# ****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];

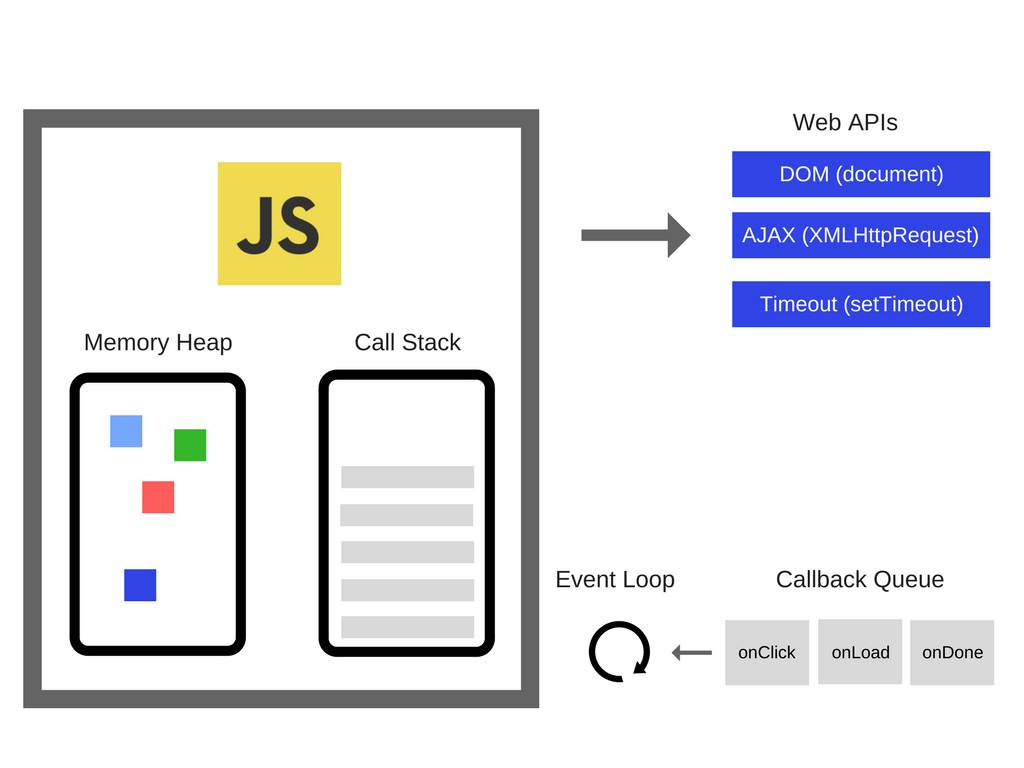
console.log(filterArray(array)); // [6, 7, 8]

## 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