

JavaScript advanced

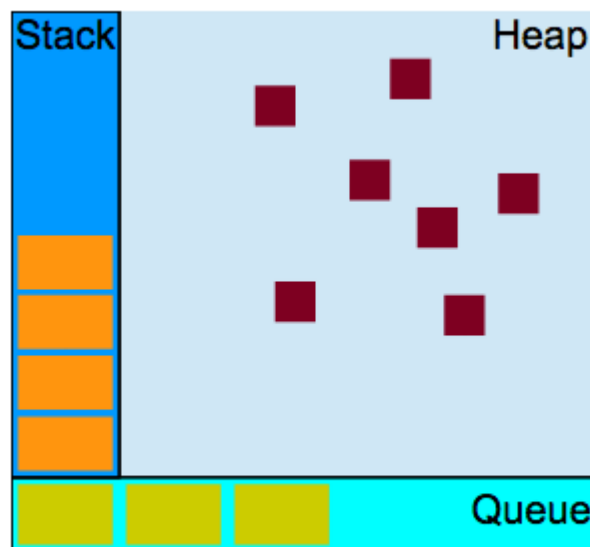
Resources: <https://github.com/ssakethraj/33-js-concepts>
<https://javascript.info/>
<https://www.javascripture.com>
<https://google.github.io/styleguide/jsguide.html>

Call Stack

The tooling that is required to build a modern web app is overwhelming with Webpack, Babel, ESLint, Mocha, Karma, Grunt...

V8, chrome's Runtime

Javascript is a single threaded single concurrent language, meaning it can handle one task at a time or a piece of code at a time. It has a single **call stack (Last In, First Out (LIFO))** which along with other parts like heap, queue constitutes the Javascript Concurrency Model (implemented inside of V8).



Visual Representation of JS Model(credits)

Heap is where the memory is (memory allocation is done)
Web Apis (https ... other)

Java Script one time at time single thread (have one call stack)

What happens when things are slow ? (blocking)

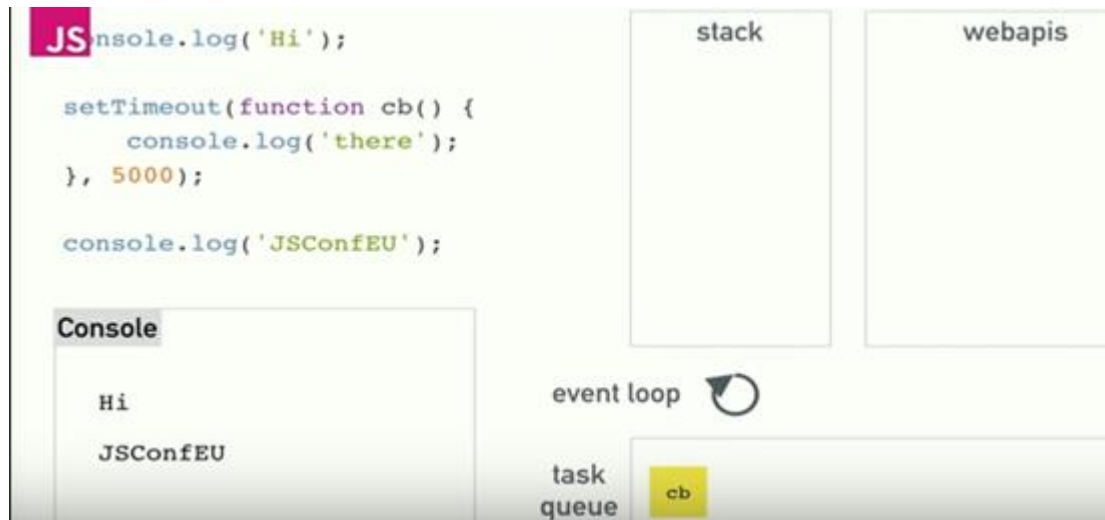
-> things that need time to execute other commands are waiting

-> Solution is async callbacks

Async Callback in the Call Stack

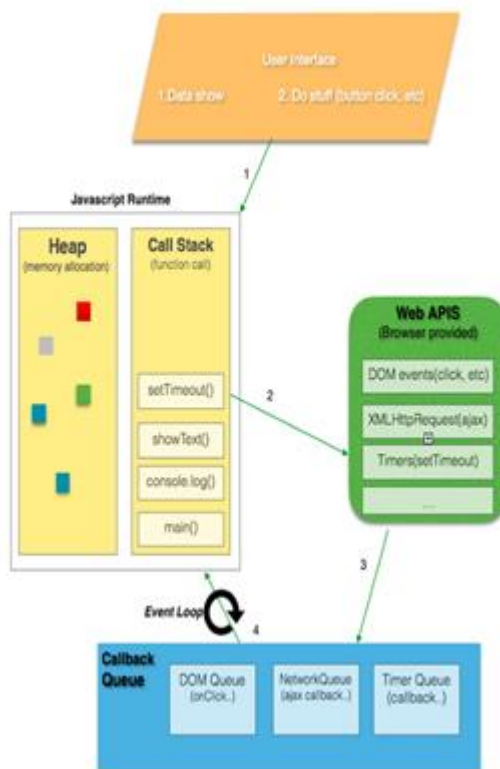
-> method (code) is passed in **webapis** when is done then goes to TASK QUEUE

-> event loop checks if stack is empty it returns to stack first method (code) from task queue



The decoupling of the caller from the response allows for the JavaScript runtime to do other things while waiting for your asynchronous operation to complete and their callbacks to fire. This is where **browser APIs kick in and call its APIs, which are basically threads created by browser implemented in C++ to handle async events like DOM events, http request, setTimeout,** etc. (After knowing this, in **Angular 2, Zones** are used which monkey patches these APIs to cause runtime change detection, which I can get a picture now, how they were able to achieve it.)

The JavaScript engine (which is found in a hosting environment like the browser), is a single-threaded interpreter comprising of a heap and a single call stack. The browser provides web APIs like the DOM, AJAX, and Timers.



When the JavaScript engine first encounters your script, it creates a global execution context and pushes it to the current execution stack. Whenever the engine finds a function invocation, it creates a new execution context for that function and pushes it to the top of the stack.

Types of Execution Context

There are three types of execution context in JavaScript.

- **Global Execution Context** — This is the default or base execution context. The code that is not inside any function is in the global execution context. It performs two things: it creates a global object which is a window object (in the case of browsers) and sets the value of `this` to equal to the global object. There can only be one global execution context in a program.
- **Functional Execution Context** — Every time a function is invoked, a brand new execution context is created for that function. Each function has its own execution context, but it's created when the function is invoked or called. There can be any number of function execution contexts. Whenever a new execution context is created, it goes through a series of steps in a defined order which I will discuss later in this article.
- **Eval Function Execution Context** — Code executed inside an `eval` function also gets its own execution context, but as `eval` isn't usually used by JavaScript developers, so I will not discuss it here.



The execution context is created during the creation phase. Following things happen during the creation phase:

1. **LexicalEnvironment** component is created.
2. **VariableEnvironment** component is created.

A *lexical environment* is a structure that holds **identifier-variable mapping**. (here **identifier** refers to the name of variables/functions, and the **variable** is the reference to actual object [including function object and array object] or primitive value)

```
var a = 20;
var b = 40;

function foo() {
  console.log('bar');
}
```

So the lexical environment for the above snippet looks like this:

```
lexicalEnvironment = {
  a: 20,
  b: 40,
  foo: <ref. to foo function>
}
```

Each Lexical Environment has three components:

1. Environment Record
2. Reference to the outer environment,
3. This binding.

The environment record is the place where the variable and function declarations are stored inside the lexical environment.

There are also two types of *environment record* :

- **Declarative environment record** — As its name suggests stores variable and function declarations. The lexical environment for function code contains a declarative environment record.
- **Object environment record** — The lexical environment for global code contains a object environment record. Apart from variable and function declarations, the object environment record also stores a global binding object (window object in browsers). So for each of binding object's property (in case of browsers, it contains properties and methods provided by browser to the window object), a new entry is created in the record.

For the **function code**, the *environment record* also contains an `arguments` object that contains the mapping between indexes and arguments passed to the function and the *length(number)* of the arguments passed into the function

The **reference to the outer environment** means it has access to its outer lexical environment. That means that the JavaScript engine can look for variables inside the outer environment if they are not found in the current lexical environment.

In this component, the value of `this` is determined or set.

In the global execution context, the value of `this` refers to the global object. (in browsers, `this` refers to the Window Object)

Variable Environment: It's also a Lexical Environment whose `EnvironmentRecord` holds bindings created by *VariableStatements* within this execution context.

Note — As you might have noticed that the `let` and `const` defined variables do not have any value associated with them during the creation phase, but `var` defined variables are set to `undefined` .

This is because, during the creation phase, the code is scanned for variable and function declarations, while the function declaration is stored in its entirety in the environment, the variables are initially set to `undefined` (in case of `var`) or remain uninitialized (in case of `let` and `const`).

From: <https://blog.bitsrc.io/understanding-execution-context-and-execution-stack-in-javascript-1c9ea8642dd0>

The Event Loop

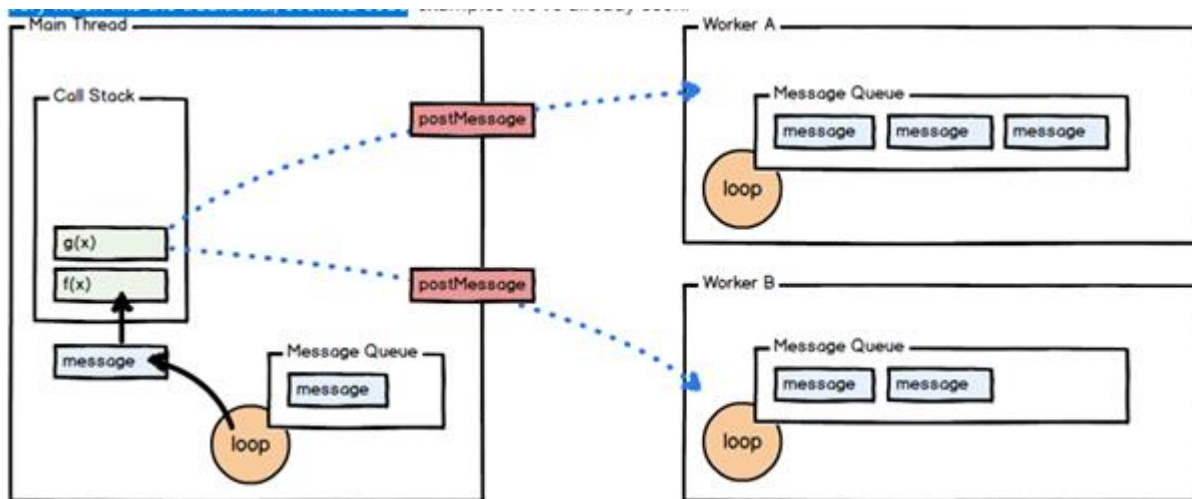
The decoupling of the caller from the response allows for the JavaScript runtime to do other things while waiting for your asynchronous operation to complete and their callbacks to fire.

JavaScript runtimes contain a message queue which stores a list of messages to be processed and their associated callback functions. These messages are queued in response to external events (such as a mouse being clicked or receiving the response to an HTTP request) given a callback function has been provided

Web Workers

Using Web Workers enables you to offload an expensive operation to a separate thread of execution, freeing up the main thread to do other things. The worker includes a separate message queue, event loop, and memory space independent from the original thread that instantiated it. Communication between the worker and the main thread is done via message passing, which looks very much like the traditional, evented code

Closures and event block



JavaScript's support for closures allow you to register callbacks that, when executed, maintain access to the environment in which they were created even though the execution of the callback creates a new call stack entirely. This is particularly of interest knowing that our callbacks are called as part of a different message than the one in which they were created. Consider the following example:

How numbers are encoded in JavaScript

All numbers in JavaScript are floating point. This blog post explains how those floating point numbers are represented internally in 64 bit binary

- How numbers are encoded in JavaScript — Dr. Axel Rauschmayer
- ? What You Need to Know About JavaScript Number Type — Max Wizard K
- ? What Every JavaScript Developer Should Know About Floating Point Numbers — Chewxy
- ? The Secret Life of JavaScript Primitives — Angus Croll
- ? Primitive Types — Flow
- ? (Not) Everything in JavaScript is an Object - Daniel Li
- ? JavaScript data types and data structures - MDN

Value Types and Reference Types

Javascript has 5 data types that are passed by *value*: Boolean, null, undefined, String, **and** Number. We'll call these primitive types.

When we assign these variables to other variables using =, we **copy** the value to the new variable. They are copied by value

Javascript has 3 data types that are passed by *reference*: **Array, Function, and Object**. These are all technically Objects, so we'll refer to them collectively as Objects.

Variables that are assigned a non-primitive value are given a *reference* to that value. That reference points to the object's location in memory. The variables don't actually contain the value.

When a reference type value, an object, is copied to another variable using =, the address of that value is what's actually copied over as if it were a primitive. **Objects are copied by reference** instead of by value.

When the equality operators, == and ===, are used on reference-type variables, they check the reference. If the variables contain a reference to the same item, the comparison will result in `true`.

```
var arr1 = ['Hi!'];  
var arr2 = ['Hi!'];  
  
console.log(arr1 === arr2); // -> false
```

If we have two distinct objects and want to see if their properties are the same, the easiest way to do so is to turn them both into strings and then compare the strings

```
var arr1str = JSON.stringify(arr1);  
var arr2str = JSON.stringify(arr2);  
  
console.log(arr1str === arr2str); // true
```

When we pass primitive values into a function, the function copies the values into its parameters. It's effectively the same as using =

Remember that assignment through function parameters is essentially the

same as assignment with =

Pure Functions

Functions in JavaScript are objects.

We refer to functions that don't affect anything in the outside scope as *pure functions*. As long as a function only takes primitive values as parameters and doesn't use any variables in its surrounding scope, it is automatically pure, as it can't affect anything in the outside scope. All variables created inside are garbage-collected as soon as the function returns.

A function that takes in an Object, however, can mutate the state of its surrounding scope. If a function takes in an array reference and alters the array that it points to, perhaps by pushing to it, variables in the surrounding scope that reference that array see that change. After the function returns, the changes it makes persist in the outer scope. This can cause undesired side effects that can be difficult to track down.

Many native array functions, including `Array.map` and `Array.filter`, are therefore written as pure functions. **They take in an array reference and internally, they copy the array and work with the copy instead of the original.**

Implicit, Explicit, Nominal, Structuring and Duck Typing

Javascript's implicit coercion simply refers to Javascript attempting to coerce an unexpected value type to the expected type. So you can pass a string where it expects a number, an object where it expects a string etc, and it will try to convert it to the right type. This is a Javascript feature that is best avoided.

Double Equals vs. Triple Equals

When using triple equals `===` in JavaScript, we are testing for **strict equality**. This means both the **type** and the **value** we are comparing have to be the same.

When using double equals in JavaScript we are testing for **loose equality**. Double equals also performs **type coercion**. Type coercion means that two values are compared only **after** attempting to convert them into a common type.

```
77 === '77'  
// false (Number v. String)
```

```
77 == '77'  
// true
```

There are only six falsy values in JavaScript you should be aware of:

- **false** — boolean false
- **0** — number zero

- "" — empty string
- **null**
- **undefined**
- **NaN** — Not A Number

When comparing `null` and `undefined`, they are only equal to themselves and each other:

```
null == null // true
undefined == undefined // true
null == undefined // true
```

Lastly, `NaN` is not equivalent to anything. Even cooler, it's not even itself!

```
NaN == null
// false

NaN == undefined
// false

NaN == NaN
// false
```

loose equality can lead to more headaches than it's worth. Memorizing the six falsy values and the rules associated with them can go a long way towards understanding loose equality.

Triple Equals is superior to double equals. Whenever possible, you should use triple equals to test equality. By testing the type and value you can be sure that you are always executing a true equality test.

`undefined` is one of JavaScript's primitive types which means checking its type using the `typeof` operator returns the string `'undefined'`

It is the default value of any declared, but unassigned variable:

```
var x
console.log(x) // undefined
```

It is the value returned when trying to access an undeclared object property:

```
var obj = {}
console.log(obj.a) // undefined
```

It is the default return value of a function which does not return:

```
function f() {}
console.log(f()) // undefined
```

`null` is also a JavaScript primitive type, but checking its type using the `typeof` operator does not

return what you'd expect:

```
console.log(typeof null) // object
```

null does not show up as a default value anywhere. Instead it is usually returned by functions which are expected to return an object when one could not be retrieved from the given parameters.

For example, in browsers `document.getElementById` returns `null` if no element with the given ID was found in the HTML document:

```
console.log(document.getElementById('some-id-which-no-element-has')) // null
```

null represents a lack of identification.

undefined and null represent the absence of a value. Therefore, any code we write which aims to check for the absence of a value should account for both undefined and null.

```
console.log(null === undefined) // false
var obj = {}
console.log(obj === {}) // false (because objects are compared by reference)
console.log(obj === obj) // true (because reference to same object)
```

```
console.log({} == {}) // false (because objects are compared by reference)
```

you this to check is everytinh ok with variabile

```
console.log(typeof value === 'undefined' || value === null)
```

from: <https://tomeraberba.ch/html/post/checking-for-the-absence-of-a-value-in-javascript.html>

You have to differentiate between cases:

1. Variables can be `undefined` or *undeclared*. You'll get an error if you access an undeclared variable in any context other than `typeof`.

```
if(typeof someUndeclaredVar == whatever) // works
if(someUndeclaredVar) // throws error
```

A variable that has been declared but not initialized is `undefined`.

```
let foo;
if (foo) //evaluates to false because foo === undefined
```

2. Undefined *properties*, like `someExistingObj.someUndefProperty`. An undefined property doesn't yield an error and simply returns `undefined`, which, when converted to a boolean, evaluates to `false`. So, if you don't care about `0` and `false`, using `if(obj.undefProp)` is ok. There's a common idiom based on this fact:

```
value = obj.prop || defaultValue
```

which means "if `obj` has the property `prop`, assign it to `value`, otherwise assign the default value `defaultValue`".

Some people consider this behavior confusing, arguing that it leads to hard-to-find errors and recommend using the `in` operator instead

```
value = ('prop' in obj) ? obj.prop : defaultValue
```

also check <https://webbjocke.com/javascript-check-data-types/>

Articles on topic

- ? JavaScript Double Equals vs. Triple Equals — Brandon Morelli
- ? What is the difference between `=`, `==`, and `===` in JS? — Codecademy
- ? Should I use `===` or `==` equality comparison operator in JavaScript? — Panu Pitkamaki
- ? `==` vs `===` JavaScript: Double Equals and Coercion — AJ Meyghani
- ? Why Use the Triple-Equals Operator in JavaScript? — Louis Lazaris
- ? What is the difference between `==` and `===` in JavaScript? — Craig Buckler
- ? Why javascript's `typeof` always return "object"? — Stack Overflow
- ? Checking Types in Javascript — Toby Ho
- ? How to better check data types in JavaScript — Webbjocke
- ? Checking for the Absence of a Value in JavaScript — Tomer Aberbach

Expression vs Statement

Expressions are Javascript code snippets that result in a single value. Expressions can be as long as you want them to be, but they would always result in a single value.

```
2 + 2 * 3 / 2
```

```
(Math.random() * (100 - 20)) + 20
```

```
functionCall()
```

```
window.history ? useHistory() : noHistoryFallback()
```

Statements are the headache of functional programming ðŸ˜€. Basically, statements perform actions, they do things.

In javascript, statements can never be used where a value is expected. So they cannot be used as function arguments, right-hand side of assignments, operators operand, return values...

These are all javascript statements:

1. if
2. if-else
3. while
4. do-while
5. for
6. switch
7. for-in
8. with (deprecated)
9. debugger
10. variable declaration

A function declaration is a statement

whenever you declare a function where Javascript is expecting a value, it will attempt to treat it as a value, if it can't use it as a value, an error will be thrown.

Whereas declaring a function at the global level of a script, module, or top level of a block statement (that is, where it is not expecting a value), will result in a function declaration.

```
(function () {}) // this returns function () {}
```

```
r: 2+2 // valid
```

```
foo()
```

```
const foo = () => {}
```

setTimeout, setInterval and requestAnimationFrame

setTimeout allows us to run a function once after the interval of time.

```
1 function sayHi(phrase, who) {  
2   alert( phrase + ', ' + who );  
3 }  
4  
5 setTimeout(sayHi, 1000, "Hello", "John"); //
```

`setTimeout` expects a reference to a function.

setInterval allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval

All arguments have the same meaning. But unlike `setTimeout` it runs the function not only once, but regularly after the given interval of time.

In case of asynchronous operations, `setInterval` will create long queue of requests which will be very counterproductive.

In case of time intensive synchronous operations, `setInterval` may break the rhythm.

Also, if any error occurs in `setInterval` code block, it will not stop execution but keeps on running faulty code. Not to mention they need a `clearInterval` function to stop it.

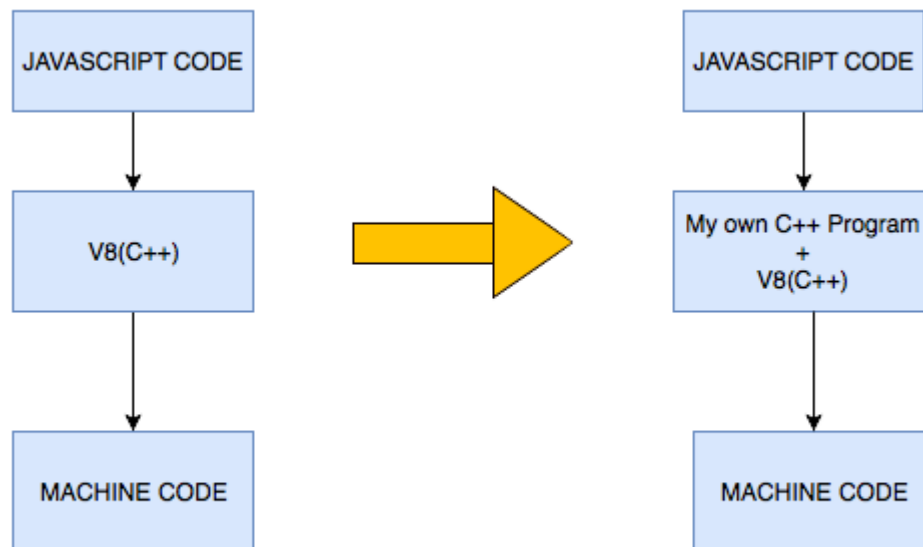
Alternatively, you can use `setTimeout` recursively in case of time sensitive operations

-> do not use **setInterval**

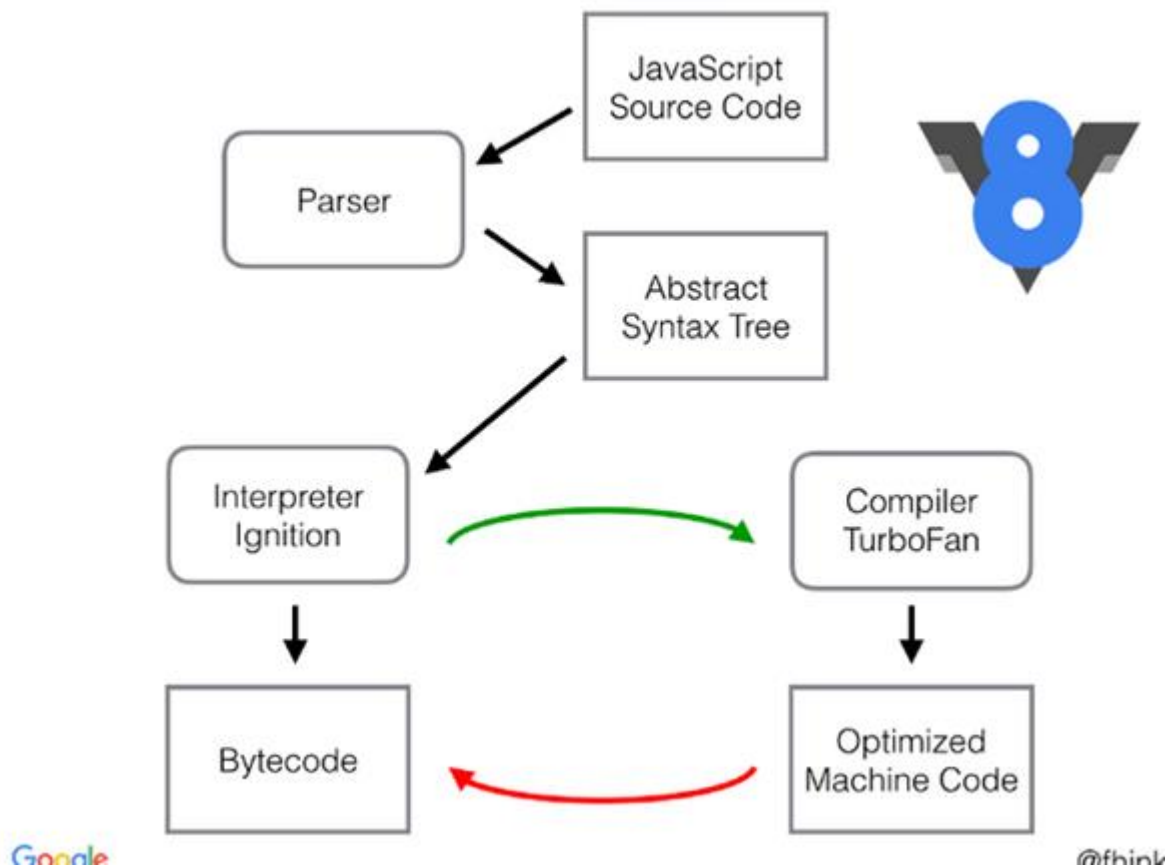
JavaScript Engines

JavaScript Engine? It is a program that converts Javascript code into lower level or machine code that microprocessors can understand

There are different JavaScript engines including Rhino, JavaScriptCore, and SpiderMonkey. These engines follow the ECMAScript Standards. ECMAScript defines the standard for the scripting language. JavaScript is based on ECMAScript standards. These standards define how the language should work and what features it should have. You can learn more about ECMAScript [here](#).



When V8 compiles JavaScript code, the parser generates an abstract syntax tree. A syntax tree is a tree representation of the syntactic structure of the JavaScript code. Ignition, the interpreter, generates bytecode from this syntax tree. TurboFan, the optimizing compiler, eventually takes the bytecode and generates optimized machine code from it.



Bytecode is an abstraction of machine code. Compiling bytecode to machine code is easier if the bytecode was designed with the same computational model as the physical CPU. This is why interpreters are often register or stack machines. **Ignition is a register machine with an accumulator register.**

More here : <https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775>

DOM and Layout Trees

The Document Object Model, usually referred to as the DOM, is an essential part of making websites interactive. It is an interface that allows a programming language to manipulate the content, structure, and style of a website. JavaScript is the client-side scripting language that connects to the DOM in an internet browser.

At the most basic level, a website consists of an HTML document. The browser that you use to view the website is a program that interprets HTML and CSS and renders the style, content, and structure into the page that you see.

In addition to parsing the style and structure of the HTML and CSS, the **browser creates a representation of the document known as the Document Object Model.** This **model** allows JavaScript to access the text content and elements of the website **document** as **objects**.

In fact a lot of what you might think of as a “JavaScript Thing” is more accurately a “DOM API”. For instance, we can write JavaScript that watches for a `mouseenter` event on an element. But that “element” is really a DOM node. We attach that listener via a DOM property on that DOM node. When that event happens, it’s the DOM node that emits that event.

Documentation: <https://www.w3.org/TR/DOM-Level-2-Core/introduction.html>

When you create a script—whether it's inline in a `<script>` element or included in the web page by means of a script loading instruction—you can immediately begin using the API for the `document` or `window` elements to manipulate the document itself or to get at the children of that document, which are the various elements in the web page. Your DOM programming may be something as simple as the following, which displays an alert message by using the `alert()` function from the `window` object, or it may use more sophisticated DOM methods to actually create new content, as in the longer example below.

Web browsers rely on layout engines to parse HTML into a DOM. Some layout engines, such as Trident/MSHTML, are associated primarily or exclusively with a particular browser, such as Internet Explorer. Others, including Blink, WebKit, and Gecko, are shared by a number of browsers, such as Google Chrome, Opera, Safari, and Firefox. The different layout engines implement the DOM standards to varying degrees of compliance.

DOM implementations:

- libxml2
- MSXML
- Xerces is a collection of DOM implementations written in C++, Java and Perl
- XML for <SCRIPT> is a JavaScript-based DOM implementation[10]
- PHP.Gt DOM is a server-side DOM implementation based on libxml2 and brings DOM level 4 compatibility[11] to the PHP programming language
- Domino is a Server-side (Node.js) DOM implementation based on Mozilla's dom.js. Domino is used in the MediaWiki stack with Visual Editor.
- SimpleHtmlDom is a simple HTML document object model in C#, which can generate HTML string programmatically.

APIs that expose DOM implementations:

- JAXP (Java API for XML Processing) is an API for accessing DOM providers
- Lazarus (Free Pascal IDE) contains two variants of the DOM - with UTF-8 and ANSI format

Inspection tools:

- DOM Inspector is a web developer tool

The CSSOM and DOM trees are combined into a render tree, which is then used to compute the layout of each visible element and serves as an input to the paint process that renders the pixels to screen. Optimizing each of these steps is critical to achieving optimal rendering performance.

In the previous section on constructing the object model, we built the DOM and the CSSOM trees based on the HTML and CSS input. However, both of these are independent objects that capture different aspects of the document: one describes the content, and the other describes the style rules that need to be applied to the document. How do we merge the two and get the browser to render pixels on the screen?

- The DOM and CSSOM trees are combined to form the render tree.
- Render tree contains only the nodes required to render the page.
- Layout computes the exact position and size of each object.
- The last step is paint, which takes in the final render tree and renders the pixels to the screen.

Factories and Classes

JavaScript is a prototype-based language, and every object in JavaScript has a hidden internal property called `[[Prototype]]` that can be used to extend object properties and methods.

To find the `[[Prototype]]` of this newly created object, we will use the `getPrototypeOf()` method.

```
Let x = {}
```

```
Object.getPrototypeOf(x);
```

```
output: {constructor: f, __defineGetter__: f, __defineSetter__: f, ...}
```

```
let y = [];
```

```
Object.getPrototypeOf(y);
```

```
[constructor: f, concat: f, pop: f, push: f, ...]
```

These prototypes can be chained, and each additional object will inherit everything throughout the chain. The chain ends with the `Object.prototype`

Constructor functions are functions that are used to construct new objects. The `new` operator is used to create new instances based off a constructor function. We have seen some built-in JavaScript constructors, such as `new Array()` and `new Date()`, but we can also create our own custom templates from which to build new objects.

To begin, a constructor **function is just a regular function. It becomes a constructor when it is called on by an instance with the `new` keyword**. In JavaScript, we capitalize the first letter of a constructor function by convention

```
// Initialize a constructor function for a new Hero
```

```
function Hero(name, level) {
```

```
  this.name = name;
```

```
  this.level = level;
```

```
}
```

```
let hero1 = new Hero('Bjorn', 1);
```

```
Object.getPrototypeOf(herol);  
output: constructor: f Hero(name, level)
```

We can add a method to Hero using prototype. We'll create a greet () method.

```
// Add greet method to the Hero prototype  
Hero.prototype.greet = function () {  
  return `${this.name} says hello.`;  
}  
herol.greet();
```

We can use the call () method to copy over properties from one constructor into another constructor. Let's create a Warrior and a Healer constructor.

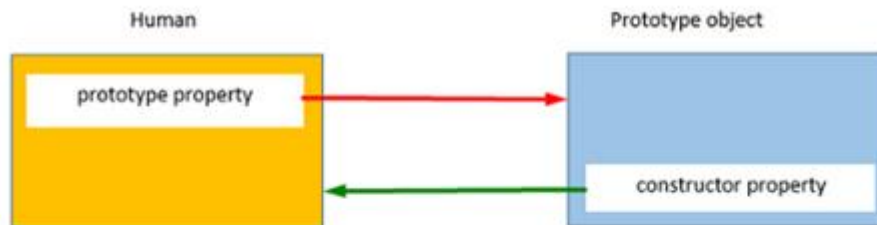
```
...  
// Initialize Warrior constructor  
function Warrior(name, level, weapon) {  
  // Chain constructor with call  
  Hero.call(this, name, level);  
  // Add a new property  
  this.weapon = weapon;  
}
```

Prototype properties and methods are not automatically linked when you use call () to chain constructors. We will use Object.create () to link the prototypes, making sure to put it before any additional methods are created and added to the prototype.

```
Warrior.prototype = Object.create(Hero.prototype);  
Healer.prototype = Object.create(Hero.prototype);  
// All other prototype methods added below
```


Since **Javascript is a functional programming language** where everything is just a function, in order to have a class like (creating a blueprint for the objects to be created) functionality in javascript, constructor functions are used

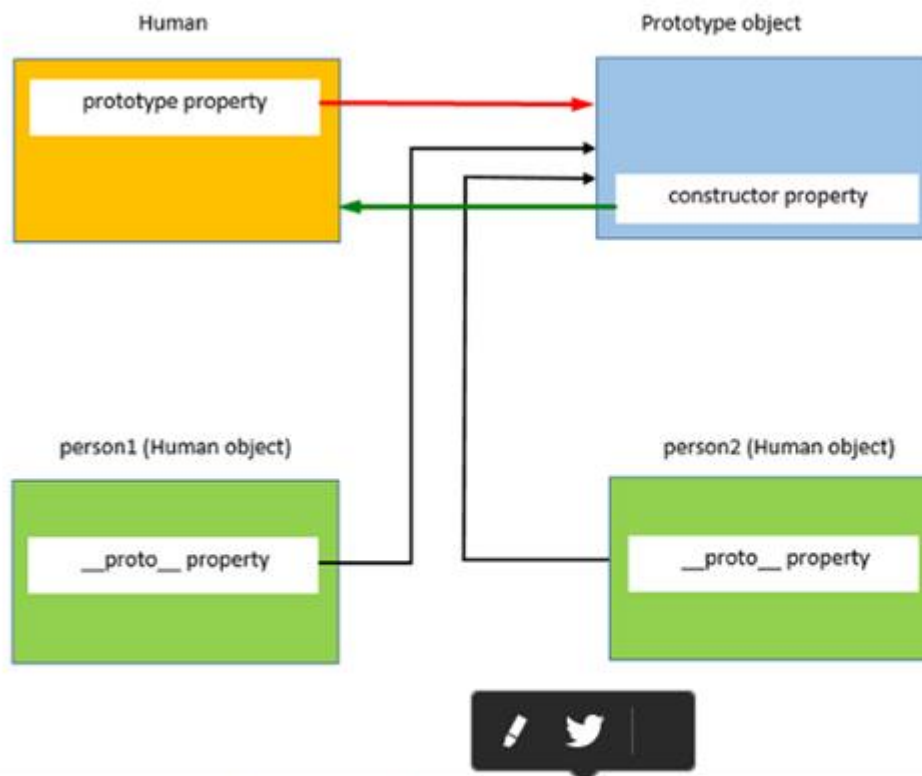
Whenever a new function is created in javascript, Javascript engine by default adds a prototype property to it, this property is an object and we call it "prototype object". By default this prototype object has a constructor property which points back to our function, and another property `__proto__` which is an object



As shown in the above image, `Human` constructor function has a `prototype` property that points to the `Prototype object`. The `prototype` object has a `constructor` property that points back to the `Human` constructor function. Let's see an example below:

```
> function Vehicle(make, model, color) {
  this.make = make,
  this.model = model,
  this.color = color,
  this.getName = function () {
    return this.make + " " + this.model;
  }
}
< undefined
> Vehicle.prototype
< ▼ {constructor: f} ⓘ
  ► constructor: f Vehicle(make, model, color)
  ► __proto__: Object
```





The prototype object of the constructor function is shared among all the objects created using the constructor function.

Now, this prototype object can be used to add new properties and methods to the constructor function using the following syntax

```
car.prototype.year = "2016";
```

Prototype properties and methods are shared between all the instances of the constructor function, but when any one of the instances of a constructor function makes any change in any primitive property, it will only be reflected in that instance and not among all the instances:

The prototype object of the constructor function is shared among all the objects created using the constructor function.

Javascript classes are nothing but just a new way of writing the constructor functions by utilizing the power of prototype

Static methods class

Static methods are functions on class itself, and not on its prototype, unlike the other methods in the class, which are defined on its prototype.

Static methods are declared using `static` keyword, and are mostly used to create utility functions.

They are called without creating the instance of the class. See an example below.

In traditional class-oriented languages, you create *classes*, which are templates for *objects*. When you want a new object, you *instantiate* the class, which tells the language engine to *copy* the methods and properties of the class into a new entity, called an *instance*. The *instance* is your object, and, after instantiation, has absolutely no active relation with the parent class.

But JavaScript

Rather, it creates an object that is linked to a *prototype*. Changes to that prototype propagate to the new object, *even after* instantiation

```
// joe has no toString property . . .
const joe    = { name : 'Joe' },
      sara   = { name : 'Sara' };

Object.hasOwnProperty(joe, toString); // false
Object.hasOwnProperty(sara, toString); // false

// . . . But we can call it anyway!
joe.toString(); // '[object Object]', instead of ReferenceError!
sara.toString(); // '[object Object]', instead of ReferenceError!
```

Explanation:

The output from our `toString` calls is utterly useless, but note that this snippet doesn't raise a single `ReferenceError`! That's because, while neither `joe` or `sara` has a `toString` property, *their prototype does*.

When we look for `sara.toString()`, `sara` says, "I don't have a `toString` property. Ask my prototype." JavaScript, obligingly, does as told, and asks `Object.prototype` if *it* has a `toString` property. Since it does, it hands `Object.prototype`'s `toString` back to our program, which executes it.

Here's a quick recap before we see precisely where these prototypes come from:

- `joe` and `sara` do *not* "inherit" a `toString` property;
- `joe` and `sara`, as a matter of fact, do *not* "inherit" from `Object.prototype` *at all*;
- `joe` and `sara` *are* linked to `Object.prototype`;
- Both `joe` and `sara` are linked to the *same* `Object.prototype`.
- To find the prototype of an object -- let's call it `O` -- you use: `Object.getPrototypeOf(O)`.

Here's what you need to know about `Object` and `Object.prototype` :

1. The function, `Object` , has a property, called `.prototype` , which points to an object (`Object.prototype`);
2. The object, `Object.prototype` , has a property, called `.constructor` , which points to a function (`Object`).

As it turns out, this general scheme is true for *all* functions in JavaScript. When you create a function -- `someFunction` -- it will have a property, `.prototype` , that points to an object, called `someFunction.prototype` .

More info here: <https://scotch.io/tutorials/better-javascript-with-es6-pt-ii-a-deep-dive-into-classes>

<https://scotch.io/tutorials/better-javascript-with-es6-pt-ii-a-deep-dive-into-classes>

Factory Design Pattern.

In a nutshell, if we implement instance creation inside the function itself and don't require user to use `Object.create` or `new` keyword, then we have a factory.

More here: <https://medium.com/@SntsDev/the-factory-pattern-in-js-es6-78f0afad17e9>

this, call, apply and bind

You can use `call()`/`apply()` to invoke the function immediately. `bind()` returns a bound function that, when executed later, will have the correct context ("**this**") for calling the original function. So `bind()` can be used when the function needs to be called later in certain events when it's useful.

```
//Demo with javascript .call()

var obj = {name:"Niladri"};

var greeting = function(a,b,c){
    return "welcome "+this.name+" to "+a+" "+b+" in "+c;
};

console.log(greeting.call(obj,"Newtown","KOLKATA","WB"));
// returns output as welcome Niladri to Newtown KOLKATA in WB
```

Similarly to `call()` method the first parameter in `apply()` method sets the "**this**" value which is the object upon which the function is invoked. In this case it's the "**obj**" object above. The only difference of `apply()` with the `call()` method is that the second parameter of the `apply()`

method accepts the arguments to the actual function as an array.

```
var obj = {name:"Niladri"};

var greeting = function(a,b,c){
    return "welcome "+this.name+" to "+a+" "+b+" in "+c;
};

//creates a bound function that has same body and parameters
var bound = greeting.bind(obj);

console.dir(bound); ///returns a function

console.log("Output using .bind() below ");

console.log(bound("Newtown","KOLKATA","WB")); //call the bound function
```

for **bind()** we are returning a **bound function with the context which will be invoked later**.

One thing that is confusing about Javascript is that **this**'s value will change depending on the context that this is called in. By default it refers to the global scope, but when called within a function, it refers to that function

Different objects can share methods without rewriting them for each individual object.

Apply, call, and bind are sometimes called **function methods**, since they are called alongside a function.

More here: <https://dev.to/kbk0125/javascripts-apply-call-and-bind-explained-by-hosting-a-cookout-32jo>

Using **call** to invoke an anonymous function

```
const animals = [
    { species: 'Lion', name: 'King' },
    { species: 'Whale', name: 'Fail' }
];

for (let i = 0; i < animals.length; i++) {
    (function(i) {
        this.print = function() {
            console.log('#' + i + ' ' + this.species
                + ': ' + this.name);
        }
        this.print();
    }).call(animals[i], i);
}
```

from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind

<https://developer.mozilla.org/en->

new, Constructor, instanceof and Instances

Introduction:

<https://codeburst.io/javascript-for-beginners-the-new-operator-cee35beb669e>

```
Object.create(proto, [ propertiesObject ])
```

`Object.create` methods accept two arguments:

1. `proto`: the object which should be the prototype of the newly-created object. It has to be an object or null.
2. `propertiesObject`: properties of the new object. This argument is optional.

Basically, you pass into `Object.create` an object that you want to inherit from, and it returns a new object that inherits from the object you passed into it.

1 Background: `typeof` vs. `instanceof`

In JavaScript, you have to choose when it comes to checking the type of a value. The rough rule of thumb is:

- `typeof` checks if a value is an element of a primitive type:

```
if (typeof value === 'string') ...
```

- `instanceof` checks if a value is an instance of a class or a constructor function:

```
if (value instanceof Map) ...
```


Object api

hasOwnProperty is a method available on object instances that allows to check if an object has a property directly on its instance.

The **Object.keys** static method returns an array with the property keys on an object:

Object.assign. This new method allows to easily copy values from one object to another

Object.freeze to shallowly freeze an object to prevent its properties from being changed

defineProperty

Defines a new property with name equal to `propertyName` on `obj` with the supplied descriptor.

entries Returns an Array of key/value pairs for the properties of `obj`.

See: <https://www.javascripture.com/Object> also

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Meta_programming

and [reflect](https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Reflect)

(https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Reflect)

and [proxy](https://www.javascripture.com)

(<https://www.javascripture.com>)

map, reduce, filter

Map, **Filter** and **Reduce** are chainable and, together, you hold an unlimited amount of mystical power! ? ✨ ✨

Given these more complex arrays of songs where we have an array of arrays with songs from Spotify and LastFM. Each song has a duration in seconds.

```
const spotifySongs = [
  { id: 1, name: "Curl of the Burl", artist: "Mastodon", duration: 204 },
  { id: 2, name: "Oblivion", artist: "Mastodon", duration: 306 },
  { id: 3, name: "Flying Whales", artist: "Gojira", duration: 444 },
  { id: 4, name: "L'Enfant Sauvage", artist: "Gojira", duration: 246 }
];

const lastFmSongs = [
  { id: 5, name: "Chop Suey", artist: "System of a Down", duration: 192 },
  { id: 6, name: "Throne", artist: "Bring me the Horizon", duration: 186 },
  { id: 7, name: "Destrier", artist: "Agent Fresco", duration: 132 },
  { id: 8, name: "Out of the Black", artist: "Royal Blood", duration: 240 }
];

const allSongs = [...spotifySongs, ...lastFmSongs];
```

```
// Let's reduce the array of arrays into a single one
const songNames = allSongs
  .reduce((acc, currValue) => {
    return acc.concat(currValue);
  }, [])
// Let's map it out with the seconds turned into minutes
.map(song => {
  return { ...song, duration: Math.floor(song.duration / 60) };
})
// Let's filter the ones under 3 minutes
.filter(song => {
  return song.duration > 3;
})
// Now let's map out the song names the quick way
.map(song => song.name)
// Join'em up
.join(" , ");

console.log(songNames); // Oblivion , Flying Whales , L'Enfant Sauvage , Out of the Black
```

Now we must get a string separated by commas with all the songs that have a duration superior to 3 minutes.

Closures

The use of closures is associated with languages where functions are first-class objects, in which functions can be returned as results from higher-order functions, or passed as arguments to other function calls; if functions with free variables are first-class, then returning one creates a closure. This includes functional programming languages such as Lisp and ML, as well as many modern, multi-paradigm languages, such as Python and Rust. Closures are also frequently used with callbacks, particularly for event handlers, such as in JavaScript, where they are used for interactions with a dynamic web page.

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2)); // 7
console.log(add10(2)); // 12
```

In this example, we have defined a function `makeAdder(x)`, that takes a single argument `x`, and returns a new function. The function it returns takes a single argument `y`, and returns the sum of `x` and `y`.

In essence, `makeAdder` is a function factory. It creates functions that can add a specific value to their argument. In the above example, the function factory creates two new functions—one that adds five to its argument, and one that adds 10.

`add5` and `add10` are both closures. They share the same function body definition, but store different lexical environments. In `add5`'s lexical environment, `x` is 5, while in the lexical environment for `add10`, `x` is 10.

Closures are useful because they let you associate data (the lexical environment) with a function that operates on that data. This has obvious parallels to object-oriented programming, where objects allow you to associate data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Every closure has three scopes:

- Local Scope (Own scope)
- Outer Functions Scope
- Global Scope

It is unwise to unnecessarily create functions within other functions if closures are not needed for a particular task, as it will negatively affect script performance both in terms of processing speed and memory consumption.

For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. The reason is that whenever the constructor is called, the methods would get reassigned (that is, for every object creation).

```
1: function createCounter() {
2:   let counter = 0
3:   const myFunction = function() {
4:     counter = counter + 1
5:     return counter
6:   }
7:   return myFunction
8: }
9: const increment = createCounter()
10: const c1 = increment()
11: const c2 = increment()
12: const c3 = increment()
13: console.log('example increment', c1, c2, c3)
```

outout: 1, 2 3, Look up at *execution context* ; global, funciton , state

`createCounter()` retuns function but save sate of the counter in the `increment` varivable

The closure is a collection of all the variables in scope at the time of creation of the function.

When a function returns a function, that is when the concept of closures becomes

more relevant. The returned function has access to variables that are not in the global scope, but they solely exist in its closure.

High Order Functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions. Higher-order functions allow us to abstract over actions, not just values.

```
function unless(test, then) {  
  if (!test) then();  
}  
  
repeat(3, n => {  
  unless(n % 2 == 1, () => {  
    console.log(n, "is even");  
  });  
});
```

One area where higher-order functions shine is data processing. To process data, we'll need some actual data. This chapter will use a data set about scripts

Recursion

Loosely defined, recursion is the process of taking a big problem and sub-dividing it into multiple, smaller instances of the same problem.

Put into practice, that generally means writing a function that calls *itself*.

All recursive functions should have three key features:

- A Termination Condition
Fail-safe. Think of it like your emergency brake. It's put there in case of bad input to prevent the recursion from ever running. In our factorial example, `if (x < 0) return;`
- A Base Case
Stops recursion, *s the goal* of our recursive function
- Recursion
Function call itself.

Example:

```
function factorial(x) {  
  // TERMINATION  
  if (x < 0) return; // BASE  
  if (x === 0) return 1; // RECURSION  
  return x * factorial(x - 1);  
}
```

It will create stack of execution and then go back and call every method in stack from first (last added) to last (first added)

Collections and Generators

A plain `Object`, after all, is pretty much nothing but an open-ended collection of key-value pairs. You can get, set, and delete properties, iterate over them—all the things a hash table can do.

Set

A `Set` is a collection of values. It's mutable, so your program can add and remove values as it goes.

a set never contains the same value twice (**unique**)

can contain any type of JS value

does not have indexing

Here are all the operations on sets:

- `new Set` creates a new, empty set.
- `new Set(iterable)` makes a new set and fills it with data from [any iterable value](#).
- `set.size` gets the number of values in the set.
- `set.has(value)` returns `true` if the set contains the given value.
- `set.add(value)` adds a value to the set. If the value was already in the set, nothing happens.
- `set.delete(value)` removes a value from the set. If the value wasn't in the set, nothing happens. Both `.add()` and `.delete()` return the set object itself, so you can chain them.
- `set[Symbol.iterator]()` returns a new iterator over the values in the set. You won't normally call this directly, but this method is what makes sets iterable. It means you can write `for (v of set) {...}` and so on.
- `set.forEach(f)` is easiest to explain with code. It's like shorthand for:

```
for (let value of set)
  f(value, value, set);
```

This method is analogous to the `.forEach()` method on arrays.

- `set.clear()` removes all values from the set.
- `set.keys()`, `set.values()`, and `set.entries()` return various iterators. These are provided for compatibility with `Map`, so we'll talk about them below.

advanced

- Functional helpers that are already present on arrays, like `.map()`, `.filter()`, `.some()`, and `.every()`.
- Non-mutating `set1.union(set2)` and `set1.intersection(set2)`.
- Methods that can operate on many values at once:
`set.addAll(iterable)`, `set.removeAll(iterable)`, and
`set.hasAll(iterable)`.

Map

A Map is a collection of key-value pairs.

- `new Map` returns a new, empty map.
- `new Map(pairs)` creates a new map and fills it with data from an existing collection of `[key, value]` pairs. *pairs* can be an existing `Map` object, an array of two-element arrays, a generator that yields two-element arrays, etc.
- `map.size` gets the number of entries in the map.
- `map.has(key)` tests whether a key is present (like `key in obj`).
- `map.get(key)` gets the value associated with a key, or undefined if there is no such entry (like `obj[key]`).
- `map.set(key, value)` adds an entry to the map associating *key* with *value*, overwriting any existing entry with the same key (like `obj[key] = value`).
- `map.delete(key)` deletes an entry (like `delete obj[key]`).
- `map.clear()` removes all entries from the map.
- `map[Symbol.iterator]()` returns an iterator over the entries in the map. The iterator represents each entry as a new `[key, value]` array.
- `map.forEach(f)` works like this:

```
for (let [key, value] of map)
  f(value, key, map);
```

The odd argument order is, again, by analogy to `Array.prototype.forEach()`.

- `map.keys()` returns an iterator over all the keys in the map.
- `map.values()` returns an iterator over all the values in the map.
- `map.entries()` returns an iterator over all the entries in the map, just like `map[Symbol.iterator]()`. In fact, it's just another name for the same method.

`Map` and `Set` objects keep a strong reference to every key and value they contain. This means that

if a DOM element is removed from the document and dropped, garbage collection can't recover that memory until that element is removed from containing map or set

WeakMap and WeakSet

WeakMap and WeakSet are specified to behave exactly like Map and Set, but with a few restrictions:

- `WeakMap` supports only `new`, `.has()`, `.get()`, `.set()`, and `.delete()`.
- `WeakSet` supports only `new`, `.has()`, `.add()`, and `.delete()`.
- The values stored in a `WeakSet` and the keys stored in a `WeakMap` must be objects.

Note that neither type of weak collection is iterable. You can't get entries out of a weak collection except by asking for them specifically, passing in the key you're interested in.

These carefully crafted restrictions enable the garbage collector to collect dead objects out of live weak collections. The effect is similar to what you could get with weak references or weak-keyed dictionaries, but ES6 weak collections get the memory management benefits *without exposing the fact that GC happened to scripts*.

The main difference is that `WeakSet` can only contain objects & not any other type.

The other main difference between the two is that references to objects in a `WeakSet` are held "weakly". This means that if there is no other reference to an object in the `WeakSet` it will get garbage collected. The same is not true for `Set`. An object stored in a `Set` will not be garbage collected even if nothing is referencing it.

Generators are functions that you can use to control the iterator. They can be suspended and later resumed at any time.

```
1  function * generatorForLoop(num) {
2    for (let i = 0; i < num; i += 1) {
3      yield console.log(i);
4    }
5  }
6
7  const genForLoop = generatorForLoop(5);
8
9  genForLoop.next(); // first console.log - 0
10 genForLoop.next(); // 1
11 genForLoop.next(); // 2
12 genForLoop.next(); // 3
13 genForLoop.next(); // 4
```

genreator-for-loop.js hosted with ❤ by GitHub

[view raw](#)

What does it do? In fact, it just wraps our for-loop from the example above with some changes. But the most significant change is that it does not ring immediately. And this is the most important feature in generators — we can get the next value in only when we really need it, not all the values at once. And in some situations it can be very convenient.

How can we declare the generator function? There is a list of possible ways to do this, but the main thing is to add an asterisk after the function keyword.

Yield returns a value only once, and the next time you call the same function it will move on to the next *yield* statement.

Also in generators we always get the object as output. It always has two properties *value* and *done*. And as you can expect, *value* - returned value, and *done* shows us whether the generator has finished its job or not.

```

1  function * generator() {
2    yield 5;
3  }
4
5  const gen = generator();
6
7  gen.next(); // {value: 5, done: false}
8  gen.next(); // {value: undefined, done: true}
9  gen.next(); // {value: undefined, done: true} - all other calls will produce the same result

```

Previously, we used generators with a known number of iterations. But what if we don't know how many iterations are needed. To solve this problem, it is enough to create an infinite loop in the function generator. The example below demonstrates this for a function that returns a random number.

```

1  function * randomFrom(...arr) {
2    while (true)
3      yield arr[Math.floor(Math.random() * arr.length)];
4  }
5
6  const getRandom = randomFrom(1, 2, 5, 9, 4);
7
8  getRandom.next().value; // returns random value

```

There are many more possible ways to use generators. For example, they can be useful when working with asynchronous operations. Or iterate through an on-demand item loop

Better Async functionality

Code using promises and callbacks such as —

```
function fetchJson(url) {  
  return fetch(url)  
    .then(request => request.text())  
    .then(text => {  
      return JSON.parse(text);  
    })  
    .catch(error => {  
      console.log(`ERROR: ${error.stack}`);  
    });  
}
```

can be written as (with the help of libraries such as [co.js](#))—

```
const fetchJson = co.wrap(function * (url) {  
  try {  
    let request = yield fetch(url);  
    let text = yield request.text();  
    return JSON.parse(text);  
  }  
  catch (error) {  
    console.log(`ERROR: ${error.stack}`);  
  }  
});
```

Some readers may have noticed that it parallels the use of `async/await`.

Generators as observers

Generators can also receive values using the `next(val)` function. Then the generator is called an observer since it wakes up when it receives new values. In a sense, it keeps *observing* for values and acts when it gets one. You can read more about this pattern [here](#).

Lazy Evaluation

As seen with **Infinite Data Streams** example, it is possible only because of lazy evaluation. Lazy Evaluation is an evaluation model which delays the evaluation of an expression until its value is needed. That is, if we don't need the value, it won't exist. It is **calculated** as we demand it. Let's see an example —

```
function * powerSeries(number, power) {  
  let base = number;  
  while(true) {  
    yield Math.pow(base, power);  
    base++;  
  }  
}
```

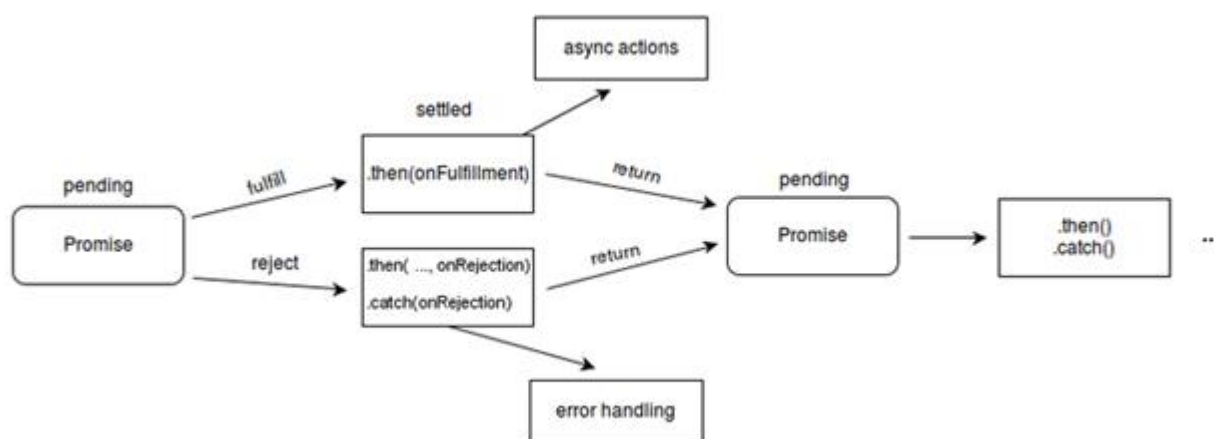
Generator objects are one-time access only. Once you've exhausted all the values, you can't iterate over it again. To generate the values again, you need to make a new generator object

Promises

Good example with http get: <https://www.sitepoint.com/overview-javascript-promises/>

Good basic example: <https://ponyfoo.com/articles/es6-promises-in-depth>

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future.



The methods `promise.then()`, `promise.catch()`, and `promise.finally()` are used to associate further action with a promise that becomes settled.

The `.then()` method takes up to two arguments; the first argument is a callback function for the resolved case of the promise, and the second argument is a callback function for the rejected case. Each `.then()` returns a newly generated promise object, which can optionally be used for chaining; for example:

Imagine you are a **kid**. Your mom **promises** you that she'll get you a **new phone** next week."

You *don't know* if you will get that phone until next week. Your mom can *really buy* you a brand new phone, or *she doesn't*.

That is a **promise**. A promise has three states. They are:

1. Pending: You *don't know* if you will get that phone
2. Fulfilled: Mom is *happy*, she buys you a brand new phone
3. Rejected: Mom is *unhappy*, she doesn't buy you a phone

// ES5: Part 1

var isMomHappy = false;

// Promise

```
var willIGetNewPhone = new Promise(
  function (resolve, reject) {
    if (isMomHappy) {
      var phone = {
        brand: 'Samsung',
        color: 'black'
      };
      resolve(phone); // fulfilled
    } else {
      var reason = new Error('mom is not happy');
      reject(reason); // reject
    }
  }
);
```

Now that we have the promise, let's consume it:

// ES5: Part 2

*var willIGetNewPhone = ... *// continue from part 1**

// call our promise

```
var askMom = function () {
  willIGetNewPhone
    .then(function (fulfilled) {
      // yay, you got a new phone
      console.log(fulfilled);
      // output: { brand: 'Samsung', color: 'black' }
    })
    .catch(function (error) {
      // oops, mom didn't buy it
      console.log(error.message);
      // output: 'mom is not happy'
    })
};
```

```
    });  
};  
  
askMom();
```

Read more here: <https://hackernoon.com/understanding-promises-in-javascript-13d99df067c1>

Thumb Rules for using promises

1. Use promises whenever you are using async or blocking code.
2. `resolve` maps to `then` and `reject` maps to `catch` for all practical purposes.
3. Make sure to write both `.catch` and `.then` methods for all the promises.
4. If something needs to be done in both the cases use `.finally`
5. We only get one shot at mutating each promise.
6. We can add multiple handlers to a single promise.
7. The return type of all the methods in `Promise` object whether they are static methods or prototype methods is again a `Promise`
8. In `Promise.all` the order of the promises are maintained in values variable irrespective of which promise was first resolved.

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

More advices: <https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-promise-27fc71e77261> and here <https://medium.com/datafire-io/es6-promises-patterns-and-anti-patterns-bbb21a5d0918> and here: https://exploringjs.com/es6/ch_promises.html

async/await

Async and Await are extensions of promises.

The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.

The keyword `await` makes JavaScript wait until that promise settles and returns its result.

If we try to use `await` in a non-async function, there would be a syntax error:

The `async` keyword before a function has two effects:

1. Makes it always return a promise.
2. Allows `await` to be used in it.

The `await` keyword before a promise makes JavaScript wait until that promise settles, and then:

1. If it's an error, the exception is generated — same as if `throw error` were called at that

very place.

2. Otherwise, it returns the result.

Together they provide a great framework to write asynchronous code that is easy to both read and write.

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status === 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    });
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404
```

new version with async and await

```
async function loadJson(url) { // (1)
  let response = await fetch(url); // (2)

  if (response.status === 200) {
    let json = await response.json(); // (3)
    return json;
  }

  throw new Error(response.status);
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404 (4)
```

Data Structures

At a high level, there are basically three types of data structures. **Stacks** and **Queues** are *array-like* structures that differ only in how items are inserted and removed. **Linked Lists**, **Trees**, and **Graphs** are structures with *nodes* that keep *references* to other nodes. **Hash Tables** depend on *hash functions* to save and locate data.

In terms of complexity, **Stacks** and **Queues** are the simplest and can be constructed from **Linked Lists**. **Trees** and **Graphs** are the most complex because they extend the concept of a linked list. **Hash Tables** need to utilize these data structures to perform reliably. In terms of efficiency, **Linked Lists** are most optimal for *recording* and *storing* of data, while **Hash Tables** are most performant for *searching* and *retrieving* of data.

Stack

-> array with two operations: push and pop

Push add element on top

Pop removes elements from top

It is Last In First Out

Queue

-> array with two primary operations: unshift and pop

Unshift puts element at the end of the array

Pop remove elements from the beginning of the array

First In First Out

Linked List

Like arrays, `LinkedLists` store data elements in *sequential* order. Instead of keeping indexes, linked lists hold *pointers* to other elements. The *first node* is called the **head** while the *last node* is called the **tail**. In a **singly-linked list**, each node has only one pointer to the *next* node. Here, the *head* is where we begin our walk down the rest of the list. In a **doubly-linked list**, a pointer to the *previous* node is also kept. Therefore, we can also start from the *tail* and walk “backwards” toward the head.

Linked lists have *constant-time insertions* and *deletions* because we can just change the pointers. To do the same operations in arrays requires *linear time* because subsequent items need to be shifted over. Also, linked lists can grow as long as there is space. However, even “dynamic” arrays that automatically resize could become unexpectedly expensive. Of course, there’s always a tradeoff. To lookup or edit an element in a linked list, we might have to walk the entire length which equates to linear time. With array indexes, however, such operations are trivial.

Like arrays, linked lists can operate as *stacks* or *queue*

Tree

A `Tree` is like a *linked list*, except it keeps references to *many child nodes* in a *hierarchical* structure. In other words, each node can have no more than one parent. The **Document Object Model** (DOM) is such a structure

The **Binary Search Tree** is special because each node can have no more than *two children*.

Traversal through the tree can happen in a *vertical* or *horizontal* procedure. In **Depth-First Traversal** (DFT) in the vertical direction, a recursive algorithm is more elegant than an iterative one. Nodes can be traversed in *pre-order*, *in-order*, or *post-order*.

In **Breadth-First Traversal** (BFT) in the horizontal direction, an iterative approach is more elegant than a recursive one. This requires the use of a `queue` to track all the children nodes with each iteration. The memory needed for such a queue might not be trivial, however. If the shape of a tree is wider than deep, BFT is a better choice than DFT.

Graph

If a tree is free to have more than one parent, it becomes a `Graph`. Edges that connect nodes together in a graph can be *directed* or *undirected*, *weighted* or *unweighted*. Edges that have both direction and weight are analogous to *vectors*

A social network and the Internet itself are also graphs. The most complicated graph in nature is our

human brain, which **neural networks** attempt to replicate to give machines *superintelligence*

Hash Table

Hash Table is a dictionary-like structure that pairs *keys* to *values*. The location in memory of each pair is determined by a `hash function`, which accepts a *key* and returns the *address* where the pair should be inserted and retrieved. Collisions can result if two or more keys convert to the same address. For robustness, `getters` and `setters` should anticipate these events to ensure that all data can be recovered and no data is overwritten. Usually, `linked lists` offer the simplest solution. Having very large tables also helps.

On both the client and server, many popular libraries utilize **memoization** to maximize performance. By keeping a record of the *inputs* and *outputs* in a hash table,

No one data structure is perfect for every situation because optimizing for one property always equates to losing another. Some structures are more efficient at storing data while some others are more performant for searching through them. Usually, one is sacrificed for the other. At one extreme, **linked lists** are optimal for storage and can be made into **stacks** and **queues** (*linear* time). At the other, no other structure can match the search speed of **hash tables** (*constant* time). **Tree** structures lie somewhere in the middle (*logarithmic* time), and only a **graph** can portray nature's most complex structure: the human brain (*polynomial* time). Having the skillset to distinguish *when* and articulate *why* is a hallmark of a rockstar engineer.

Code examples: <https://github.com/trekhleb/javascript-algorithms> and here

<https://github.com/barretlee/algorithms/tree/master/chapters>

and here : <https://github.com/humanwhocodes/computer-science-in-javascript>

Expensive Operation and Big O Notation

Big O notation it is the mathematical expression of **how long an algorithm takes to run** depending on **how long is the input**, usually talking about the worst case scenario. In practice, we use Big O Notation to **classify algorithms** by how they respond to **changes in input size**, so algorithms with the same growth rate are represented with the same Big O Notation. The letter **O** is used because the rate of growth of a function is also called **order of the function**.

Sometimes we want to focus on using less memory instead of (or in addition to) using less time (iterations) and usually there's a trade-off between saving time and saving space

$O(1)$ — “Order 1”

On this order, regardless of the complexity (number of items), the time (iterations) is constant.

$O(N)$ — “Order N ”

In this order, the worst case time (iterations) grows on par with the number of items

$O(N^2)$ — “Order *N squared*”

For this kind of order, the worst case time (iterations) is the square of the number of inputs. The time grows exponentially related to the number of inputs. (For loop inside for loop)

$O(2^n)$

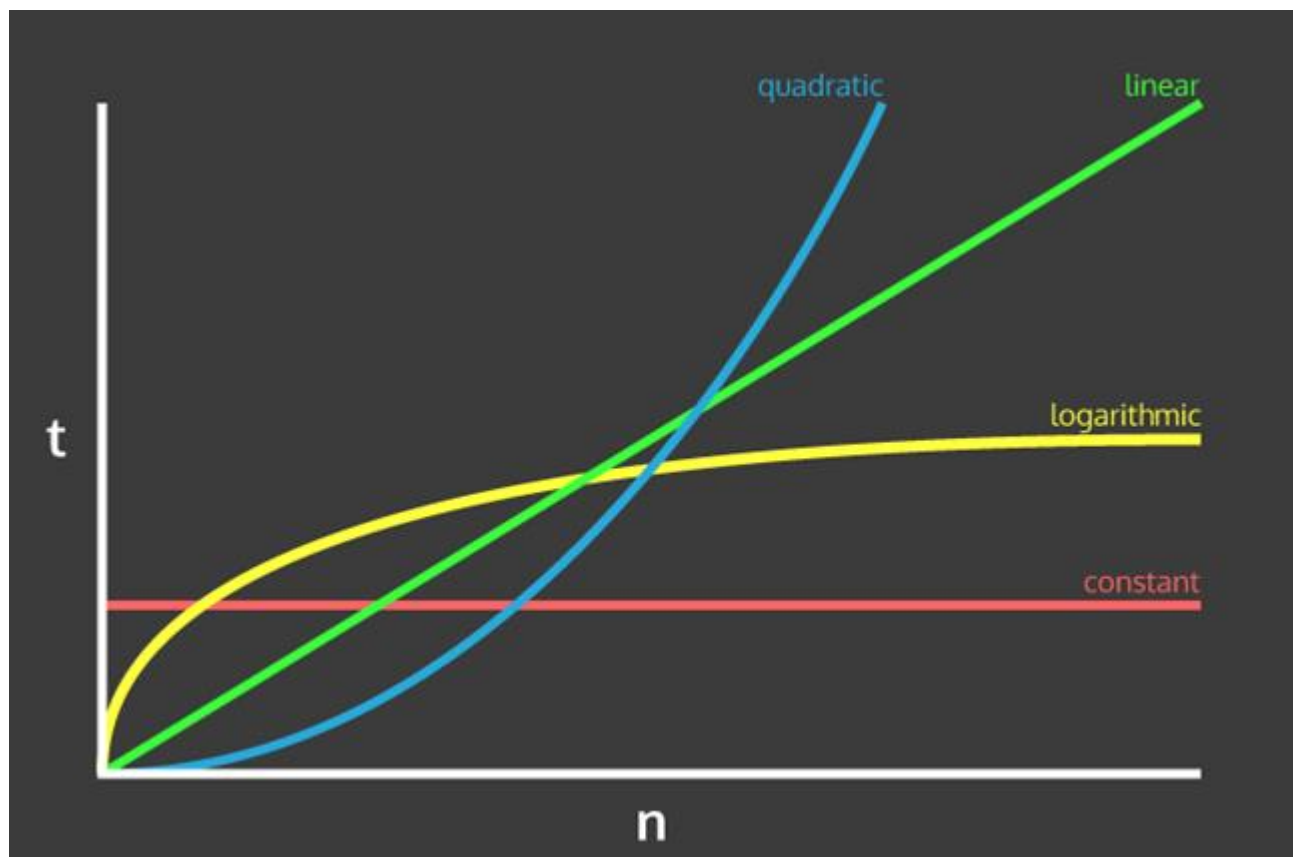
For each additional input, the time to run the algorithm doubles.

$O(n \log n)$ — “Order *N log N*”

These are the holy grail of search/sort algorithms, **they are usually the most efficient approach when dealing with large collections**. Instead of looking through the components one by one, they split the data in chunks and discard a large amount on every iteration, usually the half, or log base 2

Assuming we are using a log base 2, we could -ideally- find a specific element in a collection of one million elements using less than 20 iterations, if we scale the size of the collection to a billion we would require only less than 30 iterations.

The most popular of this algorithms is the Quicksort algorithm, which can be used to find a specific element or sort a list very efficiently. Another popular example of this order is the Merge-Sort algorithm



Here's a final recap:

- $O(1)$ — Constant Time: it only takes a single step for the algorithm to accomplish the task.
- $O(\log n)$ — Logarithmic Time: The number of steps it takes to accomplish a task are decreased by some factor with each step.
- $O(n)$ — Linear Time: The number of of steps required are directly related (1 to 1).
- $O(n^2)$ — Quadratic Time: The number of steps it takes to accomplish a task is square of n .
- $O(C^n)$ — Exponential: The number of steps it takes to accomplish a task is a constant to the n power (pretty large number).

Algorithms in JavaScript

Code example: <https://github.com/trekhleb/javascript-algorithms>

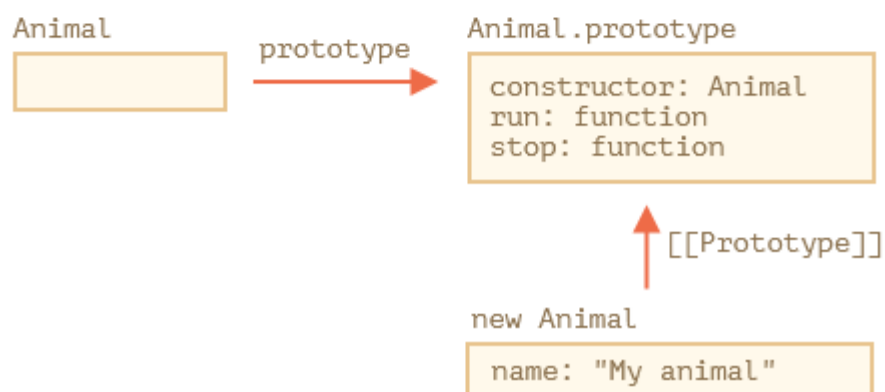
stao: <https://github.com/trekhleb/javascript-algorithms>

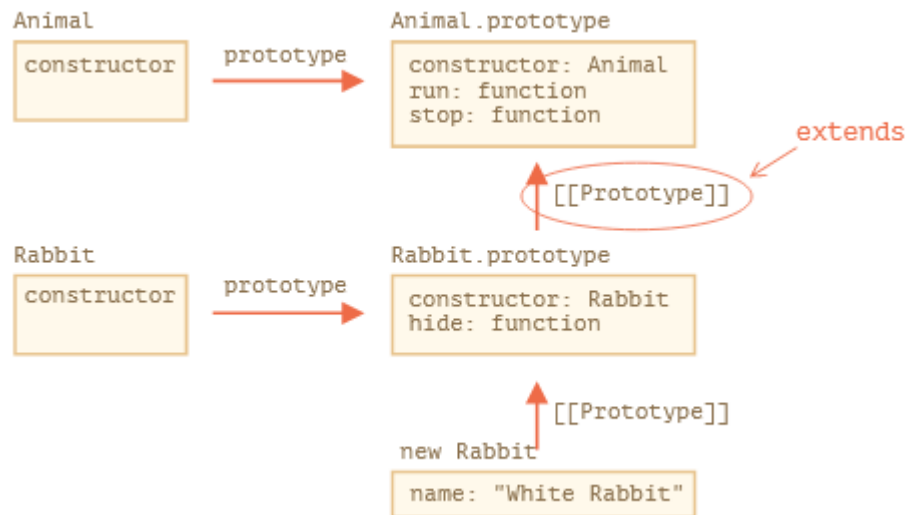
Inheritance, Polymorphism and Code Reuse

Class inheritance is a way for one class to extend another class.

```
1 class Animal {
2   constructor(name) {
3     this.speed = 0;
4     this.name = name;
5   }
6   run(speed) {
7     this.speed = speed;
8     alert(`${this.name} runs with speed ${this.speed}`);
9   }
10  stop() {
11    this.speed = 0;
12    alert(`${this.name} stands still.`);
13  }
14 }
15
16 let animal = new Animal("My animal");
```

Here's how we can represent `animal` object and `Animal` class graphically:





Object of Rabbit class have access both to Rabbit methods, such as `rabbit.hide()`, and also to Animal methods, such as `rabbit.run()`.

Internally, `extends` keyword works using the good old prototype mechanics. It sets `Rabbit.prototype[[Prototype]]` to `Animal.prototype`. So, if a method is not found in `Rabbit.prototype`, JavaScript takes it from `Animal.prototype`

For instance, to find `rabbit.run` method, the engine checks (bottom-up on the picture):

1. The rabbit object (has no run).
2. Its prototype, that is `Rabbit.prototype` (has `hide`, but not `run`).
3. Its prototype, that is (due to `extends`) `Animal.prototype`, that finally has the `run` method.

Constructors in inheriting classes must call `super(...)`, and (!) do it before using this

Old na new way: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Inheritance>

Design Patterns

Book: <https://addyosmani.com/resources/essentialjsdesignpatterns/book/#designpatternsjavascript>

Code: <https://github.com/fbeline/Design-Patterns-JS>

TODO functional design patterns: <https://www.youtube.com/watch?v=srQt1NAHYC0>

and here:

Using the concepts of pure function, function composition, currying/partial application and middlewares skillfully can alleviate a lot of design concern while making the application clean and concise. Functional libraries like Ramda and persistent immutable data structures library like Immutable-js greatly take away the burden of reimagination. Learning the concepts of functional programming is fun and inspiring. To further explore, [fantasyland/fantasy-land](#) is a great resource to understand the functional beauty in true sense.

Creational	Based on the concept of creating an object.
Class	
<i>Factory Method</i>	This makes an instance of several derived classes based on interfaced data or events.
Object	
<i>Abstract Factory</i>	Creates an instance of several families of classes without detailing concrete classes.
<i>Builder</i>	Separates object construction from its representation, always creates the same type of object.
<i>Prototype</i>	A fully initialized instance used for copying or cloning.
<i>Singleton</i>	A class with only a single instance with global access points.
Structural	Based on the idea of building blocks of objects.
Class	
<i>Adapter</i>	Match interfaces of different classes therefore classes can work together despite incompatible interfaces.
Object	
<i>Adapter</i>	Match interfaces of different classes therefore classes can work together despite incompatible interfaces.
<i>Bridge</i>	Separates an object's interface from its implementation so the two can vary independently.
<i>Composite</i>	A structure of simple and composite objects which makes the total object more than just the sum of its parts.
<i>Decorator</i>	Dynamically add alternate processing to objects.
<i>Facade</i>	A single class that hides the complexity of an entire subsystem.
<i>Flyweight</i>	A fine-grained instance used for efficient sharing of information that is contained elsewhere.
<i>Proxy</i>	A place holder object representing the true object.
Behavioral	Based on the way objects play and work together.
Class	
<i>Interpreter</i>	A way to include language elements in an application to match the grammar of the intended language.
<i>Template Method</i>	Creates the shell of an algorithm in a method, then defer the exact steps to a subclass.
Object	

<i>Chain of Responsibility</i>	A way of passing a request between a chain of objects to find the object that can handle the request.
<i>Command</i>	Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests.
<i>Iterator</i>	Sequentially access the elements of a collection without knowing the inner workings of the collection.
<i>Mediator</i>	Defines simplified communication between classes to prevent a group of classes from referring explicitly to each other.
<i>Memento</i>	Capture an object's internal state to be able to restore it later.
<i>Observer</i>	A way of notifying change to a number of classes to ensure consistency between the classes.
<i>State</i>	Alter an object's behavior when its state changes.
<i>Strategy</i>	Encapsulates an algorithm inside a class separating the selection from the implementation.
<i>Visitor</i>	Adds a new operation to a class without changing the class.

Also read here: <https://betterprogramming.pub/javascript-design-patterns-25f0faaaa15>

Partial Applications, Currying, Compose and Pipe

Here's another example of a higher-order function that returns a function:

```
function makeAdder(constantValue) {
  return function adder(value) {
    return constantValue + value;
  };
}
```

```
var add10 = makeAdder(10);
console.log(add10(20)); // prints 30
```


Closures

function composition in JavaScript

Function composition is a mechanism of combining multiple simple functions to build a more complicated one. The result of each function is passed to the next one. In mathematics, we often write something like: $f(g(x))$. So this is the result of $g(x)$ that is passed to f

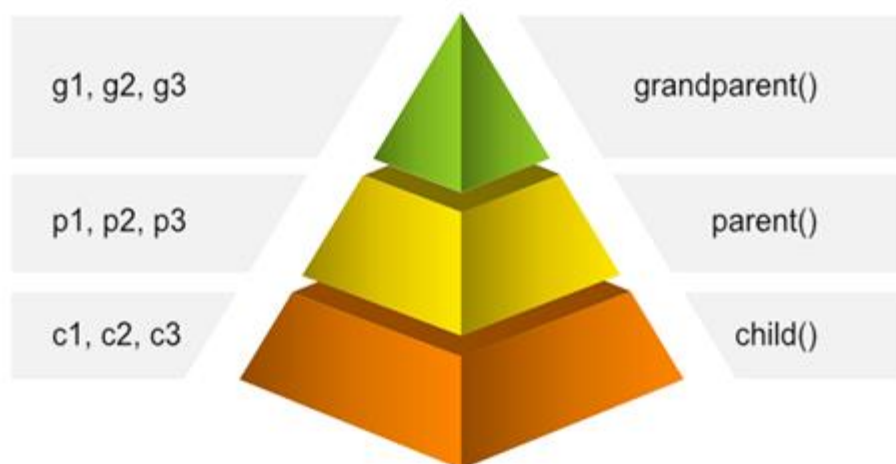
Simple example:

```
const add = (a, b) => a + b;  
const mult = (a, b) => a * b;  
add(2, mult(3, 5))
```

So our goal in this section is to create a high order function that take two or more functions and compose them

```
compose(function1, function2, ... , functionN): Function
```

Closures



Here's a contrived example of functions that use closures:

```
function g1, g2) {  
  var g  
  return function parent(p1, p2) {  
    var p3 = 33;  
    return function child(c1, c2) {  
      var c3 = 333;  
      return g1 + g2 + g3 + p1 + p2 + p3 + c1 + c2 + c3;  
    };  
  };  
}
```

implementation:

```
const compose = (...functions) => args => functions.reduceRight((arg, fn) =>
fn(arg), args);
```

```
const filter = cb => arr => arr.filter(cb);
const map = cb => arr => arr.map(cb);
```

```
compose(
  map(u => u.name),
  filter(u => u.age >= 18)
)(users) //["Jack", "Milady"]
```

In Functional Programming, functions are our building blocks. We write them to do very specific tasks and then we put them together like Lego™ blocks.

This is called *Function Composition*

Fat arrow notation for functions

```
var add10 = function(value) {
  return value + 10;
};
var mult5 = function(value) {
  return value * 5;
};
```

go to

```
var add10 = value => value + 10; // no name => and lambda to from param and body
var mult5 = value => value * 5;
```

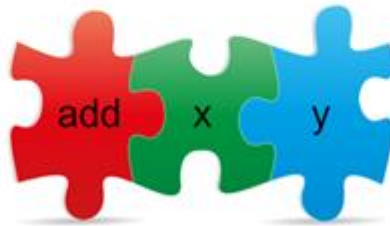
function value plus 10 then multiplay by 5

```
var mult5AfterAdd10 = value => 5 * (value + 10)
or
var mult5AfterAdd10 = value => mult5(add10(value));
```

In math, $f \circ g$ is functional composition and is read “ f composed with g ” or, more commonly, “ f after g ”. So $(f \circ g)(x)$ is equivalent to calling f after calling g with x or simply, $f(g(x))$

What is Currying?

Currying



We simply write an add function that uses 2 parameters but only takes 1 parameter at a time. *Curried* functions allow us to do this.

A Curried Function is a function that only takes a single parameter at a time.

Example:

```
var add = x => y => x + y
or
function add(x) {
    return function(y) {
        return x + y;
    }
}
```

In detail, the *add* function takes a single parameter, *x*, and returns a *function* that takes a single parameter, *y*, which will ultimately return the *result of adding x and y*

Read more: <https://stackoverflow.com/questions/36314/what-is-currying/62166542#62166542>

and here

<https://stackoverflow.com/questions/40675740/javascript-functional-programming-functions/40675883#40675883>

example of usage:

<https://stackoverflow.com/questions/113780/javascript-curry-what-are-the-practical-applications>

and here

<https://softwareengineering.stackexchange.com/questions/384529/a-real-life-example-of-using-curry-function>

ad

<https://stackoverflow.com/questions/39835395/currying-practical-implications>

Parameter order is important to fully leverage currying.

There is a way to reduce functions of more than one argument to functions of one argument, a way called **currying** after Haskell B. Curry. [1]

Currying is a process to reduce functions of more than one argument to functions of one argument with the help of lambda calculus

```
f(n, m) --> f'(n)(m)
```

```
multiply = (n, m) => (n * m)
```

```
multiply(3, 4) === 12 // true
```

```
curriedMultiply = (n) => ((m) => multiply(n, m))
```

```
triple = curriedMultiply(3)
```

```
triple(4) === 12 // true
```

Compose performs a right-to-left function composition since Pipe performs a left-to-right composition. So let's write the pipe high-order function

```
const pipe = (...functions) => args => functions.reduce((arg, fn) => fn(arg), args);
```

```
pipe(
  filter(u => u.age >= 18),
  map(u => u.name),
)(users) // ["Jack", "Milady"]
```

```
pipe(mapWords, reduceWords)(['foo', 'bar', 'baz']);
```

Some people prefer using pipe over compose because they find it more readable. At least, we can all agree that it is more natural!

Partial application means that you pre-applied a function partially regarding to any of its arguments.

```
f(n, m) --> f'(m)
```

```
multiply = (n, m) => n * m
```

```
multiply(3, 4) === 12 // true
```

```
triple = (m) => multiply(3, m)
```

```
triple(4) === 12 // true
```

Usage:

```
const handleChange = (fieldName) => (event) => {
  saveField(fieldName, event.target.value)
}
```

```
<input type="text" onChange={handleChange('email')} ... />
```

```
renderHtmlTag = tagName => content => `<${tagName}>${content}</${tagName}>`  
renderDiv = renderHtmlTag('div')  
renderH1 = renderHtmlTag('h1')  
  
console.log(  
  renderDiv('this is a really cool div'),  
  renderH1('and this is an even cooler h1')  
)
```

Curry as a term has been living for almost 40 years now and is an essential transformation in lambda calculus.

JavaScript is being used more as a functional language nowadays especially with native promises and arrow functions. To master the language you should have some knowledge in the field of lambda calculus and functional programming. JavaScript is being used more as a functional language nowadays especially with native promises and arrow functions. To master the language you should have some knowledge in the field of lambda calculus and functional programming

Partial application can be described as taking a function that accepts some number of arguments, binding values to one or more of those arguments, and returning a new function that only accepts the remaining, un-bound arguments.

What this means is that, given any arbitrary function, a new function can be generated that has one or more arguments "bound," or partially applied. And if you've been paying attention, you've realized by now that the previous examples have demonstrated partial application in a practical, albeit somewhat limited way. More here: <http://benalman.com/news/2012/09/partial-application-in-javascript/#partial-application>

also: <https://towardsdatascience.com/javascript-currying-vs-partial-application-4db5b2442be8>

JavaScript is being used more as a functional language nowadays especially with native promises and arrow functions. To master the language you should have some knowledge in the field of lambda calculus and functional programming. First thinkg you must to know is Pure Function

But in *Javascript* when writing pure functions one has to fight the temptation of mutating the *non-persistent data structures*. If you are someone who doesn't want to let go of the temptation, you can be rescued by *immutable.js*. Immutable.js provides many persistent immutable data structures which are highly efficient as it uses structural sharing via hash maps tries and vector tries as popularized by Clojure and Scala

-> Avoid array mutations

-> map and filter return a new array when applied upon an existing array which saves the

existing array from mutation

-> using *concat()*, *slice()* and *...spread*

```
//Inserting value at the end of an array

//Using push() - Impure
function insertWithPush(list, a) {
  return list.push(a) //modifies the existing array
}
//Using concat() - Pure
function insertWithConcat(list, a) {
  return list.concat([a]); //returns a new array
}

//Using ES6 ...spread operator instead of concat()
function insertWithSpread(list, a) {
  return [...list, a]; //returns a new array
}

//Remove value at a specified index

//Using splice() - Impure
function removeWithSplice(list, index) {
  return list.splice(index, 1); //modifies the existing array
}

//Using slice and concat() - Pure
function removeWithSliceAndConcat(list, index) {
  return list.slice(0, index).concat(list.slice(index+1));
}

//Using ES6 ...spread operator
function removeWithSpread(list, index) {
  return [...list.slice(0, index), ...list.slice(index+1)];
}

//Inserting a value at a specified index

//Using ES6 ...spread operator
function insertAtIndexWithSpread(list, value, index) {
  return [...list.slice(0, index), value, ...list.slice(index)];
}
```

Using *Object.assign()* and *...spread*

The **Object.assign()** method (ES6) is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object. *Properties in the target object will be overwritten by properties in the sources if they have the same key. Later sources' properties will similarly overwrite earlier ones.*

```
var obj = {
  foo: "foo",
  bar: "bar"
}

//Adding a new key-value

//Using obj[key] - Impure
function addToObject(obj, key, value) {
  return obj[key] = value; //Modifies the existing object
}

//Using Object.assign() - Pure
function addToObjectWithAssign(obj, key, value) {
  return Object.assign({}, obj, {
    key: value
  }); //returns a new object
}

//Using ...spread operator - ES7
function addToObjectWithSpread(obj, key, value) {
  return {...obj,
    key: value
  }; //returns a new object
}
```

Read more here: <https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-pure-function-d1c076bec976#.bm3rjd9c3>

Function composition is the act of pipelining the result of one function, to the input of another, creating an entirely new function.

```

function compose(f, g) {
  return function(a) {
    return f(g(a));
  }
}

function add1(a){
  return a+1;
}
function multiply5(a){
  return a*5;
}

var add1Multiply5=compose(add1,multiply5);

console.log(add1Multiply5(6)); //31

```

pipe which takes an array of functions to produce a new composed function.

```

const pipe = function(fns) {
  return function(item) {
    return fns.reduce(function(prev, fn) {
      return fn(prev);
    }, item);
  }
}

function add1(a) {
  return a + 1;
}

function multiply5(a) {
  return a * 5;
}
var add1multiply5 = pipe([add1, multiply5]);

console.log(add1multiply5(6)); //35

```


Functional programming middleware


```
function getTaxableAmount(account) {
  let next = account.calculateTax;
  return function(amount) {
    getTaxableAmountAPI(amount).then(function(taxAmount) {
      next(taxAmount);
    })
  }
}

function logger(account) {
  let next = account.calculateTax;
  return function(amount) {
    console.log("Taxable Amount :", amount);
    let tax = next(amount);
    console.log("Tax Calculated :", tax);
    return tax;
  }
}

//Middleware applied in a sequence
function applyMiddleware(account, middlewares) {
  middlewares = middlewares.slice()
  middlewares.reverse()

  middlewares.forEach(function(middleware) {
    account.calculateTax = middleware(account);
  });
}

applyMiddleware(account, [getTaxableAmount, logger]);
```



Using the concepts of pure function, function composition, currying/partial application and middlewares skillfully can alleviate a lot of design concern while making the application clean and concise. Functional libraries like Ramda and persistent immutable data structures library like Immutable-js greatly take away the burden of reimagination. Learning the concepts of functional programming is fun and inspiring. To further explore, [fantasyland/fantasy-land](#) is a great resource to understand the functional beauty in true sense.

Referential Transparency is a fancy term to describe that a pure function can safely be replaced by its expression. An example will help illustrate this.

Functional programming

Organizing Your Code

Organizing your code is also very important. This involves separating your functions into multiple files.

I like to create a file called `functional.js` and this is where I put `compose` and related functional functions.

All of the functions we created above I would put in `html.js`.

I am also creating a `dom.js` for DOM manipulation (you will see in the codepen.)

Breaking our code out into multiple library files allows us to reuse these functions in other projects.

Now when we write the main program in `main.js`, there will be very little code.

Code exmple here: <https://codepen.io/joelnet/pen/QdVpwB>

Ramda library in js for functional programming.

Also see: <https://www.telerik.com/blogs/practical-functional-javascript-ramda>

also: <https://hackernoon.com/the-beauty-in-partial-application-currying-and-function-composition-d885bdf0d574>