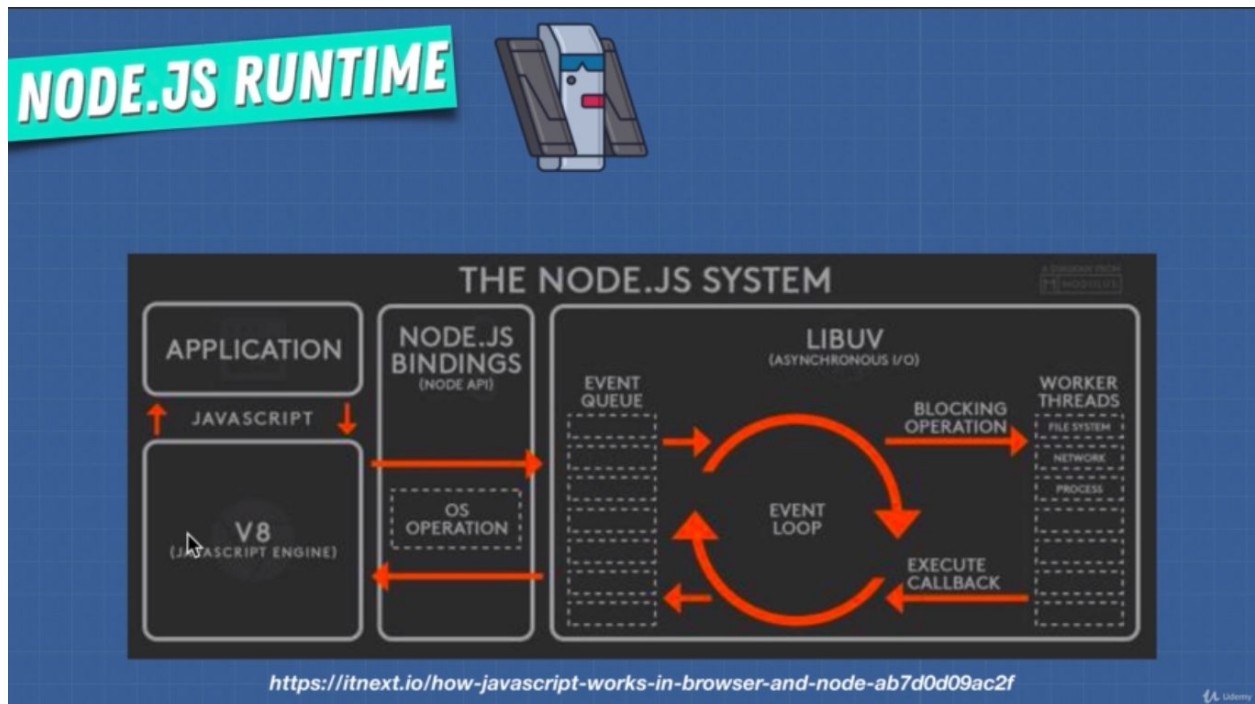
element.addEventListener('click', onClick);
// Do stuff
element.removeEventListener('click', onClick);
element.parentNode.removeChild(element);
// Now when element goes out of scope,
// both element and onClick will be collected even in old browsers that
```

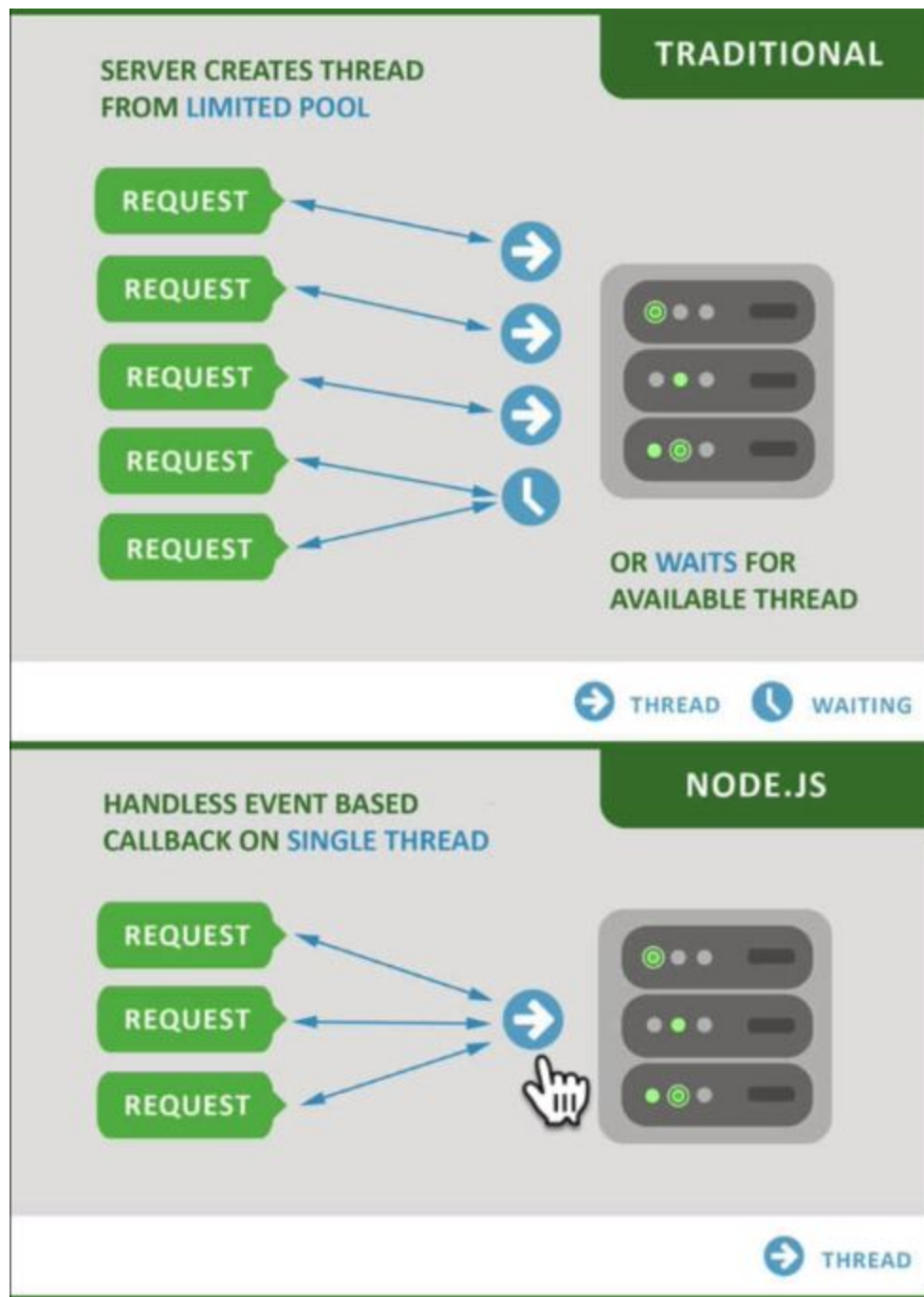
don't handle cycles well.

- Mark and Sweep
 - JS can mark what references and values in memory are still needed
 - After marking, JS then **sweeps** out the unneeded memory

Node.js

- Node.js is a JS runtime
- It is written in C++





Single Threaded Model

- JS is single-threaded

- The call stack only executes one function at a time
 - We circumvent the limitations of 'single-threadedness' with the Web API
-

Section 2

Execution Context

- The environment in which the current code is being evaluated in. There can only be one execution context running at any point of time! This is because Javascript is single-threaded.
 - **arguments** is an object that is made available to **execution contexts** made with the **function** keyword
 - Because **arguments** is an object, **Array** methods are not available
 - We can use **Array.from(arguments)** to create an array with the **arguments** values
 - The **spread** operator can be used to convert **arguments** in a **function** into an **Array**
-

Lexical Environment

- Lexical environments means where the code was written.
 - The Global Environment is the parent environment to all other environments created in the code
 - In the browser, the global environment is called **window**
 - In Node.js, the global environment is called **global**
-

Hoisting

Hoisting is a term you will *not* find used in any normative specification prose prior to [ECMAScript® 2015 Language Specification](#). **Hoisting** was thought up as a general way of thinking about how execution contexts (specifically the creation and execution phases) work in JavaScript. However, the concept can be a little confusing at first.

Conceptually, for example, a strict definition of **hoisting** suggests that **variable** and **function declarations** are physically moved to the top of your code, but this is not in fact what happens. Instead, the variable and function declarations are put into memory during the *compile* phase, but stay exactly where you typed them in your code.

One of the advantages of JavaScript putting **function declarations** into memory before it executes any code segment is that *it allows you to use a **function** before you declare it in your code*. For example:

```
function catName(name) {  
  console.log("My cat's name is " + name);  
}  
catName("Tigger"); /* The result of the code above is: "My cat's name is  
Tigger" */
```

The above code snippet is how you would expect to write the code for it to work. Now, let's see what happens when we call the **function** before we write it:

```
catName("Chloe");  
function catName(name) {  
  console.log("My cat's name is " + name);  
} /* The result of the code above is: "My cat's name is Chloe" */
```

Even though we call the **function** in our code first, *before the **function** is written*, the code still works. This is because of how context execution works in JavaScript.

Hoisting works well with other data types and variables. The variables can be initialized and used before they are declared.

*JavaScript only **hoists** declarations, not initializations*. If a variable is declared and initialized after using it, the value will be undefined. For example:

```
console.log(num); // Returns undefined var num; num = 6;
```

If you declare the variable after it is used, but initialize it beforehand, it will return the value:

```
num = 6; console.log(num); // returns 6 var num;
```

Below are more examples demonstrating hoisting.

```
//Example 1 - Does not hoist
var x = 1; // Initialize x
console.log(x + " " + y); // '1 undefined'
var y = 2; // Initialize y
//This will not work as JavaScript only hoists declarations
//Example 2 - Hoists
var num1 = 3; //Declare and initialize num1
num2 = 4; //Initialize num2
console.log(num1 + " " + num2); //'3 4'
var num2; //Declare num2 for hoisting
//Example 3 - Hoists
a = 'Cran'; //Initialize a
b = 'berry'; //Initialize b
console.log(a + " " + b); // 'Cranberry'
var a, b; //Declare both a & b for hoisting
```

<https://developer.mozilla.org/en-US/docs/Glossary/Hoisting>

Hoisting with const, let, and var

- **var** is the traditional way of declaring variables in JavaScript.
- ES6 (ECMAScript 6) introduced two new ways to declare variables, **const** and **let**, and are generally recommended in order to avoid unexpected hoisting complications.

var

- **var** has [function scope](#)
- **var** declarations are **hoisted**, but *not the initialization*

```
console.log(a); // undefined
var a = 3;
```

- The above code is equivalent to the below code due to **hoisting**.

```
var a;
console.log(a); // undefined
a=3;
```

const/let

- **const** and **let** have [block scope](#)
- **const** and **let** declarations are *not hoisted*, so variables declared with **const/let** can only be accessed after their declaration in the code.

```
console.log(a); //Uncaught ReferenceError: Cannot access 'a' before
//initialization
```

```
let a = 3;

console.log(b) //Uncaught ReferenceError: Cannot access 'b' before
//initialization
const b = 3;
```

Function Invocation

- The code inside a function *is not executed* when the function is **defined**.
- The code inside a function *is executed* when the function is **invoked**.
- It is common to use the term "call a function" instead of "invoke a function".
- It is also common to say "call upon a function", "start a function", or "execute a function".

Invoking a Function as a Function

```
function myFunction(a, b) {
  return a * b;
}
myFunction(10, 2);           // Will return 20
```

- The function above does not belong to any object. But in JavaScript there is always a default global object.
- In HTML the default global object is the HTML page itself, so the function above "belongs" to the HTML page.
- In a browser the page object is the browser window. The function above automatically becomes a window function.
- myFunction() and window.myFunction() is the same function:

```
function myFunction(a, b) {
  return a * b;
}
window.myFunction(10, 2);    // Will also return 20
```

Invoking a Function as a Method

- In JavaScript you can define functions as object methods.
- The following example creates an object (myObject), with two properties (firstName and lastName), and a method (fullName):

```
var myObject = {
  firstName: "John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}
myObject.fullName();           // Will return "John Doe"
```

- The fullName method is a function. The function belongs to the object. myObject is the owner of the function.
- The thing called this, is the object that "owns" the JavaScript code. In this case the value of this is myObject.

Invoking a Function with a Function Constructor

- If a **function invocation** is preceded with the *new* keyword, it is a *constructor* invocation.
- It looks like you create a new function, but since JavaScript functions are objects you actually create a new object:

```
// This is a function constructor:
function myFunction(arg1, arg2) {
  this.firstName = arg1;
  this.lastName = arg2;
}

// This creates a new object
var x = new myFunction("John", "Doe");
x.firstName;           // Will return "John"
```

- A constructor invocation creates a new object. The new object inherits the properties and methods from its constructor.
- The *this* keyword in the constructor does not have a value.
- The value of *this* will be the new object created when the function is invoked.

[https://www.w3schools.com/js/js_function_invocation.asp]

Scope

Scope determines the visibility or accessibility of a variable or other resource in the area of your code.

[<https://dev.to/sandy8111112004/javascript-introduction-to-scope-function-scope-block-scope-d11>]

Global Scope

There's only one **Global scope** in the JavaScript document. The area outside all the functions is considered the global scope and the variables defined inside the global scope can be accessed and altered in any other scopes.

```
//global scope
var fruit = 'apple'
console.log(fruit);           //apple

function getFruit(){
    console.log(fruit);       //fruit is accessible here
}

getFruit();                   //apple
```

Local Scope

Variables declared inside the functions become **local** to the function and are considered in the corresponding **local scope**. *Every function has its own scope*. The same variable can be used in different functions because they are bound to the respective functions and are not mutually visible.

```
//global scope
function foo1(){
    //local scope 1
    function foo2(){
        //local scope 2
    }
}
```

```
//global scope
function foo3(){
    //local scope 3
}

//global scope
```

Local scope can be divided into **function scope** and **block scope**. The concept of **block scope** is introduced in ECMAScript 6 (ES6) together with the new ways to declare variables -- `const` and `let`.

Function Scope

Whenever you declare a variable in a function, the variable is visible only within the function. You can't access it outside the function. **var** is the keyword to define a variable for a function-scope accessibility.

```
function foo(){
    var fruit = 'apple';
    console.log('inside function: ',fruit);
}

foo();                //inside function: apple
console.log(fruit);    //error: fruit is not defined
```

Block Scope

A **block scope** is the area within `if` and `switch` conditions or `for` and `while` loops. Generally speaking, *whenever you see {curly brackets}*, it is a **block**. In ES6, **const** and **let** keywords allow developers to declare variables in the **block scope**, which means *those variables exist only within the corresponding block*.

```
function foo(){
    if(true){
        var fruit1 = 'apple';        //exist in function scope
        const fruit2 = 'banana';    //exist in block scope
        let fruit3 = 'strawberry';   //exist in block scope
    }
    console.log(fruit1);
    console.log(fruit2);
    console.log(fruit3);
}
```

```
foo();  
//result:  
//apple  
//error: fruit2 is not defined  
//error: fruit3 is not defined
```

Lexical Scope

Another point to mention is the **lexical scope**. **Lexical scope** means the child scopes have access to the variables defined in the parent scope. *The child functions are lexically bound to the execution context of their parents.*

Dynamic Scope

Lexical scope is the set of rules about how the *Engine* can look-up a variable and where it will find it. The key characteristic of lexical scope is that it is defined at author-time, when the code is written (assuming you don't cheat with `eval()` or `with`).

Dynamic scope seems to imply, and for good reason, that there's a model whereby scope can be determined dynamically at runtime, rather than statically at author-time. That is in fact the case. Let's illustrate via code:

```
function foo() {  
    console.log( a ); // 2  
}  
  
function bar() {  
    var a = 3;  
    foo();  
}  
  
var a = 2;  
  
bar();
```

Lexical scope holds that the RHS reference to `a` in `foo()` will be resolved to the global variable `a`, which will result in value 2 being output.

Dynamic scope, by contrast, doesn't concern itself with how and where functions and scopes are declared, but rather where they are called from. In other words, the scope chain is based on the call-stack, not the nesting of scopes in code.

So, if JavaScript had dynamic scope, when `foo()` is executed, theoretically the code below would instead result in 3 as the output.

```
function foo() {  
    console.log( a ); // 3 (not 2!)  
}  
  
function bar() {  
    var a = 3;  
    foo();  
}  
  
var a = 2;  
  
bar();
```

How can this be? Because when `foo()` cannot resolve the variable reference for `a`, instead of stepping up the nested (lexical) scope chain, it walks up the call-stack, to find where `foo()` was *called from*. Since `foo()` was called from `bar()`, it checks the variables in scope for `bar()`, and finds an `a` there with value 3.

Strange? You're probably thinking so, at the moment.

But that's just because you've probably only ever worked on (or at least deeply considered) code which is lexically scoped. So dynamic scoping seems foreign. If you had only ever written code in a dynamically scoped language, it would seem natural, and lexical scope would be the odd-ball.

To be clear, **JavaScript does not, in fact, have dynamic scope**. It has **lexical scope**. Plain and simple. But the `this` mechanism is kind of like **dynamic scope**.

The key contrast: **lexical scope** is write-time, whereas **dynamic scope** (and `this`!) are runtime. **Lexical scope** cares *where a function was declared*, but **dynamic scope** cares where a function was *called from*.

Finally: `this` cares *how a function was called*, which shows how closely related the `this` mechanism is to the idea of **dynamic scoping**.

[<https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20%26%20closures/apA.md>]

Context vs. Scope

- **Context** refers to the *this* keyword, i.e. the object
- **Scope** refers to the *visibility of variables*

Scope Chain

To understand the **scope chain** you must know how [closures](#) work.

A **closure** is formed when you nest functions, inner functions can refer to the variables present in their outer enclosing functions even after their parent functions have already executed.

JavaScript resolves identifiers within a particular context by traversing up the **scope chain**, moving from locally to globally.

Consider this example with three nested functions:

```
var currentScope = 0; // global scope
(function () {
  var currentScope = 1, one = 'scope1';
  alert(currentScope);
  (function () {
    var currentScope = 2, two = 'scope2';
    alert(currentScope);
    (function () {
      var currentScope = 3, three = 'scope3';
      alert(currentScope);
      alert(one + two + three); // climb up the scope chain to get one and
two
    }());
  }());
}());
```

[\[https://stackoverflow.com/questions/1484143/scope-chain-in-javascript\]](https://stackoverflow.com/questions/1484143/scope-chain-in-javascript)

this - call, apply, bind

this

```
this
```

- “**this**” is the object which the function is a property of
- “**this**” gives functions access to their object and its properties

- “**this**” helps us execute the same code for multiple objects
- “**this**” can be thought of as “who called me?” i.e., what is to the left of the dot, such as window.a()
- “**this**” is **dynamically scoped**, i.e. it doesn’t matter where it was written, it matters where it was called

```

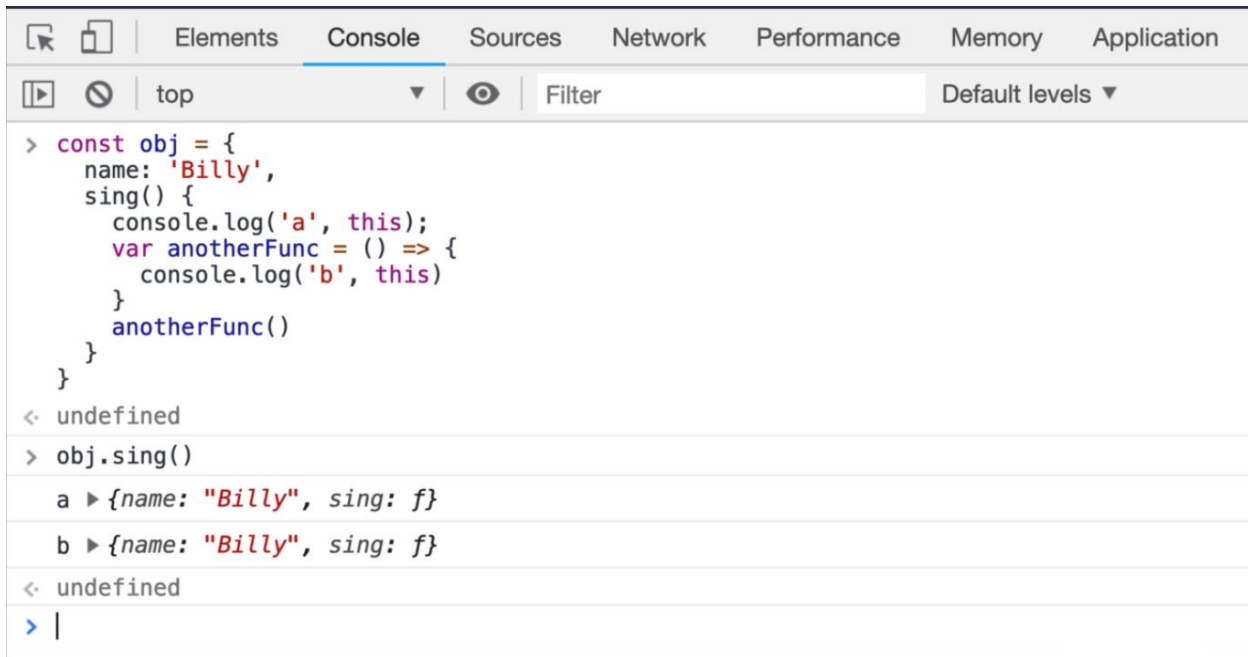
> const obj = {
  name: 'Billy',
  sing() {
    console.log('a', this);
    var anotherFunc = function() {
      console.log('b', this)
    }
    anotherFunc()
  }
}
< undefined
> obj.sing()
a ▶ {name: "Billy", sing: f}
b ▶ Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, ...}
< undefined
> obj.sing()

> var b = {
  name: 'jay',
  say() {console.log(this)}
}
var c = {
  name: 'jay',
  say() {return function() {console.log(this)}}
}
var d = {
  name: 'jay',
  say() {return () => console.log(this)}
}

c.say()()
▶ Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, ...}
< undefined
>

```

- **Arrow functions** bind “this” to the lexical scope



```
> const obj = {
  name: 'Billy',
  sing() {
    console.log('a', this);
    var anotherFunc = () => {
      console.log('b', this)
    }
    anotherFunc()
  }
}
< undefined
> obj.sing()
a ▶ {name: "Billy", sing: f}
b ▶ {name: "Billy", sing: f}
< undefined
> |
```

call()

- function a() === function.call()
- **call()** takes (<object>, ...<params>)
- Calls a method of an object, substituting an object for the current object



```
main.js  saved
1  const wizard = {
2    name: 'Merlin',
3    health: 50,
4    heal() {
5      return this.health = 100;
6    }
7  }
8
9  const archer = {
10    name: 'Robin Hood',
11    health: 30
12  }
13  console.log('1', archer)
14  wizard.heal.call(archer)
15  console.log('2', archer)
```

Native Browser JavaScript

```
>
1 { name: 'Robin Hood', health: 30 }
2 { name: 'Robin Hood', health: 100 }
=> undefined
>
```

apply()

- **apply()** is the same as **call()** except that **call()** takes (*<object>*, ...*<params>*) and **apply()** takes (*<object>*, *<[array of params]>*)

bind()

- The **bind()** method creates a new function that, when called, has its **this** keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.
- **function.bind(thisArg[, arg1[, arg2[, ...]]])**
 - *thisArg*
 - The value to be passed as the **this** parameter to the target function when the bound function is called. The value is ignored if the bound function is constructed using the **new** operator. When using **bind** to create a function (supplied as a callback) inside a `setTimeout`, any **primitive** value passed as *thisArg* is converted to **object**. If no arguments are provided to `bind`, the **this** of the executing scope is treated as the *thisArg* for the new function.
 - *arg1, arg2, ...*
 - Arguments to prepend to arguments provided to the bound function when invoking the target function.

```
Function.prototype.bind = function(whoIsCallingMe){
  const self = this;
  return function(){
    return self.apply(whoIsCallingMe, arguments);
  };
}
```

- **bind()** is especially useful for remedying the **dynamic scope** property of **this** by binding it to a lexical scope

```
var module = {
  x: 42,
  getX: function() {
    return this.x;
  }
}
var unboundGetX = module.getX;
console.log(unboundGetX()); // The function gets invoked at the global
scope
// expected output: undefined
```



```
var boundGetX = unboundGetX.bind(module);
console.log(boundGetX());
// expected output: 42
```

```
4   heal(num1, num2) {
5       return this.health += num1 + num2;
6   }
7   }
8
9   const archer = {
10       name: 'Robin Hood',
11       health: 30
12   }
13   console.log('1', archer)
14   const healArcher = wizard.heal.bind
15   (archer, 100, 30)
16   healArcher()
17   console.log('2', archer)
```

```
1 { name: 'Robin Hood', health: 30 }
2 { name: 'Robin Hood', health: 160 }
=> undefined
```

Functions Expressions vs Function Statements

- An **expression** produces a value and can be written wherever a value is expected, for example as an argument in a function call
- Each line below is an expression

```
myvar
3 + x
myfunc("a", "b")
```

- Roughly, a **statement** performs an action. **Loops** and **if statements** are examples of statements. A program is basically a sequence of statements (we're ignoring declarations here). Wherever JavaScript expects a statement, you can also write an expression. Such a statement is called an *expression statement*. The reverse does not hold: you cannot write a statement where JavaScript expects an expression. For example, an if statement cannot become the argument of a function.
- **Expression**: produces a value
- **Statement**: performs an action
- **Expression statement**: produces a value and performs an action

- The following is an example of an **if statement**:

```
var x;  
if (y >= 0) {  
    x = y;  
} else {  
    x = -y;  
}
```

- Expressions have an analog, the conditional operator. The above statements are equivalent to the following statement.

```
var x = (y >= 0 ? y : -y);
```

IIFE

```
(function () {} )() === (function () {} )()  
// undefined === undefined
```

- **Immediately Invoked Function Expressions** have been used in the past, before modules, to avoid polluting the namespace. It does this by limiting the scope of variables in the function to the function scope

```
var script1 = (function () {  
    function a() {  
        return 5;  
    }  
    return {  
        a: a  
    }  
})()
```

Use Strict

- Strict mode makes it easier to write "secure" JavaScript.
- Strict mode changes previously accepted "bad syntax" into real errors.
- As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.
- In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.

- In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.
- [W3Schools](https://www.w3schools.com/js/js_strict.asp) has a good list of what is not allowed in strict mode

[https://www.w3schools.com/js/js_strict.asp]

Types In JavaScript

Section 1

Static vs. Dynamically Typed

- JS is both **dynamically typed** and **weakly typed**.
- **Statically typed** means the type is enforced and won't change so easily. All variables must be declared with a type.
- **Static Typing** is exemplified by C++ and Java's use of variable declaration

```
int a == 100 // explicitly declares the value type as integer
```

- **Dynamically typed** languages infer variable types at runtime. This means once your code is run the compiler/interpreter will see your variable and its value then decide what type it is. The type is still enforced here, it just decides what the type is.
- **Dynamic Typing** is exemplified by JavaScript's use of 'var, const, let' to declare variables. The compiler takes responsibility for determining the type of the value

```
let a = 100 // compiler determines it is an integer
```

- Strong vs Weak Typing
 - **Strong Typing** *does not* allow different value types to interact, e.g.

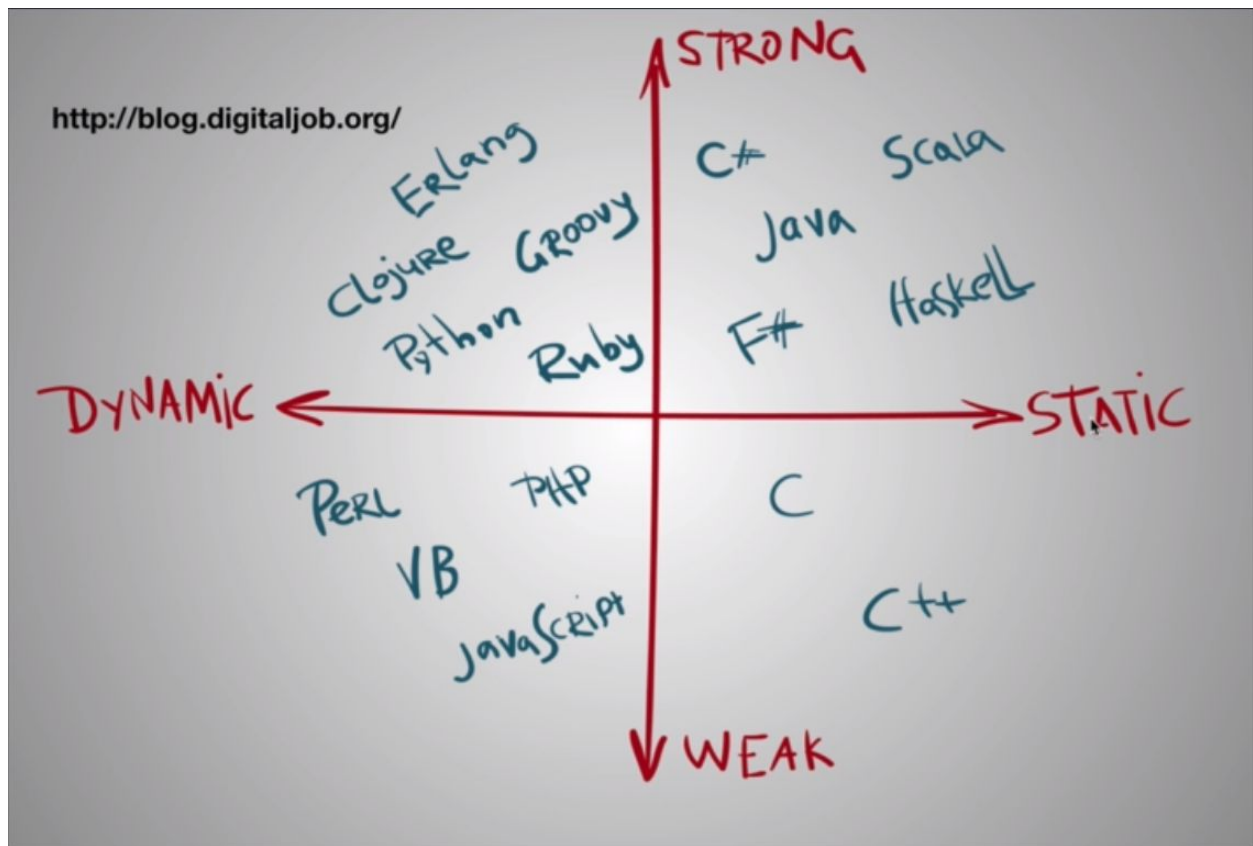
```
var hello = 'hello'
hello + 17 // ERROR
```

- **Weak Typing** *does* allow different value types to interact, e.g. **coercing** an integer into its string equivalent when interacting with a string

```
var hello = 'hello'
hello + 17 // 'hello17'
```

- **Weakly typed** languages allow types to be inferred as another type. For example, 1 + '2' // '12' In JS it sees you're trying to add a number with a

string — an invalid operation — so it coerces your number into a string and results in the string '12'.



Primitive Types

- A **primitive** is not an object and has no methods of its own. All primitives are immutable.
 - **Boolean** — true or false
 - **Null** — no value
 - **Undefined** — a declared variable but hasn't been given a value
 - **Number** — integers, floats, etc
 - **String** — an array of characters, i.e words
 - **Symbol** — a unique value that's not equal to any other value
 - **Everything else is an Object type.**
- **Primitives** are wrapped in objects

```
true.toString() === 'true'
```

- It works some like the below:

```
Boolean(true).toString() === 'true'
```

Section 2

Pass By Reference vs. Pass By Value

- **Pass by Value** is used with primitives

```
let a = 5;
let b = 6;
a = 7;
console.log(a) // 7
console.log(b) // 6
```

- **Pass by Reference** is used with objects

```
let a = {a: 1, b: 2}
let b = a;
a.b = 3;
console.log(a) // {a: 1, b: 3}
console.log(b) // {a: 1, b: 3}
```

- To **shallow clone** an object we can use **Object.assign** or the **rest operator**

```
let a = {a:1, b:2}
let b = Object.assign({}, a) // with rest operator {...a}

a.b = 3;
console.log(a) // {a:1, b:3}
console.log(b) // {a:1, b:2}
```

- To clone an array, we can use:

```
let a = [1,2,3]
let b = [].concat(a)
a[0] = 5
console.log(a) // [5,2,3]
console.log(b) // [1,2,3]
```

- To **deep clone** an object we need to **stringify** and **parse** with **JSON**
 - This is because of **pass by reference**, when objects are declared, a memory location, “**reference**”, is passed, rather than the object value being copied.

```
let a = {a:1, b: { c: 3}}
```

```
let deepClone = JSON.parse(JSON.stringify(a))
```

Type Coercion

- Type Coercion is the use of operators to coerce a type change in a value
- **Try to avoid type coercion as it is difficult to read and predict**



- JavaScript provides three different value-comparison operations:
 - **===** - Strict Equality Comparison ("strict equality", "identity", "triple equals")
 - **==** - Abstract Equality Comparison ("loose equality", "double equals")
 - **Object.is** provides SameValue (new in ES2015).
- Which operation you choose depends on what sort of comparison you are looking to

perform. Briefly:

- double equals (==) will perform a type conversion when comparing two things, and will handle NaN, -0, and +0 specially to conform to IEEE 754 (so NaN != NaN, and -0 == +0);
- triple equals (===) will do the same comparison as double equals (including the special handling for NaN, -0, and +0) but without type conversion; if the types differ, false is returned.
- Object.is does no type conversion and no special handling for NaN, -0, and +0 (giving it the same behavior as === except on those special numeric values).

```
1 == "1" // true
1 === 1 // false
NaN === NaN // false
```

Sameness Comparisons

x	y	==	===	Object.is	SameValueZero
undefined	undefined	true	true	true	true
null	null	true	true	true	true
true	true	true	true	true	true
false	false	true	true	true	true
'foo'	'foo'	true	true	true	true
0	0	true	true	true	true
+0	-0	true	true	false	true
+0	0	true	true	true	true
-0	0	true	true	false	true
0	false	true	false	false	false
""	false	true	false	false	false
""	0	true	false	false	false
'0'	0	true	false	false	false
'17'	17	true	false	false	false
[1, 2]	'1,2'	true	false	false	false
new String('foo')	'foo'	true	false	false	false
null	undefined	true	false	false	false
null	false	false	false	false	false
undefined	false	false	false	false	false
{ foo: 'bar' }	{ foo: 'bar' }	false	false	false	false
new String('foo')	new String('foo')	false	false	false	false
0	null	false	false	false	false
0	NaN	false	false	false	false
'foo'	NaN	false	false	false	false
NaN	NaN	false	false	true	true

[<https://dorey.github.io/JavaScript-Equality-Table/>]

[<https://www.ecma-international.org/ecma-262/5.1/#sec-11.9.3>]

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness]

Arrays, Functions, Objects

Dot Notation vs. Bracket Notation

- Both can be used to access **properties** of an **object** using the **key**
- **Dot Notation**
 - When working with **dot notation**, property **keys** can only be alphanumeric (and `_` and `$`). It can't start with a number.

```
let obj = {  
  cat: 'meow',  
  dog: 'woof'  
};  
let sound = obj.cat;  
console.log(sound);  
// meow
```

- **Bracket Notation**
 - When working with **bracket notation**, property **keys** only have to be a **String**. They can include *any characters*, including spaces. *Variables* may also be used *as long as the variable resolves to a String*.

```
let arr = ['a', 'b', 'c'];  
let letter = arr[1];  
console.log(letter);  
// b
```

The Two Pillars

Section 1

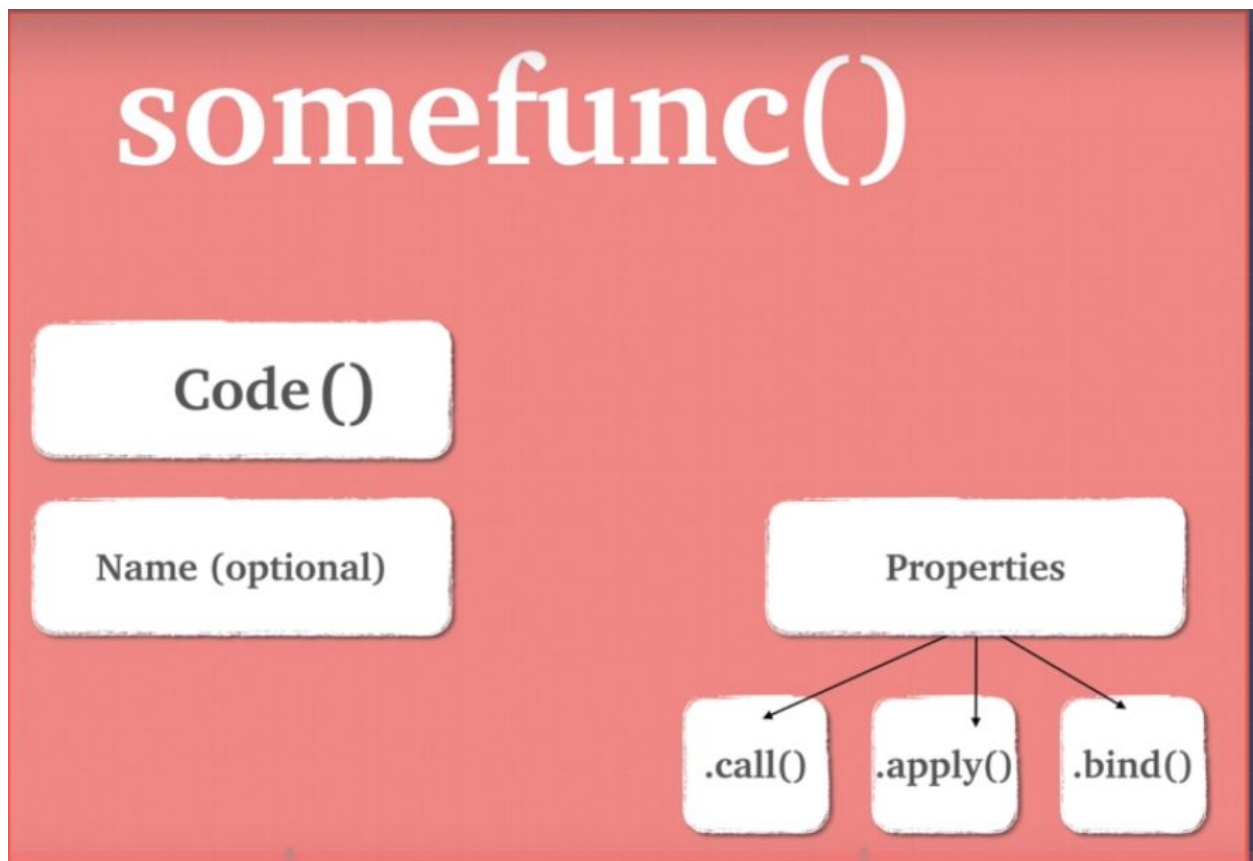
Higher Order Functions

Higher Order Functions are functions that take functions as parameters and/or return

functions. Examples of **Higher Order Functions** can be seen in Redux.

Functions vs. Objects

- **Functions** are **callable objects**



- **Functions** are first class citizens. **Functions** are:
 - Assignable to variables
 - Able to be passed as arguments
 - Able to be returned from functions
 - Beware:
 - Initializing **functions** in loops
 - Declaring **functions** with parameters without defaults
 - **Set parameter defaults!**
-

Section 2

Closures

- **Closure** allows access to variables within its lexical scope, including the variables of parents which have been removed from the **call stack** by determining which variables will be needed by child functions and retaining them in memory.
- **Closures** are like objects in that they are a mechanism for containing **state**

```
//Closure
const closure = function() {
  let count = 0;
  return function increment() {
    count++;
    return count;
  }
}

const incrementFn = closure()
incrementFn() //1
incrementFn() //2
incrementFn() //3
```

- Memory Efficient
 - By using **closures**, we can create variables that are stored in memory to be used in the future

```
const closureTest1 = (function() {
  const bigArray = new Array(7000).fill('1')
  console.log('created')
  return function(index) {
    return bigArray[index]
  }
})();

closureTest1(500)
closureTest1(300)
closureTest1(100)
```

```
created //from console.log('created')
//returned "1"
```

- A typical case without **closure** would cause the variable to be created and stored again each time it is needed. This is memory *inefficient*.

```
function closureTest(index) {
  const bigArray = new Array(7000).fill('1')
  console.log('created')
  return bigArray[index]
}
closureTest(500)
closureTest(300)
closureTest(100)
created //from console.log('created')
created //from console.log('created')
created //from console.log('created')
// returned "1"
```

[https://www.w3schools.com/js/js_function_closures.asp]

- Encapsulation
 - **Encapsulation** allows us to hide/show properties of functions
 - In the below example, we don't want to expose the *launch* variable, so we do not return it

```
const makeNuclearButton = () => {
  let timeWithoutDestruction = 0;
  const passTime = () => timeWithoutDestruction++;
  const totalPeaceTime = () => timeWithoutDestruction;
  const launch = () => {
    timeWithoutDestruction = -1;
    return '💣';
  }

  setInterval(passTime, 1000);
  return {totalPeaceTime}
}

const ww3 = makeNuclearButton();
ww3.totalPeaceTime()
```

[<https://www.intertech.com/Blog/encapsulation-in-javascript/>]

Prototypal Inheritance

- A prototype is a working **object** instance. Objects inherit directly from other objects.
- **__proto__** is a reference to the parent object's **prototype** property, e.g.

```
const obj = {}  
obj.__proto__ === Object.prototype // true
```

- The **prototype** property only belongs to functions, specifically, **constructor** functions.
 - The **Object constructor** creates an object wrapper.
- The **__proto__** and **prototype** properties are used to create a chain of **inheritance** of properties between objects, beginning with **Object** and **Primitive Types**
 - **Object.create()** can be used to create **objects** with its **__proto__** property linked to the **prototype** property of the object passed as **Object.create()**'s argument
- **Object** is the base **function** (constructor)
 - The root of everything in JavaScript is **Object** which is actually a **function**

```
typeof Object // "function"
```

- **Object** has the property **prototype** which is the **base object** for all things in JavaScript, including JavaScript functions

```
typeof Object.prototype // "object"
```

Object-Oriented Programming vs. Functional Programming

OOP vs. FP

Object-Oriented Programming (OOP)

- Organizing the code into units
- An **object** is a box containing information (**state/attributes**) and operations (**methods**) that refer to the same concept, *what it is*
- **Objects** are **first-class citizens**
- Few operations on common data
- Very “stateful”, we modify state

- More **imperative**
- Better with many *things* and fewer *operations*

Functional Programming (FP)

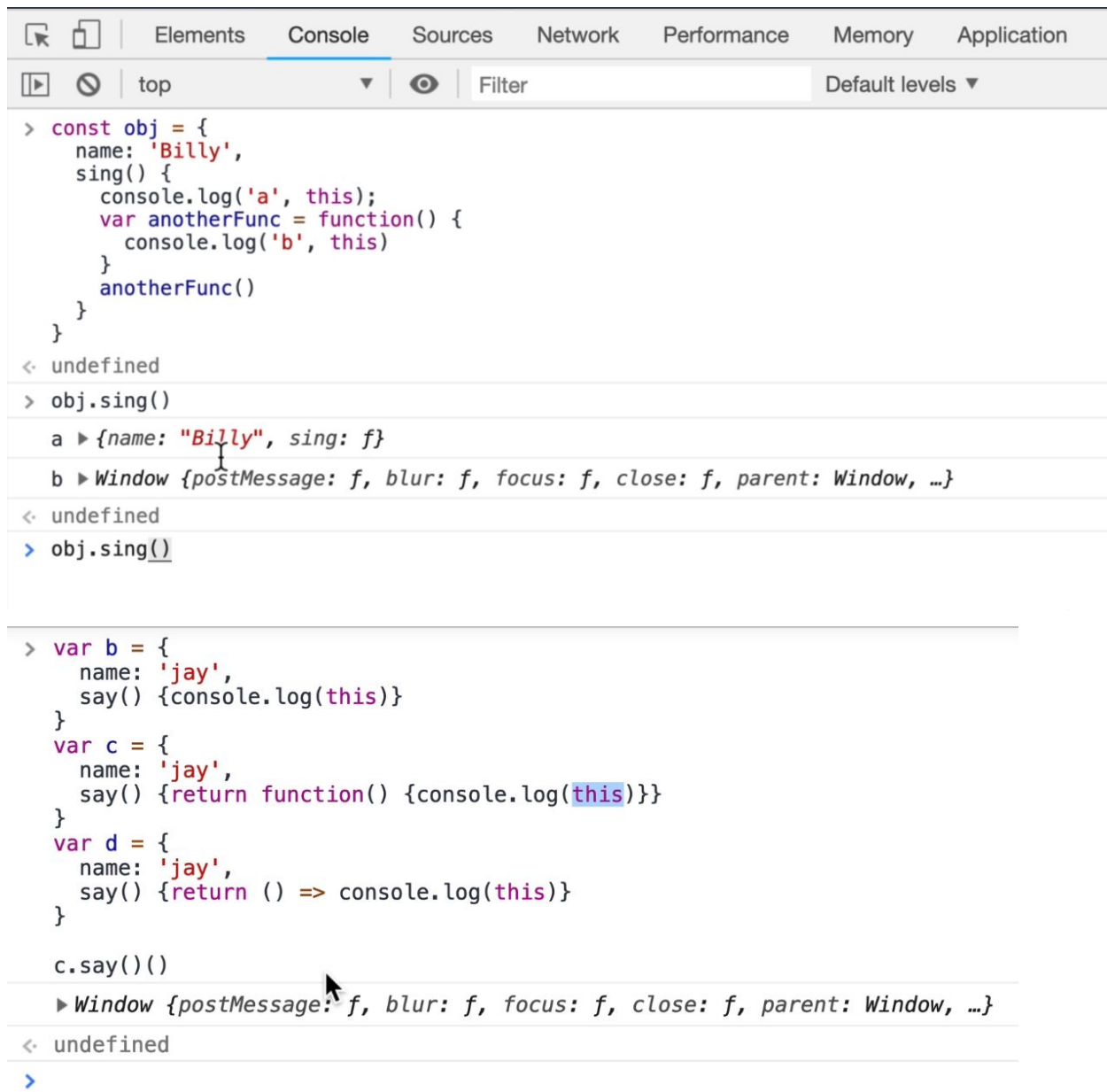
- A combination of functions
- Avoiding **side-effects**
- **Immutable** state
- **Pure functions**
- Functions are **first-class citizens**
- **Composition** is more common than procedural actions, e.g. loops, iterations, and if-else
- Manipulate data structures like trees, array, and objects
- Performing many operations for which the data is fixed
- Can run better in parallel processes due to fewer **side-effects**
- More **declarative**
- Good at processing large data for apps

Object Oriented Programming

“this” keyword

```
this
```

- “**this**” is the object which the function is a property of
- “**this**” gives functions access to their object and its properties
- “**this**” helps us execute the same code for multiple objects
- “**this**” can be thought of as “who called me?” i.e., what is to the left of the dot, such as window.a()
- “**this**” is **dynamically scoped**, i.e. it doesn’t matter where it was written, it matters where it was called



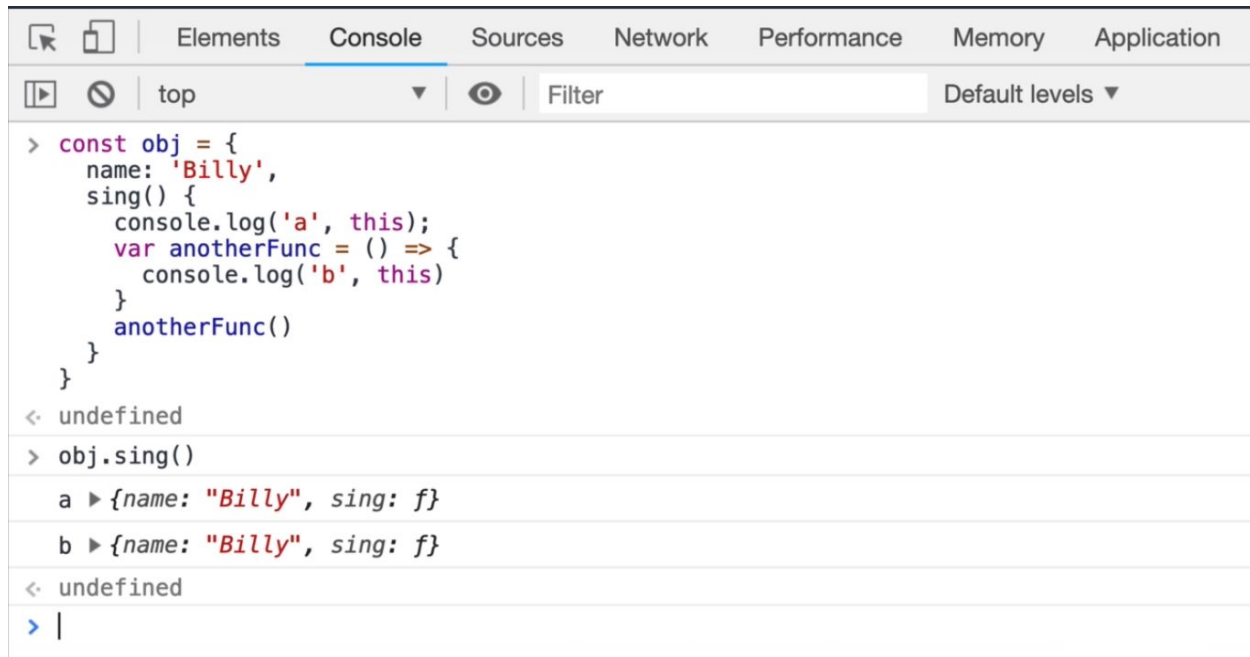
The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following code and its execution results:

```
> const obj = {
  name: 'Billy',
  sing() {
    console.log('a', this);
    var anotherFunc = function() {
      console.log('b', this)
    }
    anotherFunc()
  }
}
< undefined
> obj.sing()
a ▶ {name: "Billy", sing: f}
b ▶ Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, ...}
< undefined
> obj.sing()
```

```
> var b = {
  name: 'jay',
  say() {console.log(this)}
}
var c = {
  name: 'jay',
  say() {return function() {console.log(this)}}
}
var d = {
  name: 'jay',
  say() {return () => console.log(this)}
}

c.say()()
▶ Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, ...}
< undefined
>
```

- **Arrow functions** bind “this” to the lexical scope



The screenshot shows a web browser's developer console with the 'Console' tab selected. The code entered is:

```
> const obj = {  
  name: 'Billy',  
  sing() {  
    console.log('a', this);  
    var anotherFunc = () => {  
      console.log('b', this)  
    }  
    anotherFunc()  
  }  
}  
< undefined  
> obj.sing()  
a ▶ {name: "Billy", sing: f}  
b ▶ {name: "Billy", sing: f}  
< undefined  
> |
```

The output shows the execution of the code, with the console log displaying the objects 'a' and 'b' as returned by the `console.log` statements inside the `sing` function.

“new” keyword

- Creates a blank, plain JavaScript **object**;
 - Links (sets the **constructor** of) this **object** to another **object**;
 - Passes the newly created **object** from Step 1 as the **this** context;
 - Returns **this** if the function doesn't return its own **object**.
-

“in” operator

- The **in operator** returns true if the specified property is in the specified object or its prototype chain.
-

Prototype

- A **prototype** is a working **object** instance. Objects inherit directly from other objects.
- **__proto__** is a reference to the parent object's **prototype** property, e.g.

```
const obj = {}  
obj.__proto__ === Object.prototype // true
```

- The **prototype** property only belongs to functions, specifically, **constructor** functions.
 - The **Object constructor** creates an object wrapper.
- The **__proto__** and **prototype** properties are used to create a chain of **inheritance** of properties between objects, beginning with **Object** and **Primitive Types**
 - **Object.create()** can be used to create **objects** with its **__proto__** property linked to the **prototype** property of the object passed as **Object.create()**'s argument
- **Object** is the base **function** (constructor)
 - The root of everything in JavaScript is **Object** which is actually a **function**

```
typeof Object // "function"
```

- **Object** has the property **prototype** which is the **base object** for all things in JavaScript, including JavaScript functions

```
typeof Object.prototype // "object"
```

ES6 Classes

- The **class** keyword in JS is syntactic sugar. Under the hood, it still uses **prototypal inheritance**
- **Instances** of a **class** must be instantiated with the **new** keyword
- The **constructor** method is used to instantiate the **state** (data) of a new **object**
 - The **state** is typically unique to each **instance**
- Functions are typically not included in the **constructor** because it would create a memory reference for the function in each new **instance** of the **class**, thus using more memory than needed
 - By including functions as methods of the **class**, **instances** of the **class** can reference the function via the **prototype chain**
- **Prototypal Inheritance** has better memory efficiency than **classical inheritance** due to it sharing the memory references of its **prototype** properties with those **objects** that **inherit** from it. In **classical inheritance**, **instances** of the **class** create new memory references for each inherited property.

Java

Object.create() vs. Classes

- Both **Object.create()** and **class** are ways to create a **prototype chain**
 - Some people prefer to avoid the **constructor**, **class**, and **this** keywords as much as possible to limit confusion due to **this**
 - Some people prefer to use the **constructor**, **class**, and **this** keywords, perhaps because of its similarity to other languages with the Object-Oriented Programming paradigm
-

Private vs. Public vs. Protected

They are access modifiers and help us implement **Encapsulation** (or information hiding). They tell the compiler which other classes should have access to the field or method being defined.

Private - Only the current class will have access to the field or method.

Protected - Only the current class and subclasses (and sometimes also same-package classes) of this class will have access to the field or method.

Public - Any class can refer to the field or call the method.

4 Principles of OOP

Encapsulation

<https://www.intertech.com/Blog/encapsulation-in-javascript/>

- **Encapsulation** allows us to hide/show properties of functions
 - In the below example, we don't want to expose the *launch* variable, so we do not return it
-

Abstraction

In object-oriented programming theory, **abstraction** involves the facility to define objects that represent abstract "actors" that can perform work, report on and change their state, and "communicate" with other objects in the system. The term **encapsulation** refers to the hiding of state details, but extending the concept of data type from earlier programming languages to associate behavior most strongly with the data, and standardizing the way that different data types interact, is the beginning of **abstraction**. When **abstraction** proceeds into the operations defined, enabling objects of different types to be substituted, it is called **polymorphism**. When it proceeds in the opposite direction, inside the types or classes, structuring them to simplify a complex set of relationships, it is called **delegation** or **inheritance**.

Various object-oriented programming languages offer similar facilities for **abstraction**, all to support a general strategy of **polymorphism** in object-oriented programming, which includes the substitution of one type for another in the same or similar role. Although not as generally supported, a configuration or image or package may predetermine a great many of these bindings at compile-time, link-time, or load-time. This would leave only a minimum of such bindings to change at run-time.

[\[https://en.wikipedia.org/wiki/Abstraction_\(computer_science\)#Abstraction_in_object_oriented_programming\]](https://en.wikipedia.org/wiki/Abstraction_(computer_science)#Abstraction_in_object_oriented_programming)

Polymorphism

- **Polymorphism** is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.
 - For example, given a base **class** *shape*, **polymorphism** enables the programmer to define different area **methods** for any number of derived classes, such as *circles*, *rectangles* and *triangles*. No matter what shape an object is, applying the area **method** to it will return the correct results.
-

Inheritance

- **Inheritance** is the mechanism of basing an **object** or **class** upon another **object** (*prototype-based inheritance*) or **class** (*class-based inheritance*)
- The focus in **inheritance** is on *what it is*
 - Data, Methods, Functions
- Prototypal Inheritance
 - **Prototype-based programming** is a style of object-oriented programming in which behaviour reuse (known as **inheritance**) is performed via a process of *reusing existing objects* via **delegation** that serve as **prototypes**.
 - Advocates of prototype-based programming argue that it encourages the programmer to focus on the behavior of some set of examples and only later worry about classifying these objects into archetypal objects that are later used in a fashion similar to classes
- Classical Inheritance
 - **Class-based programming**, or more commonly class-orientation, is a style of Object-oriented programming (OOP) in which **inheritance** occurs via defining **classes of objects**, instead of **inheritance** occurring *via the objects alone*
 - A **class** is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods). In many languages, the **class** name is used as the name for the **class** (the template itself), the name for the default **constructor**

of the class (a subroutine that creates objects), and as the **type of objects** generated by instantiating the class; these distinct concepts are easily conflated.

- **Tight Coupling** can occur which refers to the “ripple effects” that can happen to **subclasses** (child class) when a change is made to a **superclass** (parent class)
 - **Tight Coupling** can lead to many unintended effects for the **subclasses**
 - **Tight Coupling** can help avoid repetition in code
 - The **fragile base class problem** is a fundamental architectural problem of object-oriented programming systems where base classes (**superclasses**) are considered “fragile” because seemingly safe modifications to a base class, when inherited by the derived classes, may cause the derived classes to malfunction.
 - The programmer cannot determine whether a base class change is safe simply by examining in isolation the methods of the base class.
- The **Gorilla-Banana Problem** refers to the problem of inheriting *too much* from **superclasses**. “You want a banana, but what you get is a gorilla holding a banana in a jungle”.
- **Classical Inheritance** requires *excellent foresight* to avoid the problems of improper inheritance. Refactoring the structure can be a nightmare.

```
//Inheritance
//what it is
class Character {
  constructor(name, weapon) {
    this.name = name;
    this.weapon = weapon;
  }
  attack() {
    return 'atack with ' + this.weapon
  }
}

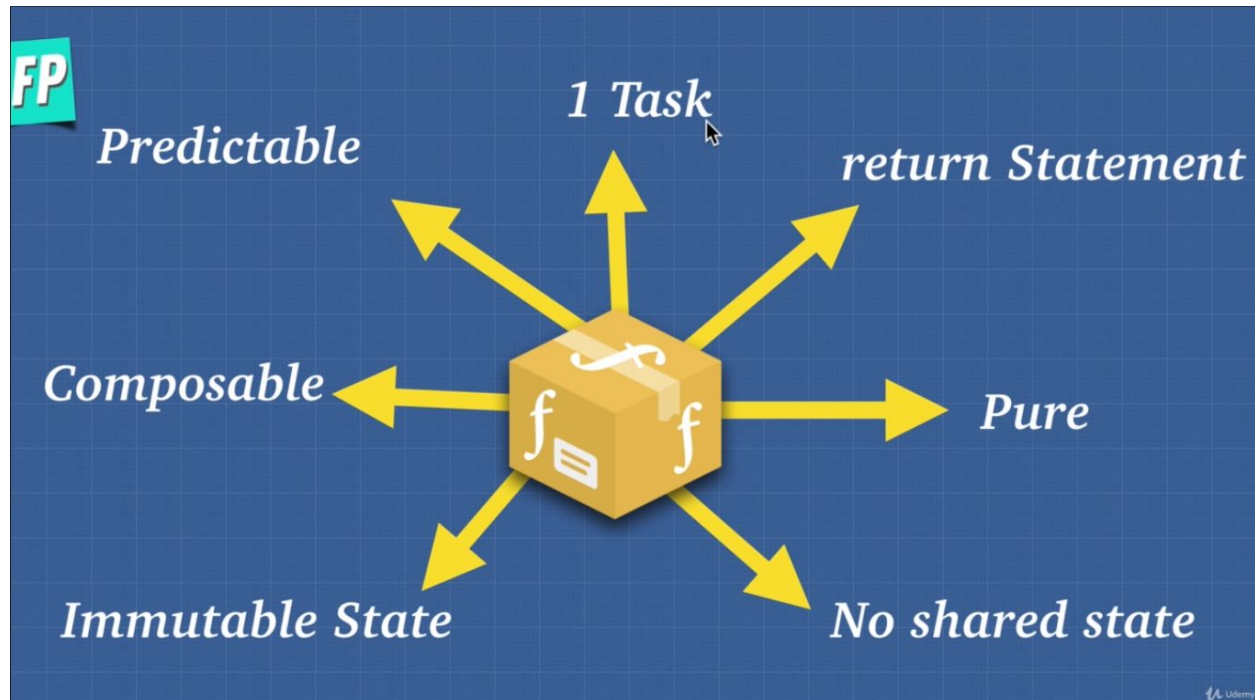
class Elf extends Character {
  constructor(name, weapon, type) {
    super(name, weapon)
    console.log('what am i?', this);
    this.type = type;
  }
}

class Ogre extends Character {
  constructor(name, weapon, color) {
    super(name, weapon);
```

Functional Programming

The goal of **Functional Programming** is to *minimize side effects* and compartmentalize functions so that when there is a bug, you know exactly where to go.

An image describing the aspects of good, pure functions



Immutability

- Immutability is about not *changing* the **state**, but rather, copying the old state, changing the new copy and *replacing* the old **state** with the new **state**
- **Immutable objects** are simpler to construct, test, and use
 - Truly **immutable objects** are always thread-safe
 - They help to avoid **temporal coupling**
 - **Temporal coupling** is coupling that occurs when there are two or more members of a class that need to be *invoked in a particular order*
 - Their usage is **side-effect free**
- A **persistent data structure** is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively **immutable**, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure.

- **Structural sharing** is one of the techniques used for optimizing persistent data structures
-

Idempotence

- **Idempotence** denotes predictability, i.e. given the same input, the function should always return the same output
-

Imperative vs. Declarative

- **Imperative code** is code that tells the machine *what to do and how to do it*
 - Machines are good with **imperative instructions**, e.g. “Please walk to the table, use your right hand to pick up the water, walk back to me and give me the water.”

An example of imperative code

```
for (let i = 0; i < 1000; i++) {console.log(i)}
```

- **Declarative code** is code that tells the machines *what to do and what should happen*, it doesn't tell the machine *how to do it*
 - Humans are good with **declarative instructions**, e.g. “Please give me that water”

An example of declarative code

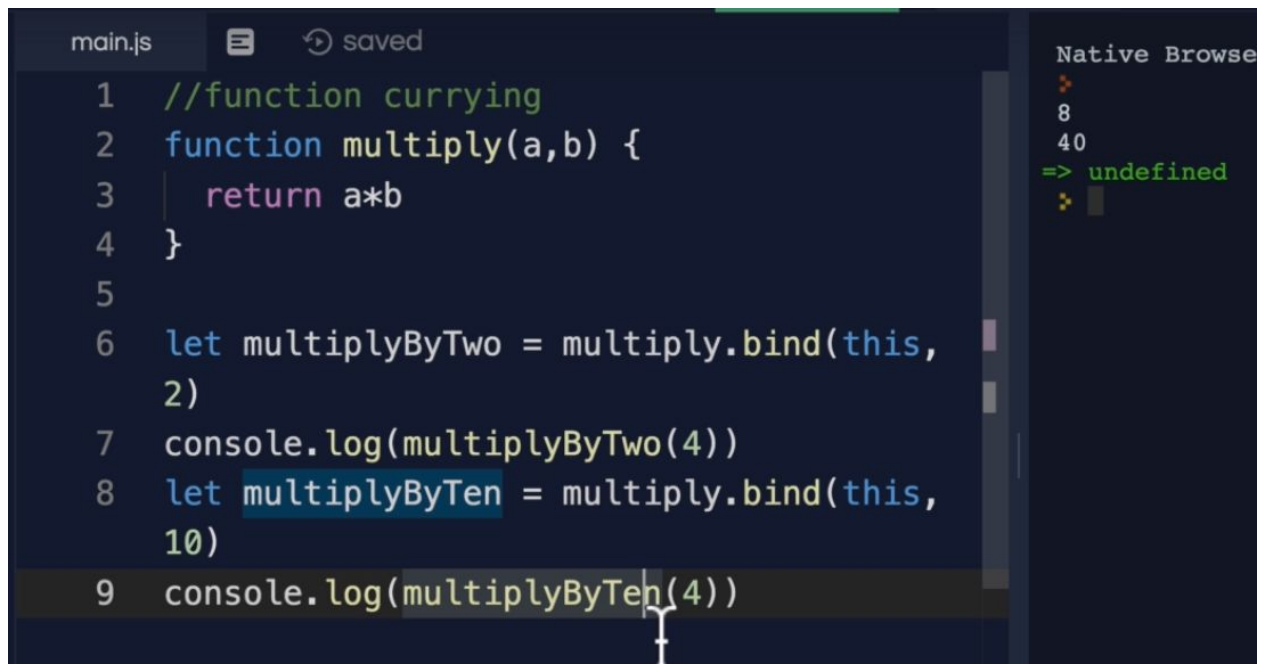
```
[1,2,3].forEach(i => console.log(i))
```

Curry

- **Currying** transforms a function with multiple arguments into a sequence/series of functions each taking a single argument.
- **Currying** can remind us of **methods** shared through **prototypes** of objects which saves *memory*

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

```
function multiply(a) {  
  return (b) => {  
    return (c) => {  
      return a * b * c  
    }  
  }  
}  
log(multiply(1)(2)(3)) // 6
```



```
main.js  saved  
1 //function currying  
2 function multiply(a,b) {  
3   return a*b  
4 }  
5  
6 let multiplyByTwo = multiply.bind(this,  
7   2)  
7 console.log(multiplyByTwo(4))  
8 let multiplyByTen = multiply.bind(this,  
9   10)  
9 console.log(multiplyByTen(4))
```

Native Browser
8
40
=> undefined

Partial Application

- **Partial Application** is similar to **currying**
- It is a process of producing a function with a smaller number of parameters

Partial Application vs. Currying

- **Currying** expects *one argument at a time*
 - **Partial Application** expects the *rest* of the needed arguments on the *second call*
-

Caching

- **Caching** is a way to store values so you can use them later

```
function addTo80(n){
  console.log('simulate long computation time')
  return n + 80
}

let cache = {}

function memoizedAddTo80(n) {
  if (n in cache){
    return cache[n];
  } else {
    console.log('simulate long computation time')
    cache[n] = n + 80;
    return cache[n];
  }
}
```

Memoization

- **Memoization** is a specific kind of **caching**
 - It **caches** a *return value* based on its parameters
 - So, if the parameters are the same, it does a *value lookup* instead of a running the function and calculating the *return value* again
 - Doing so can save valuable computing cycles

```
function addTo80(n){
  console.log('simulate long computation time')
  return n + 80
}

let cache = {}

function memoizedAddTo80(n) {
  if (n in cache){
    return cache[n];
  } else {
    console.log('simulate long computation time')
    cache[n] = n + 80;
    return cache[n];
  }
}
```

To avoid polluting the global namespace, use closures

```
function memoizedAddTo80WithClosure(n) {
  let cache = {};
  return function(n){
    if (n in cache){
      return cache[n];
    } else {
      console.log('simulate long computation time')
      cache[n] = n + 80;
      return cache[n];
    }
  }
}
```

```
}  
const memoizedAndEnclosedFunction = memoizedAddTo80WithClosure();
```

Pure Functions

- A **pure function** must always return the same output given the same input
- **Pure functions** are *easy to test*, *easy to compose*, and *avoid bugs*.
- No **side effects**: **pure functions** cannot modify anything outside of themselves

A function with a **side effect**, it modifies an array outside of the function

```
const array = [1,2,3]  
const mutateArray = (arr) => {  
  arr.pop()  
}
```

The same functionality, but with a **pure function**, i.e. no **side effects** because it creates a new array with `concat()` and returns the new array. It does not affect anything outside of its scope

```
const array = [1,2,3]  
const removeLastItem = (arr) => {  
  const newArr = [].concat(arr)  
  newArr.pop()  
  return newArr  
}
```

Referential Transparency

Referential Transparency is generally defined as the fact that an expression, in a program, *may be replaced by its value* (or anything having the same value) *without changing the result of the program*. This implies that methods should always return the same value for a given argument, without having any other effect.

Compose

- **Function composition** is an act or mechanism to *combine simple functions* to build *more complicated ones*

```
//Compose
const compose = (f,g) => (data) => f(g(data))
const multiplyBy3 = (num) => num*3
const makePositive = (num) => Math.abs(num)
const multiplyBy3AndAbsolute = compose(multiplyBy3, makePositive)
multiplyBy3AndAbsolute(-50) // 150
```

Pipe

- A **pipeline** consists of a *chain of processing elements* (processes, threads, coroutines, functions, etc.), arranged so that *the output of each element is the input of the next*; the name is by analogy to a physical pipeline.

```
pipe = (...fns) => x => fns.reduce((v, f) => f(v), x)
```

```
const pipe = (f,g) => (data) => g(f(data))
const multiplyBy3 = (num) => num*3
const makePositive = (num) => Math.abs(num)
const multiplyBy3AndAbsolute = pipe(multiplyBy3, makePositive)
multiplyBy3AndAbsolute(-50) // 150
```

Pipe vs. Compose

- **compose** works exactly the same way as **pipe**, except that it applies the functions in *right-to-left order* instead of *left-to-right*

```
const pipe = (f,g) => (data) => g(f(data))
```

```
const compose = (f,g) => (data) => f(g(data))
```

Composition vs. Inheritance

- **Inheritance** is a *top-down* approach. It can be great with good planning and a solid, simple foundation.
 - Focuses on *what it is*
 - It can reduce repetition and be easy to read
- **Composition** is more of a *bottom-up* approach.
 - Focuses on *what it does*
 - You can create all of the different elements needed (objects, functions, etc.) and piece them together as needed
 - It can be more difficult to follow and read
 - **Composition** can be easier to refactor and avoid side-effects
 - It requires less planning
 - The focus is on *what it has, what it does to data, what its abilities are*

```

//Composition
//what it has
function getAttack(character) {
  return Object.assign({}, character, { attackFn: () => {
  }
});
}

function Elf(name, weapon, type) {
  let elf = {
    name,
    weapon,
    type
  };
  return getAttack(elf);
}

Elf = attack() + sleep()
Ogre = attack() + makeFort() + sleep()

const user = {
  name: 'Kim',
  active: true,
  cart: [],
  purchases: []
}

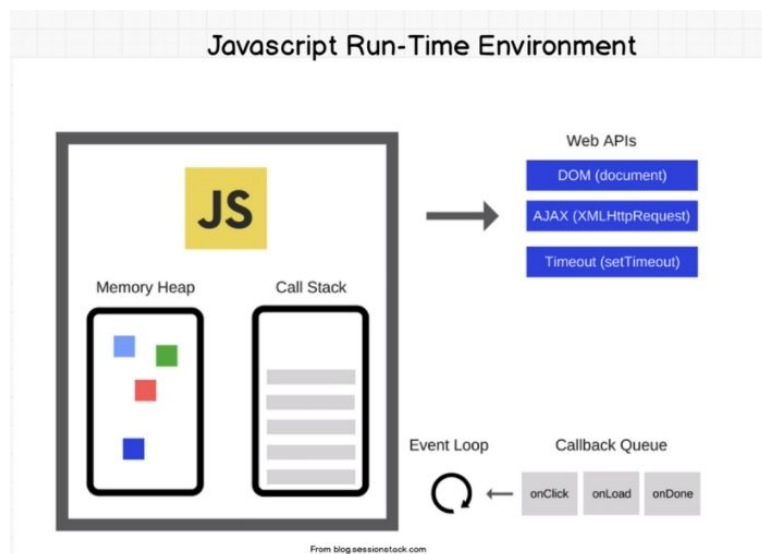
```

Delegation

Delegation refers to evaluating a member (property or method) of one **object** (the receiver) in the context of another *original object* (the sender). **Delegation** can be done **explicitly**, by *passing the sending object to the receiving object*, which can be done in any object-oriented language; or **implicitly**, by the member lookup rules of the language, which requires language support for the feature. **Implicit delegation** is the fundamental method for behavior reuse in **prototype-based programming**, corresponding to **inheritance** in **class-based programming**. The best-known languages that support delegation at the language level are Self, which incorporates the notion of delegation through its notion of mutable parent slots that are used upon method lookup on self calls, and **JavaScript**.

Extras

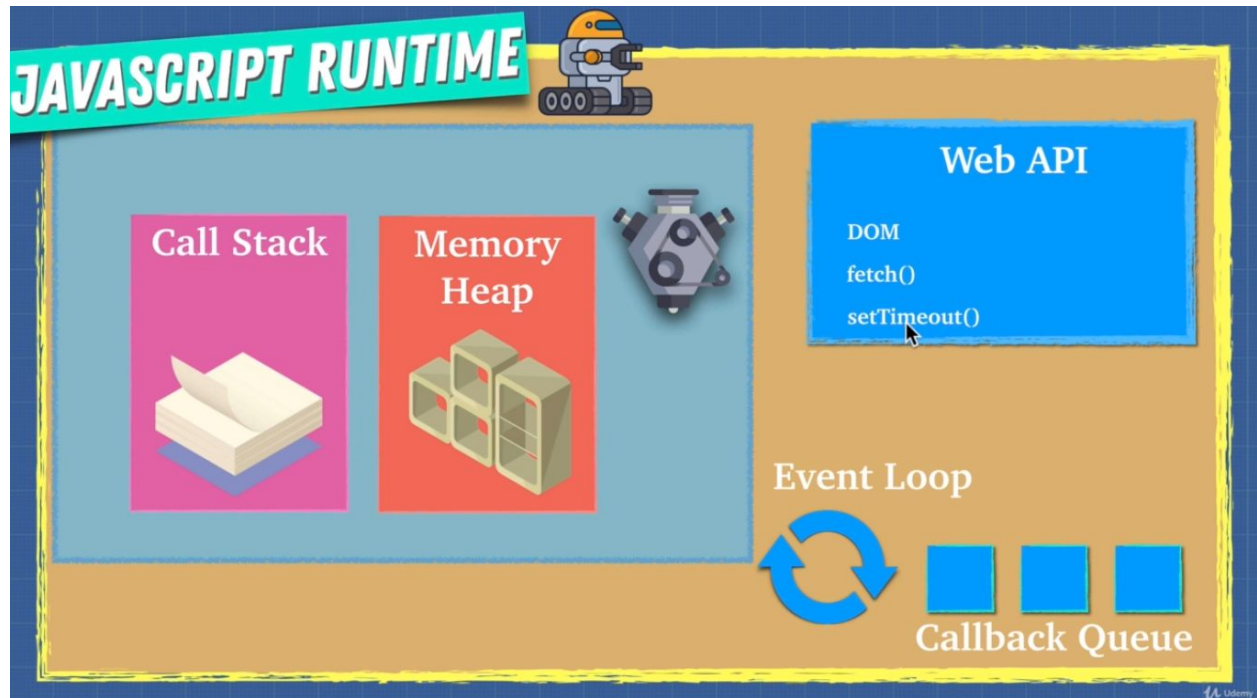
Asynchronous JavaScript



- APIs
- Async
- Data that we don't have yet
- Promises

Web APIs

- Runs in parallel with the JS Engine inside the JS Runtime



- Browsers are written in low-level languages for performance
 - Browsers have a JS API (Web API) to allow us to call and use browser-based functions within JS
- Storage
 - IndexedDB
- SetTimeout
 - When called, it is removed from the call stack and sent to the Web API
- Event Loop
 - Runs constantly and checks call stack
 - When a Web API function completes, it is added to the callback queue
 - When the call stack is empty, the **event loop** moves things from the callback queue to the call stack

[A visual example](#) of the Call Stack and the Web API in action

- DOM
- fetch
- Asynchronous

Callbacks

A **callback** function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Here is a quick example:

```
function greeting(name) {  
  alert('Hello ' + name);  
}  
function processUserInput(callback) {  
  var name = prompt('Please enter your name.');
```

```
  callback(name);  
}  
processUserInput(greeting);
```

The above example is a **synchronous callback**, as it is executed immediately.

Note, however, that **callbacks** are often used to continue code execution after an asynchronous operation has completed — these are called **asynchronous callbacks**. A good example is the **callback** functions executed inside a **.then()** block chained onto the end of a **promise** after that **promise** fulfills or rejects. This structure is used in many modern web APIs, such as **fetch()**.

[\[https://developer.mozilla.org/en-US/docs/Glossary/Callback_function\]](https://developer.mozilla.org/en-US/docs/Glossary/Callback_function)

Callback examples

```
el.addEventListener("click", submitForm);
```

```
//Callback pyramid of doom
```

```
movePlayer(100, "Left", function(){  
  movePlayer(400, "Left", function(){  
    movePlayer(10, "Right", function(){  
      movePlayer(330, "Left", function(){
```



```

    })
  })
})

```

```

grabTweets('twitter/andreineagoie', (error, andreiTweets) => {
  if(error){
    throw Error;
  }
  displayTweets(andreiTweets)
  grabTweets('twitter/elonmusk', (error, elonTweets) => {
    if(error){
      throw Error;
    }
    displayTweets(elonTweets)
    grabTweets('twitter/vitalikbuterin', (error, vitalikTweets) =>
{
      if(error){
        throw Error;
      }
      displayTweets(vitalikTweets)
    })
  })
})

```

Promises

- Before **promises**, there were **callbacks**.
- A **promise** is an **object** that may produce a single value some time in the future
 - Resolved
 - Rejected
 - Pending
- **.catch()** can be used to catch any errors thrown in front of the **.catch()** in a promise chain

An example of Promise

```

//Promise
movePlayer(100, 'Left')

```

```
.then(() => movePlayer(400, "Left"))
.then(() => movePlayer(10, "Right"))
.then(() => movePlayer(330, "Left"))
```

```
const promise = new Promise((resolve, reject) => {
  if(true) {
    resolve("stuff worked");
  }else{
    reject("Error, it broke")
  }
})
promise.then(result => console.log(result)) //stuff worked

// promise chain
promise
  .then(result => result + "!")
  .then(result2 => console.log(result2)) //stuff worked!
promise
  .then(result => result + "!")
  .then(result2 => {
    throw Error
    console.log(result2)
  })
  .catch(() => console.log('Error!')) // Error!
```

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {
```

```
    alert(result); // 4
    return result * 2;

});
```

- The whole thing works, because a call to **promise.then** returns a **promise**, so that we can call the next **.then** on it.
- When a handler returns a value, it becomes the result of that **promise**, so the next **.then** is called with it.

[\[https://javascript.info/promise-chaining\]](https://javascript.info/promise-chaining)

Promise.all()

- **Promise.all()** takes an array of promises and returns their values when all promises are resolved, or throws an error.

```
const promise1 = new Promise((resolve, reject) => {
  if (true){
    resolve('promise1')
  }else{
    reject('Error, it broke')
  }
})

const promise2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'promise2')
})

const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'promise3')
})

const promise4 = new Promise((resolve, reject) => {
  setTimeout(resolve, 5000, 'promise4')
})

Promise.all([promise1, promise2, promise3, promise4]).then((values) => {
  console.log(values) /*after 5000ms, logs ["promise1", "promise2",
  "promise3", "promise4"] */
})
```

An example use for Promise.all()

```
const urls = [
  "https://jsonplaceholder.typicode.com/users",
  "https://jsonplaceholder.typicode.com/posts",
  "https://jsonplaceholder.typicode.com/albums",
];

Promise.all(
  urls.map(url => {
    return fetch(url).then(resp => resp.json());
  })
).then(results => {
  results.map(result => console.log(result))
});
```

Async/Await

- **async** and **await** are syntactic sugar on top of **Promises**

A normal Promise-based fetch()

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(resp => resp.json())
  .then(console.log);
```

An async/await-styled fetch()

```
async function fetchUsers() {
  const response = await
  fetch("https://jsonplaceholder.typicode.com/users");
  const data = await response.json();
  console.log(data);
}

fetchUsers()
```

An async/await-styled Promise.all() with try/catch

```

const urls = [
  "https://jsonplaceholder.typicode.com/users",
  "https://jsonplaceholder.typicode.com/posts",
  "https://jsonplaceholder.typicode.com/albums",
];

const getData = async function() {
  try {
    const [users, posts, albums] = await Promise.all(
      urls.map(url => {
        return fetch(url).then(resp => resp.json());
      })
    );
    console.log(users);
    console.log(posts);
    console.log(albums);
  } catch (err) {
    console.log(err);
  }
};
getData();

```

`.finally()`

- **`.finally()`** runs code after a **promise** whether it succeeds or fails

```

const urls = [
  "https://jsonplaceholder.typicode.com/users",
  "https://jsonplaceholder.typicode.com/posts",
  "https://jsonplaceholder.typicode.com/albums",
];

Promise.all(
  urls.map(url => {
    return fetch(url).then(resp => resp.json());
  })
)
  .then(results => {
    console.log(results[0]);
    console.log(results[1]);
    console.log(results[2]);
  })
  .catch(err => console.log(err))

```

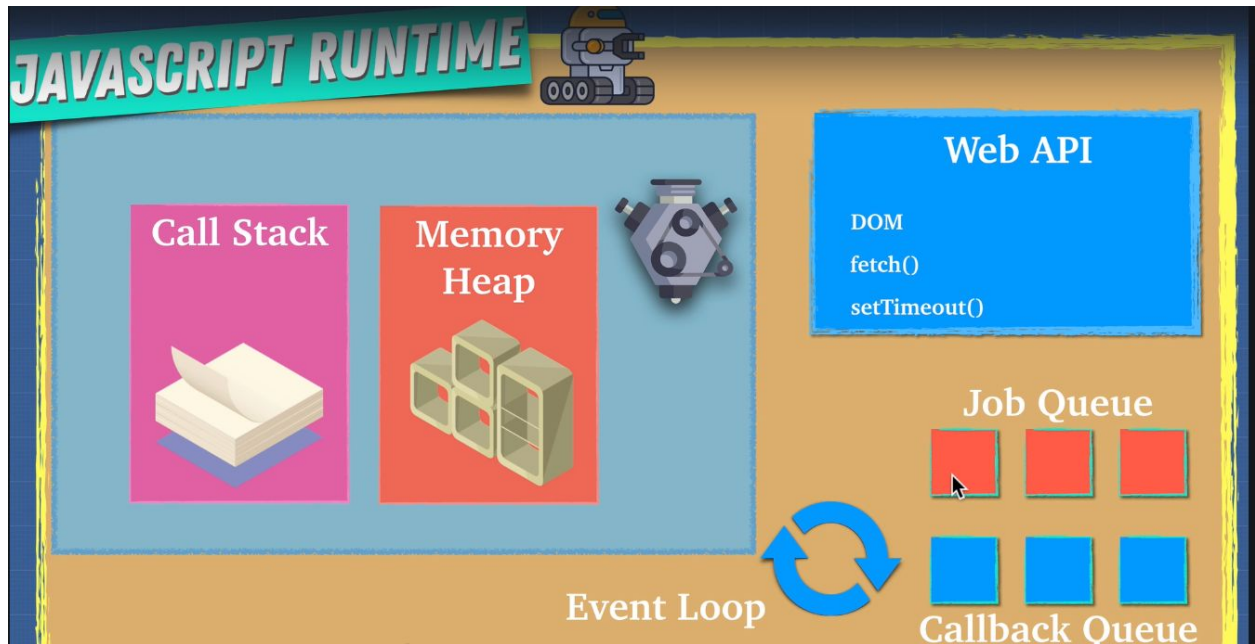
```
.finally(() => console.log("finally"));
```

for await of

```
const urls = [
  "https://jsonplaceholder.typicode.com/users",
  "https://jsonplaceholder.typicode.com/posts",
  "https://jsonplaceholder.typicode.com/albums",
];
const getData2 = async function() {
  const arrayOfPromises = urls.map((url) => fetch(url));
  for await (let request of arrayOfPromises) {
    const data = await request.json();
    console.log(data);
  }
}
```

Microtask Queue (Job Queue)

- The **event loop** checks the **job queue** before the **callback queue**, the **job queue** has *higher priority*.
- The **job queue** is for **Promises**



```
main.js  saved
1 //Callback Queue - Task Queue
2 setTimeout(()=>{console.log('1', 'is the loneliest number')}, 0)
3 setTimeout(()=>{console.log('2', 'can be as bad as one')}, 10)
4
5 //2 Job Queue - Microtask Queue
6 Promise.resolve('hi').then((data)=> console.log('2', data))
7
8 //3
9 console.log('3', 'is a crowd')
```

Native Browser JavaScript

```
3 is a crowd
2 hi
=> undefined
1 is the loneliest number
2 can be as bad as one
```

- Some older browsers may not have a **job queue** yet, so do not rely on it too much.

Task Queue (Callback Queue)

Event Loop

- Runs constantly and checks call stack
- When a Web API function completes, it is added to the callback queue

- When the call stack is empty, the **event loop** moves things from the callback queue to the call stack
 - [A visual example](#) of the Call Stack and the Web API in action
-

Parallel, Sequence, and Race

Parallel

Parallel means to run multiple **promises** in **parallel**, beginning together and returning them all when they are all finished.

```
const promisify = (item, delay) =>
  new Promise(resolve => setTimeout(() => resolve(item), delay));

// functions that return promises
const a = () => promisify("a", 100);
const b = () => promisify("b", 5000);
const c = () => promisify("c", 3000);
console.log(a,b,c) // [Function] [Function] [Function]
console.log(a(),b(),c()) // Promise {} Promise {} Promise {}
async function parallel() {
  const promises = [a(), b(), c()];
  const [output1, output2, output3] = await Promise.all(promises);
  return `parallel is done: ${output1} ${output2} ${output3}`;
}
parallel().then(console.log) // parallel is done: a b c
```

Sequence

Sequence means that each **promise** will *wait for earlier promises to resolve* before beginning.

```
const promisify = (item, delay) =>
  new Promise(resolve => setTimeout(() => resolve(item), delay));

// functions that return promises
const a = () => promisify("a", 100);
const b = () => promisify("b", 5000);
const c = () => promisify("c", 3000);
console.log(a,b,c) // [Function] [Function] [Function]
console.log(a(),b(),c()) // Promise {} Promise {} Promise {}
async function sequence() {
  const output1 = await a();
  const output2 = await b();
  const output3 = await c();
  return `sequence is done: ${output1} ${output2} ${output3}`;
}
```



```
sequence().then(console.log) // sequence is done: a b c
```

Race

Race means that *the first promise to resolve will be returned* and the others will be ignored.

```
const promisify = (item, delay) =>
  new Promise(resolve => setTimeout(() => resolve(item), delay));

// functions that return promises
const a = () => promisify("a", 100);
const b = () => promisify("b", 5000);
const c = () => promisify("c", 3000);
console.log(a,b,c) // [Function] [Function] [Function]
console.log(a(),b(),c()) // Promise {} Promise {} Promise {}
async function race() {
  const promises = [a(), b(), c()]
  const output1 = await Promise.race(promises);
  return `race is done: ${output1}`
}

race().then(console.log) // race is done: a
```

Threads, Concurrency, and Parallelism

An example of multithreading/parallelism in Node.js, generates a new thread

```
const {spawn} = require('child_process')

spawn('git', ['stuff'])
```

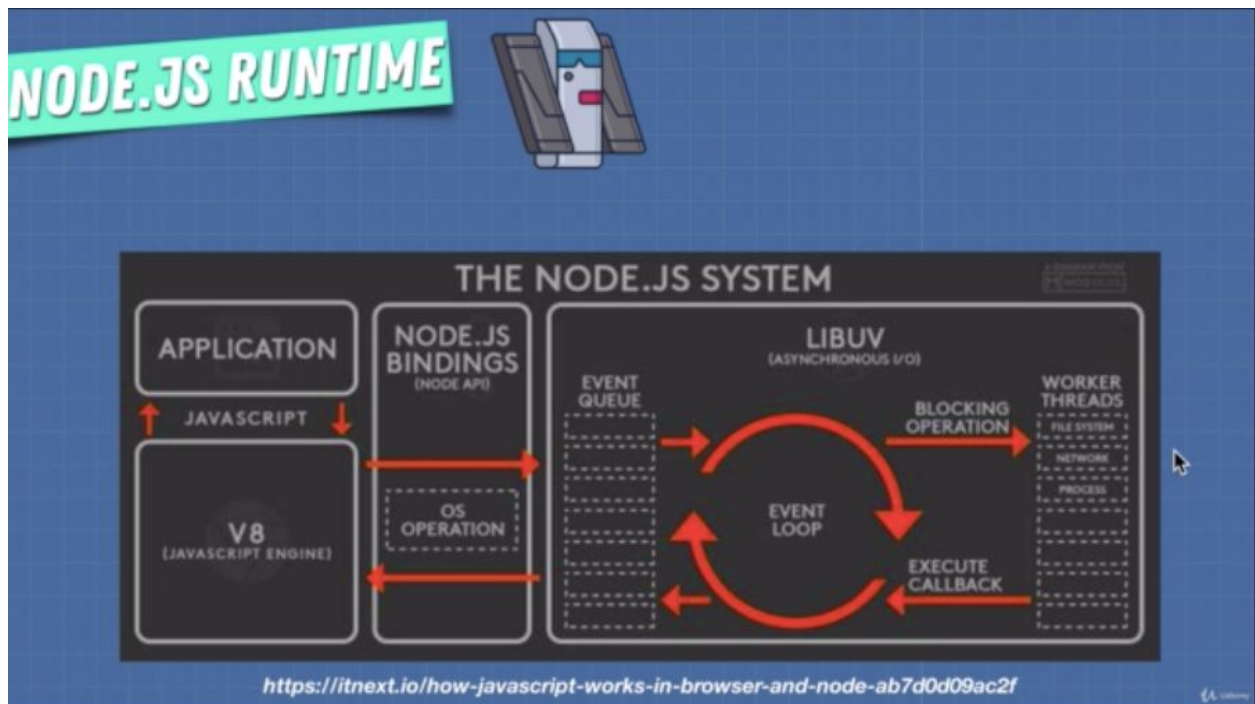
- **Concurrency** means multiple computations are happening at the same time.
[<https://medium.com/@onejohi/concurrency-in-javascript-f5bb387708d8>]

```

@aneagoie/gandreineagoie - zerotomastery.io
run share
main.js saved
1  Concurrency                               Concurrency + parallelism
2  (Single-Core CPU)                         (Multi-Core CPU)
3  |_____|                                   |_____|
4  |th1|                                     |th1|th2|
5  |_____|                                   |_____|
6  |_____|_____|                           |_____|
7  |_____|th2|                             |_____|th2|
8  |_____|_____|                           |_____|
9  |th1|                                     |th1|
10 |_____|_____|                           |_____|
11 |_____|th2|                             |_____|th2|

```

Web Workers



Error Handling

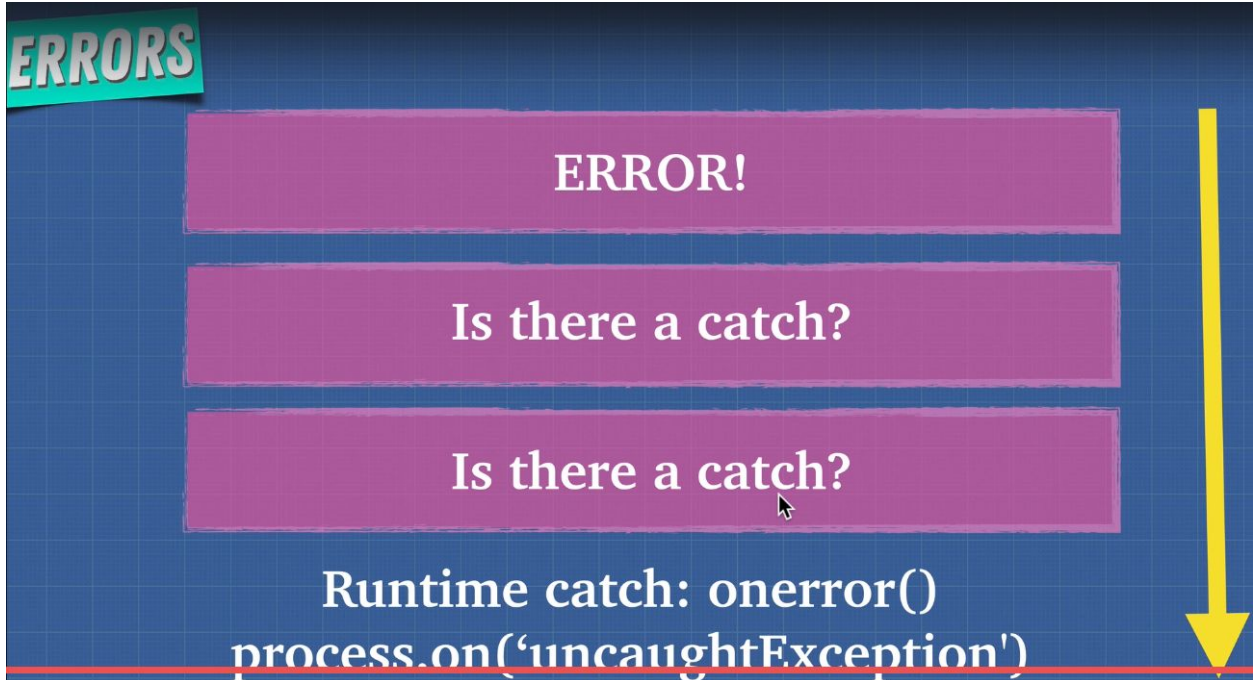
- Errors are used with the keyword **throw**
- The **name** property refers to the general class of **Error**
- The **message** property generally provides a more succinct message than one would get by converting the error object to a string.
- The **stack** property traces the **Error**'s position in the call-stack back to the global context which is typically `<anonymous>`

```
const c = new Error('again?')
console.log(c.name) // Error
console.log(c.message) // again?
console.log(c.stack) // Error: again?↵ at <anonymous>:1:11
```

- The below example shows the **stack** property at work.
- First, the class of **Error** is given “Error”,
- Then the **message**, “what??”,
- Finally the **stack** trace “at a (<anonymous>:2:13)↵at <anonymous>:5:1:5:1”, meaning the first context is the function “a” and the second context is anonymous (global context)

```
function a() {
  const b = new Error('what??')
  return b
}
a().stack // Error: what??↵at a (<anonymous>:2:13)↵at <anonymous>:5:1:5:1
```

- **Error types**
 - **Error**, general error
 - **SyntaxError**, e.g. a misplaced comma
 - **ReferenceError**, e.g. undefined variable
- When an **Error** is thrown, it searches up the call-stack for a **catch**.
- If no **catch** is found, the **runtime** handles it
- In the browser, it is **onerror()**
- In Node.js, it is **process.on('uncaughtException')**



try/catch/finally

- **try/catch** blocks can be used to handle *synchronous* code

```
function fail() {  
  try {  
    console.log("this works");  
  } catch (error) {  
    console.log("oops, error");  
  }  
}
```

```
function fail() {  
  try {  
    consol.log("this works"); //misspelled "console" as "consol"  
  } catch (error) {  
    console.log("oops, error"); //throws an error, "oops, error ReferenceError"  
  }  
}
```

```
function fail() {  
  try {  
    console.log("this works"); //successfully logs because it is above the error  
    throw new Error('whoops!') //throws an error, looks for a catch  
  } catch (error) {
```

```
    console.log("We have an error", error); // "We have an error Error: whoops!"
  }
}
```

finally

```
function fail() {
  try {
    console.log("this works"); //successfully logs because it is above the error
    throw new Error("whoops!"); //throws an error, looks for a catch
  } catch (error) {
    console.log("We have an error", error); // "We have an error Error: whoops!"
  } finally {
    console.log("final"); // logs "final" regardless of try/catch success/fail
  }
}
```

Errors in asynchronous code

- **Promises** use the `.catch()` method to catch errors
- Without a `.catch()` a **Promise can fail silently**, meaning no **Error** is shown despite there being one

*This Promise **fails silently**, it does not produce an error message*

```
Promise.resolve("asyncfail")
  .then(response => {
    throw new Error("#1 fail");
    return response;
  })
  .then(response => {
    console.log(response);
  });
```

*This promise fails and **does** produce an error via `.catch()`*

```
Promise.resolve("asyncfail")
  .then(response => {
    throw new Error("#1 fail");
    return response;
  })
  .then(response => {
    console.log(response);
  })
  .catch(error => {
    console.log(error);
  });
```

```
.catch(error => console.log(error));
```

A **.catch()** can return and use a **.then()** to further process the return from the **.catch()**

```
Promise.resolve("asyncfail")
  .then(response => {
    throw new Error("#1 fail");
    return response;
  })
  .then(response => {
    console.log(response);
  })
  .catch(error => {
    return error;
  })
  .then(response => {
    console.log(response.message); // #1 fail
  });
```

.catch() can be chained. In this example, the “#1 fail” **Error** occurs, which is caught in the first **.catch()**, the first **.catch()** throws a new **Error** which is then caught by the second **.catch()** which then console.logs “final error”

```
Promise.resolve("asyncfail")
  .then(response => {
    throw new Error("#1 fail");
    return response;
  })
  .then(response => {
    console.log(response);
  })
  .catch(error => {
    throw new Error("#2");
  })
  .then(response => {
    console.log(response.message);
  })
  .catch(error => {
    console.log("final error"); // final error
  });
```

- With nested **Promises**, it is important to have a **.catch()** nested with the **Promise** as well, otherwise, the errors are not handled correctly

*Nested **Promise**'s error is not handled correctly*

```
Promise.resolve("asyncfail")
  .then(response => {
    Promise.resolve().then(() => {
      throw new Error("#3 fail");
    }); //should have .catch() here
    return 5;
  })
  .then(response => {
    console.log(response);
  })
  .then(response => {
    console.log(response.message);
  })
  .catch(error => {
    console.log("final error");
  });
```

async/await Error Handling

- While **async/await** is *asynchronous*, it looks like and is handled like *synchronous* code. This means that **try/catch** blocks work for **async/await**.

```
(async function() {
  try {
    await Promise.resolve("#1 yeah");
    await Promise.reject("#2 no");
  } catch (error) {
    console.log(error);
  }
})();
```

Custom Errors

- **Error** is a class and can be *extended* with the **extends** keyword to create **custom Errors**.
- This can be useful for creating **secure Errors** *by not displaying too much information*.

```
class DatabaseError extends Error {
  constructor(message) {
    super(message);
    this.name = "DatabaseError";
  }
}

class PermissionError extends Error {
  constructor(message) {
    super(message);
    this.name = "PermissionError";
  }
}

class AuthenticationError extends Error {
  constructor(message) {
    super(message);
    this.name = "AuthenticationError";
    this.favoriteSnack = "grapes";
  }
}

const a = new AuthenticationError("snack");
console.log(a.favoriteSnack); // grapes
throw new AuthenticationError("oops"); // AuthenticationError: oops
```

Destructuring

Destructuring allows us to remove values from the enclosing structure of arrays and objects, you can think of it as removing the `[]` or `{}`.

Array destructuring

```
const sampleArray = [1,2,3]
const [one, two, three] = sampleArray
console.log(one, two, three) // 1 2 3
```


Object destructuring

```
const sampleObject = {one: 1, two: 2, three: 3}
const {one, two, three} = sampleObject
console.log(one, two, three) // 1 2 3
```

Renaming destructured object properties

We can **rename** *object properties* while destructuring objects, but not arrays, of course, as arrays have *elements* with *indices* and not *property names (keys)*.

Here the property name `one` is changed to `newName` during destructuring by appending `: newName` to `one`

```
const sampleObject = {one: 1, two: 2, three: 3}
const {one: newName, two, three} = sampleObject
console.log(newName, two, three) // 1 2 3
console.log(one) // Uncaught ReferenceError: one is not defined
```

JavaScript's "...” operator

This section needs some clarification and additional examples

Spread Operator

A spread operator *spreads*, or separates, the values contained in an object or array.

```
var myName = ["Marina" , "Magdy" , "Shafiq"];
var newArr = [...myName , "FrontEnd" , 24];
console.log(newArr) ; // ["Marina" , "Magdy" , "Shafiq" , "FrontEnd" , 24 ] ;
```

[Source](#)

Here the spread operator separates the values of the array into the parameter

```
const array = [1,2,3,4,5]
function sum(a,b,c,d,e){
  return a+b+c+d+e
}
sum(...array) // 15
```

Rest Pattern

The **rest pattern** is used to *collect extra values* beyond those explicitly defined.

Rest Parameter

A **rest parameter** is used to collect *the rest of the arguments* into a single parameter as an array. A **rest parameter** must be the last parameter.

Rest Destructuring

Rest destructuring is very similar to the rest parameter in that it is used to collect the **rest** of the values into an array.

A rest parameter being used in deconstructing an array: `const [one, ...rest] = array`

```
const array = [1,2,3]
const [one, ...rest] = array
one // 1
rest // [2,3]
```

An object rest destructuring

```
const animals = {
  tiger: 23,
  lion: 5,
  monkey: 2
}

const {tiger, ...rest} = animals

function objectSpread(p1,p2){
  console.log(p1)
  console.log(p2)
}

objectSpread(tiger, rest)
// 23
// {lion: 5, monkey: 2}
```

Spread and Rest together

```
function makeArray(arg1, ...args){ //a rest parameter collects extra values
  console.log(arg1) // 1
  console.log(args) // [2,3,4,5,6,7]
  console.log(arg1, ...args) // 1,2,3,4,5,6,7 via spread operator
  return [arg1, ...args] // [1,2,3,4,5,6,7] via spread operator
}
makeArray(1, 2, 3, 4, 5, 6, 7) // all arguments besides the first are //
collected by ...args
```

Modules in JavaScript

Modules in JavaScript allows us to *containerize* our JavaScript. It helps us to interact with other code and *avoid polluting the global namespace*.

Module Pros and Cons

- Pros
 - Limits global namespace pollution
 - API remaining the same, work on **modules** can be done independently of each other
 - Cons
 - Obscures dependencies
-

IIFE

- Before **modules**, JavaScript used **IIFE** to create a *module pattern* to *containerize* our code.
- An **IIFE** would create a **function scope** for the functions and variables to live inside of where they would be **private**. This would *prevent pollution of the global namespace*.

- By returning functions/variables from the **IIFE** in an object, we create an **interface** for interacting with the code inside of the **IIFE function scope**.
- This method of *containerizing* our code is called the **Revealing Module Pattern**.

Functions and variables become private to the fightModule function scope

```
//IIFE
//Module Pattern
var fightModule = (function () {
  var harry = 'potter'
  var voldemort = 'He who must not be named'

  function fight(char1, char2) {
    var attack1 = Math.floor(Math.random() * char1.length);
    var attack2 = Math.floor(Math.random() * char2.length);
    return attack1 > attack2 ? `${char1} wins` : `${char2} wins`;
  }
  return {
    fight: fight
  }
})();
```

Creates an interface

```
return {
  fight: fight
}
```

Native ES Modules

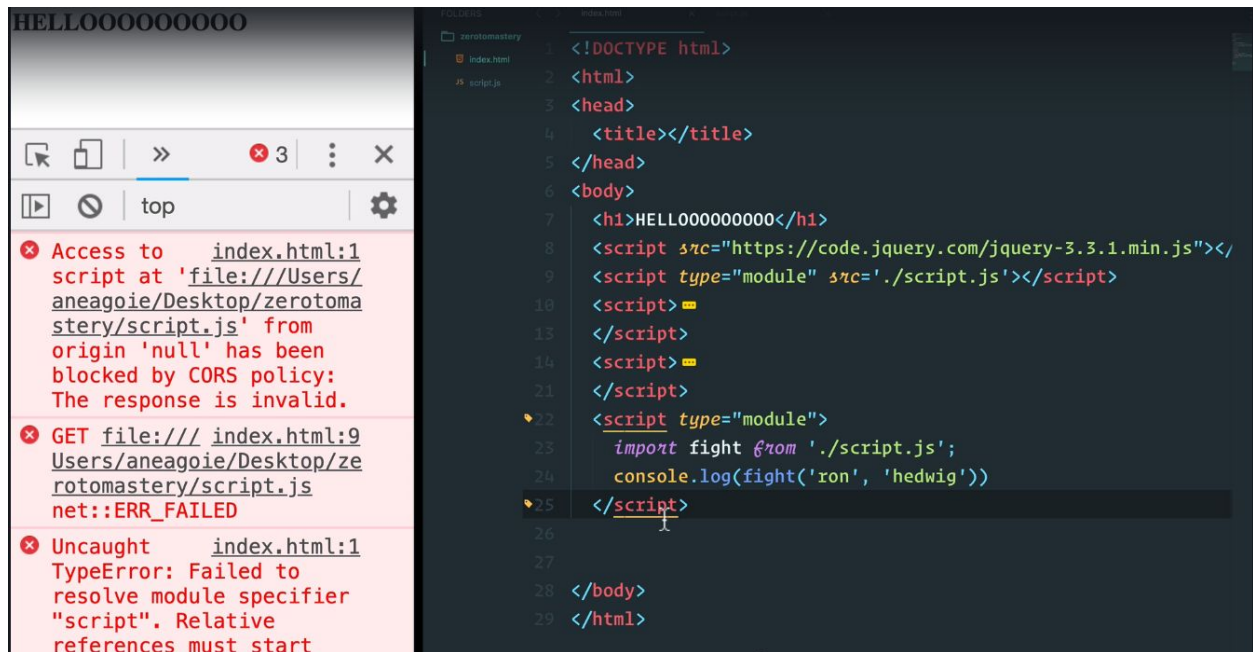
- Brought to client-side JavaScript in ES6
- Coming soon to Node.js
- Provides native module functionality in the browser

```
//Native ES6 Modules
import module1 from 'module1' //{fight};
import module2 from 'module2' //importedFunc2

export function jump() {
}
```

- When using modules in **script** tags, the **type** attribute should be **module**.

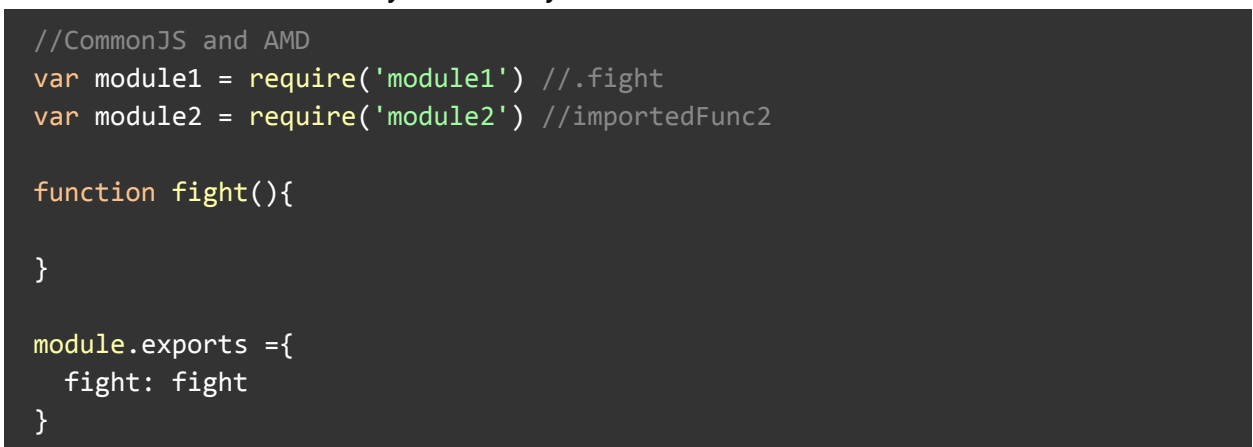
- When using **modules**, CORS policy applies and they must be served from a server proxy, e.g. [live-server](#)



Default and named exports

CommonJS

- Created for server-side coding
- Still used by Node.js
- **Modules** are loaded *synchronously*



- **Browserify** bundles all scripts in a single `bundle.js` while understanding the module syntax and maintaining separation

UMD

UMD (Universal Module Definition) is essentially an if/else that determines whether to use **AMD** or **CommonJS**

AMD

AMD (Asynchronous Module Definition) is an alternative to **CommonJS** for creating modules in JavaScript

```
//AMD
define(["module1", "module2"], function(module1Import, module2Import) {
  var module1 = module1Import; //.fight
  var module2 = module2Import; //importedFunc2

  function dance() {}
  return {
    dance: dance,
  };
});
```

Template Literals

Template literals let you do more with **strings**.

```
const templateLiteral = `I am a template literal`
```

Multi-line strings

Template literals let you write multi-line strings

```
const templateLiteral = `I am a very long multi-line
template literal`
console.log(templateLiteral) // I am a very long multi-line
                             //template literal
```

Avoid escaping characters

Template literals can help you avoid some annoying features of normal strings.

This line of code will throw an error because the single-quotes used to enclose the string are being interrupted by the apostrophe used in the word "I'm"

```
const annoyingString = 'I'm really annoying'
```

You would need to write it this way:

```
const annoyingString = "I'm really annoying"
```

The same is true of strings using double-quotes

```
const annoyingString2 = "don't try to quote anyone here: "He said he would be there""
```

You would need to write it this way:

```
const annoyingString2 = "don't try to quote anyone here: 'He said he would be there'"
```

Thankfully, **template literals** help us solve this problem:

```
const lovelyTemplateLiterals = `It's so easy to use apostrophes and quote people`, he said`
```

Variable Insertion

Template literals are great for inserting the value of a variable into a string using the `${}` pattern

```
const myVariable = 10
const insertedVariable = `I can see ${myVariable} elephants` // "I can see 10 elephants"
```

Avoid String concatenation

With traditional strings, we had to insert variables and concatenate (add together) strings this way

```
const myVariable = "String2"
const stringCat = "String1 " + myVariable + " String3" // "String1 String2 String3"
```

```
const firstName = "foo"
const lastName = "bar"
const fullName = firstName + " placeholder " + lastName // "foo placeholder //bar"
```

Semicolons

A semi-colon `;` is used to end a line of code.

You may have noticed that there are not always semi-colons written at the end of a line of code. That's because the JS interpreter is able to intelligently determine where a semi-colon should be placed. *However*, it does not always put the semi-colon where you might expect or want.

Iterators

Generator

Symbols

Map

A `Map` is similar to an `Object` in most cases. The primary difference is that a `Map` can use any type as a key.

	Map	Object
Accidental keys	A Map does not contain any keys by default. It only contains what is explicitly put into it.	An Object has a prototype, so it contains default keys that could collide with your own keys if you're not careful. Note: As of ES5, this can be bypassed by using <code>Object.create(null)</code> , but this is seldom done.
Key types	A Map's keys can be any value (including functions, objects, or any primitive).	The keys of an Object must be either a <code>String</code> or a <code>Symbol</code> .
Key order	The keys in Map are ordered. Thus, when iterating over it, a Map object returns keys in order of insertion.	The keys of an Object are not ordered. Note: Since ECMAScript 2015, objects <i>do</i> preserve

		creation order for string and <code>Symbol</code> keys. In JavaScript engines that comply with the ECMAScript 2015 spec, iterating over an object with only string keys will yield the keys in order of insertion.
Size	The number of items in a Map is easily retrieved from its <code>size</code> property	The number of items in an Object must be determined manually
Iteration	A Map is an <code>iterable</code> , so it can be directly iterated	Iterating over an Object requires obtaining its keys in some fashion and iterating over them
Performance	Performs better in scenarios involving frequent additions and removals of key-value pairs	Not optimized for frequent additions and removals of key-value pairs

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

WeakMap

- A `WeakMap` is essentially a `Map`, but the keys are more ephemeral.
- Keys of `WeakMaps` are of the type `Object` only. `Primitive data types` as keys are not allowed (e.g. a `Symbol` can't be a `WeakMap` key)

By contrast, native `WeakMaps` hold "weak" references to key objects, which means that they do not prevent garbage collection in case there would be no other reference to the key object. This also avoids preventing garbage collection of values in the map. Native `WeakMaps` can be particularly useful constructs when mapping keys to information about the key that is valuable only if the key has not been garbage collected.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap

Set

Set objects are collections of values. You can iterate through the elements of a set in insertion order. A value in the Set **may only occur once**; it is unique in the Set's collection.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

This document has been created as a supplement to the Udemy course [Advanced JavaScript Concepts](#) by [Andrei Neagoie](#). In addition to the information from Andrei Neagoie, information and images have been compiled from many different sources around the internet. I do not claim ownership of anything in this document and seek to correctly attribute all information. When no source is given, it can be assumed that the information comes from Andrei Neagoie's course. Compiled by [Bryan Windsor](#). 