# ****How JavaScript works: functional style and how it compares to other approaches****

In reality js is not object-oriented language. It is prototype-based language and not a class-based language.

When you create an object using the class syntax it automatically gets the prototype.

In code below calling "new" keyword before function will changes it's context and "victor" is a new object and it can call speak().

let Person = function(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.speak = function() {

return `Hello, my name is ${this.name} and I am ${this.age}`;

}

let victor = new Person(`Victor`, 23)

console.log(victor.speak());

Extending our object just as we did in the class syntax above where **Work extends class Person**, can be donе with Prototypes like this:

let Work = function(name, age, work) {

Person.call(this, name, age);

this.work = work;

}

Functions regarder as values and can be passed to variables.

const getSum = function(num) {

return num + num;}

getSum(9);

cosnt addNum = getSum;

## Declarative vs. Imperative JavaScript

The imperative approach is more like stating all the steps you would need to achieve a problem. While the declarative approach just declares or says what you want to be done.

### The imperative approach

const filterArray = (array) => {

let filteredArray = [];

for(let i = 0; i < array.length; i++) {

if(array[i] > 5) {

filteredArray.push(array[i]);

}

}

return filteredArray;

}

const array = [1, 2, 3, 4, 5, 6, 7, 8]

filterArray(array)

Rather than tell the computer what we want we just give instructions in steps on what we want to achieve. Our steps include:

* Declare an empty array
* Loop through a given array
* if/else if each item is greater than 5
* Push each element that passes the test into the empty array declared earlier
* Display our new array

### ****The declarative approach in JavaScript:****

// Filter method to give us a new array

const filterArray = array => array.filter(x => x > 5);

const array = [1, 2, 3, 4, 5, 6, 7, 8];

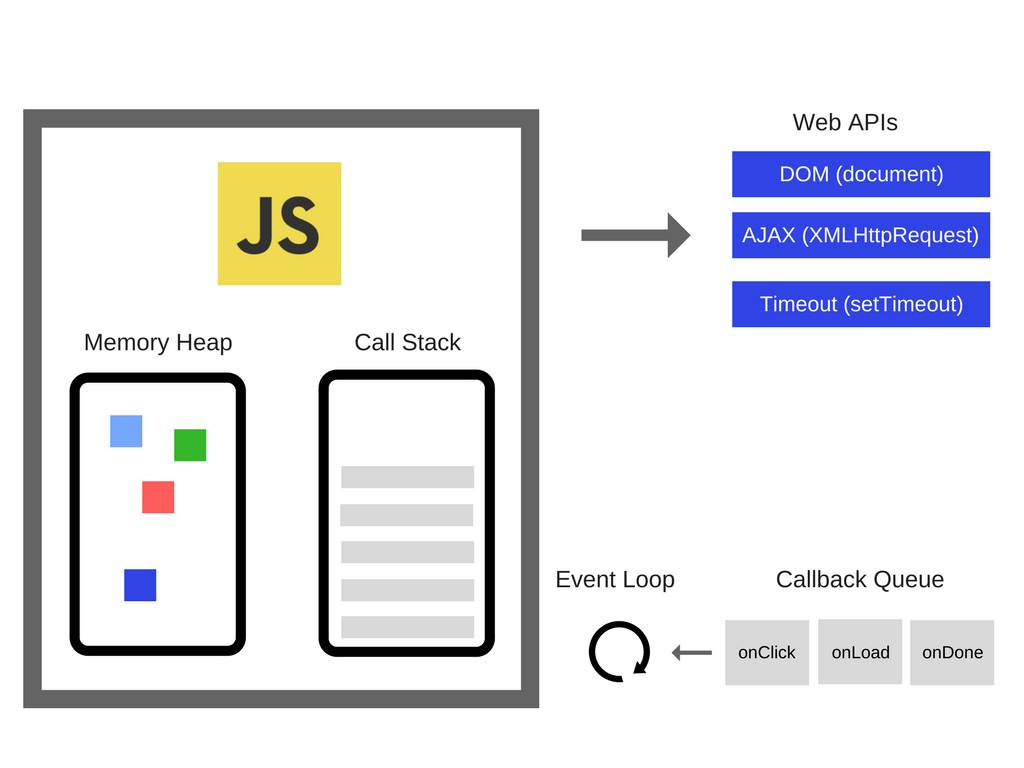
console.log(filterArray(array)); // [6, 7, 8]

## Why is the functional approach preferred by most?

* This also offers code reusability where functions are composable (treated as components) just as is mostly done in React.
* Debugging is easier here
* Any developer can read and understand your code fast. Because you write what you think and not how the computer should think for you.

# How JavaScript works: an overview of the engine, the runtime, and the call stack

## ****The Runtime****

There are APIs in the browser that have been used by almost any JavaScript developer out there (e.g. “setTimeout”). Those APIs, however, are not provided by the Engine.

## The Call Stack

JavaScript is a single-threaded programming language, which means it has a single Call Stack ,therefore it can do one thing at a time.

Each entry in the Call Stack is called a **Stack Frame**.

**“Blowing the stack”** this happens when you reach the maximum Call Stack size.

While the Call Stack has functions to execute, the browser can’t actually do anything else — it’s getting blocked. So, how can we execute heavy code without blocking the UI and making the browser unresponsive? Well, the solution is **asynchronous callbacks**.

# How JavaScript works: inside the V8 engine + 5 tips on how to write optimized code

Before version 5.9 of V8 came out (released earlier this year), the engine used two compilers:

* full-codegen — a simple and very fast compiler that produced simple and relatively slow machine code.
* Crankshaft — a more complex (Just-In-Time) optimizing compiler that produced highly-optimized code.

The V8 Engine also uses several threads internally:

* The main thread does what you would expect: fetch your code, compile it and then execute it
* There’s also a separate thread for compiling, so that the main thread can keep executing while the former is optimizing the code
* A Profiler thread that will tell the runtime on which methods we spend a lot of time so that Crankshaft can optimize them
* A few threads to handle Garbage Collector sweeps

## Steps of compiling code in V8

The engine use two compiler:

1. full-codegen: a simple and very fast compiler that produced simple and relatively slow machine code.
2. Crankshaft: a more complex (just-in-time) optimizing compiler that produce highly-optimized code.

**V8 use many threads internally:**

1. Main thread: fetch code, compile it then execute;
2. Separate thread: while main is compiling it is optimizing code;
3. Profiler thread: will tell the runtime on which method we spend a lot of time so that Crankshaft can optimize them;
4. A few threads to handle Garbage collector sweeps.

When first executing the JavaScript code, V8 leverages full-codegen which directly translates the parsed JavaScript into machine code without any transformation. This allows it to start executing machine code very fast. Note that V8 does not use intermediate bytecode representation this way removing the need for an interpreter.

When your code has run for some time, the profiler thread has gathered enough data to tell which method should be optimized.

Next, Crankshaft optimizations begin in another thread. It translates the JavaScript abstract syntax tree to a high-level static single-assignment (SSA) representation called Hydrogen and tries to optimize that Hydrogen graph. Most optimizations are done at this level.

## Levels of optimizing

### Inlining

Process of replacing the call site with the body of the called function

### Hidden classes

Object’s properties in js can change dynamically during the runtime . in languages like Java that properties can’t change dynamically they store in a memory space and there is fixed-offset between them, but it can’t be done in JS and dictionary like method (store memory place of the property and save that place) is very inefficient, so V8 uses hidden classes.

function Point(x, y) {

this.x = x;

this.y = y;

}

var p1 = new Point(1, 2);

First when new called, JS creates a hidden class C0, when this.x=x executed JS will creates hidden class C1 that based on C0, C1 describes the location in the memory where the property x can be found and V8 will update C0, in this case, “x” is stored at offset 0. JS will add a transition to class C0 means that if a property added to point object the hidden class should switch from C0 to C1. Then this process will happen for this.y=y , A new hidden class called “C2” is created, a class transition is added to “C1” stating that if a property “y” is added to a Point object (that already contains property “x”) then the hidden class should change to “C2”, and the point object’s hidden class is updated to “C2”.

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

var p1 = new Point(1, 2);  
p1.a = 5;  
p1.b = 6;

var p2 = new Point(3, 4);  
p2.b = 7;  
p2.a = 8;

In P2 JS will create another hidden class, because it b is in the first place.

**In such cases, it’s much better to initialize dynamic properties in the same order so that the hidden classes can be reused.**

## Inline cashing

So how does it work? V8 maintains a cache of the type of objects that were passed as a parameter in recent method calls and uses this information to make an assumption about the type of object that will be passed as a parameter in the future. If V8 is able to make a good assumption about the type of object that will be passed to a method, it can bypass the process of figuring out how to access the object’s properties, and instead, use the stored information from previous lookups to the object’s hidden class.

Whenever a method is called on a specific object, the V8 engine has to perform a lookup to the hidden class of that object in order to determine the offset for accessing a specific property. After two successful calls of the same method to the same hidden class, V8 omits the hidden class lookup and simply adds the offset of the property to the object pointer itself. For all future calls of that method, the V8 engine assumes that the hidden class hasn’t changed, and jumps directly into the memory address for a specific property using the offsets stored from previous lookups. This greatly increases execution speed.

## Compilation to machine code

In the middle of compiling code V8 will turn from an unoptimized code to the optimized one and it will not start method execution from start.

## How to write optimized JavaScript

1. Order of object properties: always instantiate your object properties in the same order so that hidden classes, and subsequently optimized code, can be shared.
2. Dynamic properties: adding properties to an object after instantiation will force a hidden class change and slow down any methods that were optimized for the previous hidden class. Instead, assign all of an object’s properties in its constructor.
3. Methods: code that executes the same method repeatedly will run faster than code that executes many different methods only once (due to inline caching).
4. Arrays: avoid sparse arrays where keys are not incremental numbers. Sparse arrays which don’t have every element inside them are a hash table. Elements in such arrays are more expensive to access. Also, try to avoid pre-allocating large arrays. It’s better to grow as you go. Finally, don’t delete elements in arrays. It makes the keys sparse.
5. Tagged values: V8 represents objects and numbers with 32 bits. It uses a bit to know if it is an object (flag = 1) or an integer (flag = 0) called SMI (SMall Integer) because of its 31 bits. Then, if a numeric value is bigger than 31 bits, V8 will box the number, turning it into a double and creating a new object to put the number inside. Try to use 31 bit signed numbers whenever possible to avoid the expensive boxing operation into a JS object.