Difference Between var and let

The difference is scoping. var is scoped to the nearest function block and let is scoped to the nearest *enclosing* block, which can be smaller than a function block. Both are global if outside any block.

Also, variables declared with let are not accessible before they are declared in their enclosing block. As seen in the demo, this will throw a *ReferenceError* exception.

Global:
They are very similar when used like this outside a function block.

```
let me = 'go'; // globally scoped
var i = 'able'; // globally scoped
```

However, global variables defined with `let` will not be added as properties on the global `window` object like those defined with `var`.

```
console.log(window.me); // undefined
console.log(window.i); // 'able'
```

Function:
They are identical when used like this in a function block.

```
function ingWithinEstablishedParameters() {
        let terOfRecommendation = 'awesome worker!'; //function block scoped
        var sityCheerleading = 'go!'; //function block scoped
}
```

Difference Between var and let

Block:

Here is the difference. `let` is only visible in the `for()` loop and `var` is visible to the whole function.

```
function allyIlliterate() {
	//tuce is *not* visible out here
	for( let tuce = 0; tuce < 5; tuce++ ) {
		//tuce is only visible in here (and in the for() parentheses)
		//and there is a separate tuce variable for each iteration of the loop
	}

	//tuce is *not* visible out here
}

function byE40() {
	 //nish *is* visible out here
	 for( var nish = 0; nish < 5; nish++ ) {
		//nish is visible to the whole function
	}
	//nish *is* visible out here
}
```

Redeclaration:

Assuming strict mode, `var` will let you re-declare the same variable in the same scope.

```
'use strict';
var me = 'foo';
var me = 'bar'; // No problem, 'me' is replaced.
```

On the other hand, `let` will not:

```
'use strict';
let me = 'foo';
let me = 'bar'; // SyntaxError: Identifier 'me' has already been declared
```

Javascript datatypes

JavaScript Primitives

A **primitive value** is a value that has no properties or methods.

A **primitive data type** is data that has a primitive value.

JavaScript defines 5 types of primitive data types:

- string
- number
- boolean
- null
- undefined

Javascript datatypes

Symbol datatype

- Every symbol value returned from Symbol() is unique.
- A symbol value may be used as an identifier for object properties; this is the data type's only purpose.
- To avoid name clashes

```
let newVal = Symbol();
let newVal1 = Symbol();

console.log(newVal); //Symbol()
console.log(typeof(newVal)); //symbol
console.log(newVal === newVal1); //false

const obj={};
obj[newVal]=1;
obj[newVal1]=2;
console.log(obj[newVal]) //1
```

```
const symbol1 = Symbol();
const symbol2 = Symbol(42);
const symbol3 = Symbol('foo');

console.log(typeof symbol1);// expected output:

"symbol"console.log(symbol3.toString());// expected output:

"Symbol(foo)"console.log(Symbol('foo') === Symbol('foo'));// expected output: false
```

Type Conversions

```
let num = 20.20;
let numString = num.toString();

Console.log(num.__proto__);
```

Exercise 1

1. Create a string x using a string literal, with value 'I love loving your love'.

   find length of x.
   find the first and the last index of the word 'love'.
   replace 'love' with 'bunny'.
   find a char at index 12.
   reverse the string such that you get 'love your loving love I'

2. Find the maximum possible integer value

3. create a number y using number literal with value of 25.555
   convert y to an integer value of 25.
   convert y to a string.
   convert y to a boolean.

Variable Scopes

- Global Scope
- Block Scope
- Lexical Scope
- Scope Chain

Global Scope

Global Scope

```
let outer=1; //global scope
console.log(outer);
console.log(window.outer); // undefined

window.global = 2; //global scope
console.log(window.global);

x=19; // global scope same as window.x=19
console.log(window.x);
```

```
if(true){
 let local=0; //local scope
 dontUse=-1; //global scope
}

if(true){
 console.log(outer);
 console.log(global);
console.log(dontUse);
 console.log(local);
}
```

Block Scope

Block Scope

```
{
 let fromBlock = 0; // Block scope
 console.log(fromBlock);  //0
 let fromBlock = 1; //re-declaration error
fromBlock=1; //valid
}
 console.log(fromBlock); //error
```

Nested Block

```
let fromGlobal = 2; // Block scope
if(true){
  let fromIf = 1; // lexical scope
  {
    console.log(fromIf);  //1
    console.log(fromGlobal );  //2
    }
   {
    console.log(fromIf);  //1
    }
}
```

Scope Chain

Scope Chain

```
let fromGlobal = 2; // Block scope
if(true){
  let fromIf = 1; // lexical scope
  console.log(fromIf);  //1
  {
    let fromIf = 2; //shadowing outer block variable
    console.log(fromIf);  //2
   }
   {
    console.log(fromIf);  //1
    }
}
```

```
let fromGlobal = 2; // Block scope
if(true){
  let fromIf = 1; // lexical scope
  console.log(fromIf);  //1
  {
     console.log(fromIf);  //2
      let fromIf = 2; //fromIf not defined
   }
   {
    console.log(fromIf);  //1
    }
}
```

Arrays

```
let groceryList = ["bread", "milk", "eggs", "tomato", "milk"];

console.log(Array.isArray(groceryList));

console.log(groceryList.constructor === Array)

console.log(groceryList.length);
console.log(groceryList.indexOf("candy"));//-1
console.log(groceryList.indexOf("milk",2));//4
```

Mutable Arrays

```
let groceryList = ["bread", "milk", "eggs", "tomato"];
groceryList.push("candy");
console.log(groceryList)
let item=groceryList.pop();
console.log(item);
console.log(groceryList)
groceryList.shift(); // left shift by 1
console.log(groceryList);
groceryList.unshift("eggs"); //insert "eggs" in first element
console.log(groceryList);
groceryList.unshift("potatoes");
console.log(groceryList);
groceryList = ["bread", "milk", "eggs", "tomato"];
console.log(groceryList);
groceryList.splice(1,1); // removes value at index 1
console.log(groceryList);
groceryList.splice(1,1,"spinach");
console.log(groceryList);//replaces value at index 1 with "spinach"
groceryList.splice(1,2);
console.log(groceryList);
groceryList.splice(1,2,"sugar","oranges");
console.log(groceryList);
```

Iterating Arrays

```
let groceryList = ["bread", "milk", "eggs", "tomato", "milk"];
```

```
//Method -1
for(let i=0;i<groceryList.length;i++){
   console.log(groceryList[i]); // console.log($
}

for(let i=0;i<groceryList.length;i++){
  console.log(`${groceryList[i]}`);
}

//Method -2 : forEach
groceryList.forEach(
  (item,index)=>{
    console.log(`${index}-${item}`)
  }
)
```

```
//Method-3
for(let [index, item] of groceryList.entries()){
  console.log(`${index}-${item}`)
}

for(let item of groceryList){
  console.log(item);
}
//same as
for(let item of groceryList.values()){
  console.log(item);
}

//prints 0 1 2 3 4
for(let key of groceryList.keys()){
  console.log(key);
}
```

Multi Dimensional Arrays

```
const matrix = [
                [10],
                [20,24],
                [40,60,70]
              ];

console.log(matrix);
console.log(matrix.length);
console.log(matrix[0].length);
console.log(matrix[1].length);


for(let i=0;i<matrix.length;i++){
  for(let j=0;j<matrix[i].length;j++){
    console.log(matrix[i][j]);
  }
}
```

```
const numArray = [4,1,20,-10];
let max1= Number.NEGATIVE_INFINITY;
let max2 = Number.NEGATIVE_INFINITY;

for(let val of numArray){
  if(val>max1){
    max2 = max1;
    max1 = val;
  } else {
    if(val > max2){
      max2 = val;
    }
  }

}
console.log(max1)
console.log(max2)
```

Array Constructor

Array.from()

```
console.log(Array.from('foo'));
// expected output: Array ["f", "o", "o"]

console.log(Array.from([1, 2, 3], x => x + x));
// expected output: Array [2, 4, 6]
```

`Array.isArray()` method determines whether the passed value is an `Array`

```
Array.isArray([1, 2, 3]);  // true
Array.isArray({foo: 123}); // false
Array.isArray('foobar');   // false
Array.isArray(undefined);  // false
```

The Array.of() method creates a new Array instance from a variable number of arguments, regardless of number or type of the arguments.

`Array.of(7)` creates an array with a single element, 7, whereas **`Array(7)`** creates an empty array with a `length` property of 7

```
Array.of(7);       // [7]
Array.of(1, 2, 3); // [1, 2, 3]

Array(7);          // [ , , , , , , ]
Array(1, 2, 3);    // [1, 2, 3]
```

Object Introduction

In JavaScript, almost "everything" is an object.

- Booleans can be objects (if defined with the new keyword)
- Numbers can be objects (if defined with the new keyword)
- Strings can be objects (if defined with the new keyword)
- Dates are always objects
- Maths are always objects
- Regular expressions are always objects
- Arrays are always objects
- Functions are always objects
- Objects are always objects

All JavaScript values, except primitives, are objects.

Creating a JavaScript Object

There are different ways to create new objects:

- Define and create a single object, using an object literal.
- Define and create a single object, with the keyword new.
- Define an object constructor, and then create objects of the constructed type
- In ECMAScript 5, an object can also be created with the function Object.create()

1. Using an Object Literal

```
Ex.
var person = {firstName: "John", lastName: "Doe", age: 50, eyeColor: "blue"};
```

2. Using the JavaScript Keyword new

```
Ex.
var person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

```
var person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"}

var x = person;
x.age = 10;

The above will change both x.age and person.age
Note: x are person are references pointing to same object
```

Creating a JavaScript Object

JavaScript Object Constructors

```
Ex.
// Constructor function for Person objects
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

// Create a Person object
var myFather = new Person("John", "Doe", 50, "blue");
console.log(myFather.age);
```

About **this**

```
var test = {
  prop: 42,
  func: function() {
    return this.prop;
  }
};

console.log(test.func());
// expected output: 42
```

In the global execution context (outside of any function), this refers to the global object whether in strict mode or not.

```
// In web browsers, the window object is also the global object:
console.log(this === window); // true
a = 37;
console.log(window.a); // 37
this.b = "MDN";
console.log(window.b) // "MDN"
console.log(b) // "MDN"
```

JavaScript Properties

- Properties are the values associated with a JavaScript object.
- A JavaScript object is a collection of unordered properties.
- Properties can usually be changed, added, and deleted, but some are read only.

Accessing JavaScript Properties

```
objectName.property     // person.age

objectName["property"]  // person["age"]

objectName[expression]  // x = "age"; person[x]
```

Adding and Deleting New Properties

We can add or delete new properties to an existing object.

```
person.nationality = "English";

delete person.age;
```

for...in Loop

```
for (variable in object) {
  // code to be executed
}
```

Property Attributes

- All properties have a name. In addition they also have a value.
- The value is one of the property's attributes.
- **Other attributes are: enumerable, configurable, and writable**.
- These attributes define how the property can be accessed (is it readable?, is it writable?)
- In JavaScript, all attributes can be read, but only the value attribute can be changed (and only if the property is writable).
- (ECMAScript 5 has methods for both getting and setting all property attributes)

The static method **Object.defineProperty()** defines a new property directly on an object, or modifies an existing property on an object, and returns the object.

```
const object1 = {};

Object.defineProperty(object1, 'property1', {
  value: 42,
  writable: false
});

object1.property1 = 77; // throws an error in strict mode
console.log(object1.property1); // expected output: 42
```

Object Introduction

```
const profile = {
  name : "John",
  age : 30,
  single : false,
  "got a job" : true,
  kids: [{
        name : "peter",
        age : 3
      },
      {
        name : "kate",
        age : 2
      }
      ]
};
```

```
console.log(profile.name);
//same as
console.log(profile["name"]);
console.log(profile["got a job"]);

console.log(profile.kids);

console.log(profile.kids[1].name);

profile.kids[1].age += 1;
console.log(profile.kids[1].age);
```

```
profile.kids.push({name: "jake",age: 1});
console.log(profile.kids);

profile.kids.pop();
console.log(profile.kids);
console.log("--------------------")
delete profile["got a job"]
console.log(profile);
console.log("--------------------")
delete profile.kids[1].age;
console.log(profile);
console.log(profile.hasOwnProperty('name'))
console.log(profile.hasOwnProperty('got a job'))
```

JavaScript Object Prototypes

All JavaScript objects inherit properties and methods from a prototype

The prototype is an object that is associated with every function and object by default in JavaScript, where function's prototype property is accessible and modifiable and Object's prototype property (aka attribute) is not visible.
Every function includes prototype object by default.

All JavaScript objects inherit properties and methods from a prototype:

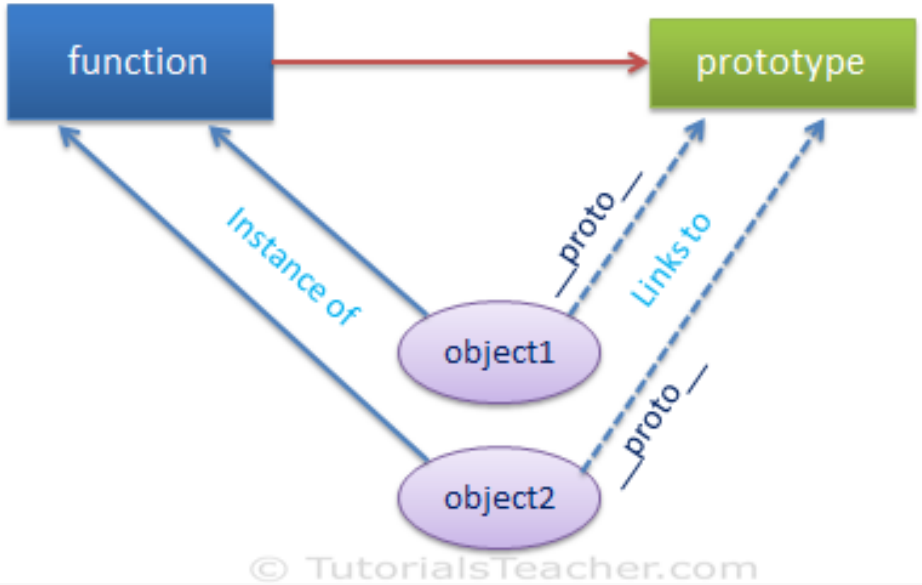
- Date objects inherit from Date.prototype
- Array objects inherit from Array.prototype
- Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain.

Date objects, Array objects, and Person objects inherit from Object.prototype.

JavaScript Object Prototypes

Every object which is created using literal syntax or constructor syntax with the new keyword, includes **__proto__** property that points to prototype object of a function that created this object.



Function's prototype property can be accessed using <function-name>.prototype.
However, an object (instance) does not expose prototype property, instead you can access it using __proto__.

```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}

var studObj = new Student();

console.log(Student.prototype); // object
console.log(studObj.prototype); // undefined
console.log(studObj.__proto__); // object

console.log(typeof Student.prototype); // object
console.log(typeof studObj.__proto__); // object

console.log(Student.prototype === studObj.__proto__ ); // true
```

JavaScript Object Prototypes

An object's prototype property is invisible. Use **Object.getPrototypeOf(obj)** method instead of __proto__ to access prototype object

```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}

var studObj = new Student();

Student.prototype.sayHi= function(){
    alert("Hi");
};

var studObj1 = new Student();
var proto = Object.getPrototypeOf(studObj1);  // returns
Student's prototype object

alert(proto.constructor); // returns Student function
```

The prototype object is being used by JavaScript engine in two things,

1) to find properties and methods of an object
2) to implement inheritance in JavaScript.

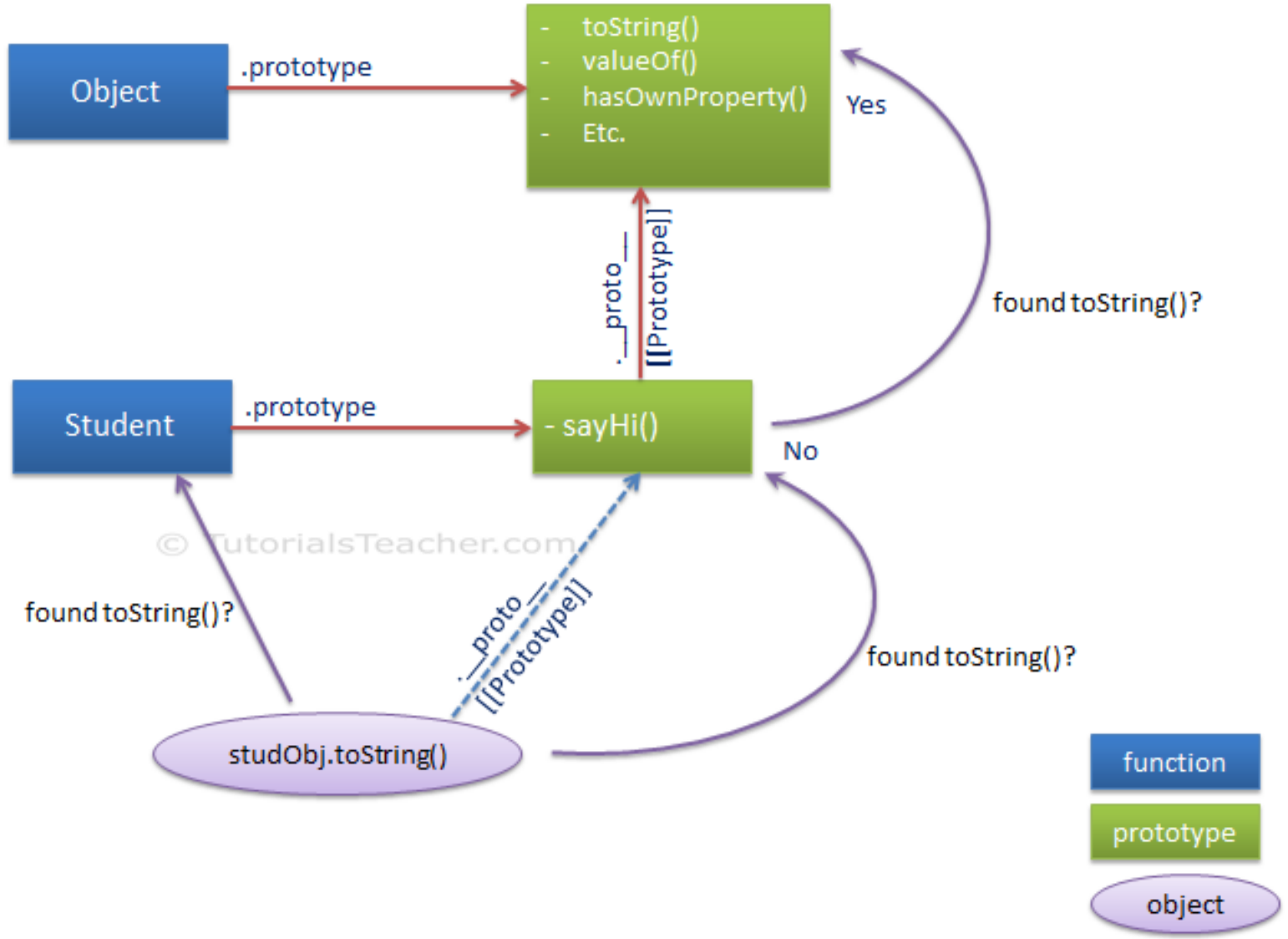
```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}

Student.prototype.sayHi = function(){
    alert("Hi");
};

var studObj = new Student();
console.log(studObj.toString());
```

JavaScript Object Prototypes

In the above example, toString() method is not defined in Student, so how and from where it finds toString()?



JavaScript Object Prototypes

To Attach property to a object

```
function Student() {
    this.name = 'John';
    this.gender = 'Male';
}

var studObj1 = new Student();
studObj1.age = 15;
alert(studObj1.age); // 15

var studObj2 = new Student();
alert(studObj2.age); // undefined
```

To attach new properties (or methods) to **all existing objects** of a given type.

- The JavaScript `prototype` property allows you to add new properties to object constructors

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}

Person.prototype.nationality = "English";

Person.prototype.name = function() {
  return this.firstName + " " + this.lastName;
};
```

Prototypal Inheritance in JavaScript

Classical Inheritance Vs Prototypal Inheritance

Function Constructors

```
//function constructor
const Car = function(color){
  //_color is internal property of Car
  this._color=color;
};
const blueCar = new Car('blue');
const redCar = new Car('red');
// The following doesnt create object
const grayCar = Car ('gray');
//undefined
console.log(grayCar);
//_color is added as property to window object
console.log(window._color);
//solution to avoid this is
const MyCar = function(color){
        //_color is internal property of Car
        if(!new.target){
          throw "car must be called with new keyword"
        }
        this._color=color;
};
const MyGrayCar = MyCar('gray');
```

Extending Function Constructors

```
//function constructor
const Car = function(color){
  //_color is internal property of Car
  this._color=color;
};
Car.protoype.getColor = function(){
                return this._color;
  }
const blueCar = new Car('blue');
console.log(blueCar.getColor());
// console.dir(Car); inspect
const ToyCar = function(model){
  Car.call(this,color);
  this._model = model;
};
ToyCar.prototype = Object.create(Car.prototype);
ToyCar.prototype.getModel = function(){
      return this.model;
}
console.dir(ToyCar); //inspect
const speedyBlueCar = new ToyCar('blue','speedy');
console.log(speedyBlueCar); //inspect
console.log(speedyBlueCar.getColor(), speedyBlueCar.getModel());
console.log(speedyBlueCar);//inspect
```

The ES5 **`Object.create()`** method creates a new object, using an existing object as the prototype of the newly created object.

Rather than using Function constructors, use ES6 Classes which Abstracts function constructors

Extending Function Constructors

Another Example

```
function Person(firstName, lastName) {
    this.FirstName = firstName || "unknown";
    this.LastName = lastName || "unknown";
}

Person.prototype.getFullName = function () {
    return this.FirstName + " " + this.LastName;
}
function Student(firstName, lastName, schoolName, grade){
    Person.call(this, firstName, lastName);
    this.SchoolName = schoolName || "unknown";
    this.Grade = grade || 0;
}
Student.prototype = Person.prototype;
//Student.prototype = new Person();
//Student.prototype.constructor = Student;

var std = new Student("James","Bond", "XYZ", 10);
console.log(std.getFullName()); // James Bond
console.log(std instanceof Student); // true
console.log(std instanceof Person); // true
```

JavaScript ES5 Object Methods

```
// Adding or changing an object property
Object.defineProperty(object, property, descriptor)

// Accessing Properties
Object.getOwnPropertyDescriptor(object, property)

// Returns all properties as an array
Object.getOwnPropertyNames(object)

// Returns enumerable properties as an array
Object.keys(object)

// Accessing the prototype
Object.getPrototypeOf(object)

// Prevents adding properties to an object
Object.preventExtensions(object)
```

```
var person = {
  firstName: "John",
  lastName : "Doe",
  language : "EN"
};
// Change a property
Object.defineProperty(person, "language",
                                {value : "NO"});
```

```
const object1 = {
  a: 1,
  b: 2,
  c: 3
};

console.log(Object.getOwnPropertyNames(object1));
```

JavaScript ES5 Object Methods

ES5 allows getters and setters to be changed:

```
// Defining a getter
get: function() { return language }
// Defining a setter
set: function(value) { language = value }
```

```
var person = {
   firstName: 'Jimmy',
   lastName: 'Smith',
   get fullName() {
      return this.firstName + ' ' + this.lastName;
   },
   set fullName (name) {
      var words = name.toString().split(' ');
      this.firstName = words[0] || '';
      this.lastName = words[1] || '';
   }
}

person.fullName = 'Jack Franklin';
console.log(person.firstName); // Jack
console.log(person.lastName) // Franklin
console.log(person.fullName);
```

Array of Objects

Iterating through an object

```
const profile = {
  name : "John",
  age : 30,
  single : false,
  "got a job" : true,
  kids: [{
        name : "peter",
        age : 3
      },
      {
        name : "kate",
        age : 2
      }
      ]
};

console.log( profile.length);//undefined

//prints only keys of profile
for(prop in profile){
  console.log(prop);
}

for(prop in profile){
  console.log(`${prop} : ${profile[prop]}` );
}

console.log("Using of loop");

for(let prop of Object.keys(profile)){
  console.log(`${prop} : ${profile[prop]}` );
}

console.log('Iterating throuh child objects');
for(let prop of Object.keys(profile)){
  console.log(`${prop} : ${profile[prop]}` );
}
```

ES6 Classes

The **Function constructor** creates a new Function object which is widely creating objects. But now with ES6 we have better a way of creating objects.

OOP in JavaScript with ES6 : In ES6 we can create **classes**.

```
class User {
  constructor(name,age,email){
    this.name=name;
    this.age=age;
    this.email=email;
  }

  increaseAge(){
    this.age += 1;
  }
  static staticMethod(){
    console.log('I am a static method');
  }
}
```

```
const user1=new User("Jeff",30,"jeff@gmail.com");
const user2=new User("Sara",23,"sara@gmail.com");
const user3=new User("Bob",40,"bob@gmail.com");

console.log(user1.age)
user1.increaseAge();
console.log(user1.age)
//user1.staticMethod();// returns TypeError
User.staticMethod();
```

getters and setters of ES6 Classes

An important part of **encapsulation** is that data (object properties) should not be directly accessed or modified from outside the object. To access or modify a property we would use a **getter** (access) or a **setter** (modify)

```
class User {
  constructor(name,age,email){
    this.name=name;
    this.age=age;
    this.email=email;
  }

  get name(){
    return this._name;
  }
  set name(name){
    this._name=name;
  }
}

const user1=new User("Jeff",30,"jeff@gmail.com");
console.log(user1.name)
user1.name="Jim";
console.log(user1.name)
```

Note: The underscore in front of the properties is another example of a convention. It also prevents a stack overflow when calling our methods. Also, note that we are calling “jeff.name” not “jeff._name”. So the output is being returned from our getter.

Note: Currently there is no native support for private properties in JavaScript (it is possible to mimic private properties but we’re not going to go into that). So all the properties we’ve declared can be directly accessed from outside the object. But following **encapsulation** is best practice for OOP.

Inheritance

```
class User {
  constructor(name,age,email){
    this.name=name;
    this.age=age;
    this.email=email;
  }

  get name(){
    return this._name;
  }

  set name(name){
    this._name=name;
  }
}

)
```

```
class Administrator extends User{
  constructor(name,age,email,role){
    super(name,age,email);
    this._role=role;
  }

  get role(){
    return this._role;
  }

  set role(role){
    this._role=role;
  }
}


const admin=new Administrator("Sara",30,"sara@gmail.com","Admin");
console.log(admin.name)
console.log(admin.role
```

Maps

Map

Map is a collection of keyed data items, just like an `Object`. But the main difference is that `Map` allows keys of any type.

The main methods are:

- `new Map()` – creates the map.
- `map.set(key, value)` – stores the value by the key.
- `map.get(key)` – returns the value by the key, `undefined` if `key` doesn't exist in map.
- `map.has(key)` – returns `true` if the `key` exists, `false` otherwise.
- `map.delete(key)` – removes the value by the key.
- `map.clear()` – clears the map
- `map.size` – returns the current element count.

Maps

```
let map = new Map();
map.set('1', 'str1'); // a string key
map.set(1, 'num1'); // a numeric key
map.set(true, 'bool1'); // a boolean key
// remember the regular Object? it would convert keys to string
// Map keeps the type, so these two are different:
alert( map.get(1) ); // 'num1'
alert( map.get('1') ); // 'str1'
alert( map.size ); // 3
```

Chaining

```
map.set('1', 'str1').set(1, 'num1'). set(true,'bool1');
```

Map can also use objects as keys

```
let john = { name: "John" };
// for every user, let's store their visits count
let visitsCountMap = new Map(); // john is the key for the map
visitsCountMap.set(john, 123);
alert( visitsCountMap.get(john) ); // 123
```

Maps

When a Map is created, we can pass an array (or another iterable) with key-value pairs, like this

```
// array of [key, value] pairs
let map = new Map(
[
        ['1', 'str1'],
        [1, 'num1'],
        [true, 'bool1']
]
);
```

There is a built-in method Object.entries(obj) that returns an array of key/value pairs for an object exactly in that format.

So we can initialize a map from an object like this:

```
let map = new Map(Object.entries({
    name: "John",
    age: 30 })
);
```

Here, `Object.entries` returns the array of key/value pairs:
`[ ["name","John"], ["age", 30] ]`.
That's what `Map` needs.

Maps

Iteration over Map

For looping over a `map`, there are 3 methods:

- `map.keys()` – returns an iterable for keys,
- `map.values()` – returns an iterable for values,

// returns an iterable for entries `[key, value]`, it's used by default in `for..of`

- `map.entries()`

```
// runs the function for each (key, value) pair
recipeMap.forEach( (value, key, map) => {
    alert(`${key}: ${value}`); // cucumber: 500 etc
});
```

```
let recipeMap = new Map([
 ['cucumber', 500],
['tomatoes', 350],
['onion', 50]
]);

 // iterate over keys (vegetables)
for (let vegetable of recipeMap.keys()) {
        alert(vegetable); // cucumber, tomatoes, onion
}
// iterate over values (amounts)
for (let amount of recipeMap.values()) {
        alert(amount); // 500, 350, 50
}
// iterate over [key, value] entries
for (let entry of recipeMap) {
        // the same as of recipeMap.entries()
        alert(entry); // cucumber,500 (and so on)
}
```

Sets

A Set is a collection of values, where each value may occur only once.

Its main methods are:

- new Set(iterable) – creates the set, and if an iterable object is provided (usually an array), copies values from it into the set.
- set.add(value) – adds a value, returns the set itself.
- set.delete(value) – removes the value, returns true if value existed at the moment of the call, otherwise false.
- set.has(value) – returns true if the value exists in the set, otherwise false.
- set.clear() – removes everything from the set.
- set.size – is the elements count.

Sets

```
let set = new Set();
let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" }; // visits, some users come multiple times

set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary); // set keeps only unique values
alert( set.size ); // 3

for (let user of set) {
alert(user.name); // John (then Pete and Mary
)
```

```
let set = new Set(["oranges", "apples", "bananas"]);
for (let value of set)
    alert(value);

// the same with forEach:
set.forEach((value, valueAgain, set) => {
    alert(value);
});
```

WeakMap & WeakSet

WeakSet is a special kind of Set that does not prevent JavaScript from removing its items from memory. WeakMap is the same thing for Map.
As we know from the chapter Garbage collection, JavaScript engine stores a value in memory while it is reachable (and can potentially be used).

```
let john = { name: "John" };

let array = [ john ];

john = null; // overwrite the reference
```

Note: john is stored inside the array, so it won't be garbage-collected, we can get it as array[0]

If we use an object as the key in a regular Map, then while the Map exists, that object exists as well. It occupies memory and may not be garbage collected.

```
let john = { name: "John" };

let map = new Map();
map.set(john, "...");

john = null; // overwrite the reference
```

Note: john is stored inside the map, we can get it by using map.keys()

WeakMap & WeakSet

`Note: WeakMap` keys must be objects, not primitive values

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // overwrite the reference

// john is removed from memory!
```

Function Basics

Function Declaration

```
function add(num1, num2){
 return num1+num2;
}
```

Function Expression (Anonymous Function)

```
let addE = function(num1, num2){
                        return num1+num2;
                }
```

```
console.log(addE(1,2));
```

Handling arity of function (multiple arguments)

```
let addE = function(){
    let sum=0;
    for(let i=0; i<arguments.length; i++){
      sum += arguments[i];
    }
    return sum;
}
console.log(addE(1,2,3,4));
```

Function hoisting allowed in Function Declaration

```
console.log(addE(1,2));

function addE(num1 , num2){
  //console.log(arguments);
  return num1+num2;
}
```

Function hoisting not allowed in Function Expression

```
console.log(addE(1,2));

var addE = function(num1, num2){
                    return num1+num2;
                }
```

Spread Operator and Rest parameters

Handling arity of function (multiple arguments, handling with spread operator …)

Since ES6, we can use the *spread syntax* when calling the function:

```
func(...args);
```

Note: args is called rest parameter

Since ES6 also if you expect to treat your arguments as an array, you can also use the spread syntax in the parameter list, for example:

```
function func(...args) {
        args.forEach(arg => console.log(arg))
        }

const values = ['a', 'b', 'c']
```

```
let addE = function(...args){
  let sum=0;
  for(let i=0;i<args.length;i++){
    sum += args[i];
  }
  return sum;
}
console.log(addE(1,2,3,4));
```

```
let obj1 = {name: "Srini", age: 45};
let obj2 = {age: 50};

let combinedObj = {...obj1, ...obj2};
console.log( combinedObj) ;
```

We can combine it with normal parameters, for example if you want to receive the first two arguments separately and the rest as an array:

```
function func(first, second, ...theRest) {    //theRest has to be last argument
        //...
 }
```

Default Parameters

Prior to ES6

```
let addE= function(a,b){
  //If parameter, a has a value, keep it else assign 0
  a = a || 0;
  b = b || 0;
  return a+b;
}
console.log(addE(1,2));
console.log(addE());
```

ES6

```
let addE= function(a = 0,b = 0){
  return a+b;
}
console.log(addE(1,2));
console.log(addE());
```

```
const add = function(a =[], b = []){
  return [...a, ...b];
}

let total = add([1,2,3], [4,5]);
console.log(total);

let total = add(undefined, [4,5]);
console.log(total);
```

Callback Functions

In JavaScript, functions are objects. Because of this, functions can take functions as arguments, and can be returned by other functions.
Functions that do this are called **higher-order functions**.
Any function that is passed as an argument is called a **callback function**.

```
const host = function(cb){
  cb();
}

const callback = function(){
  console.log('you called me back from host')
}

host(callback);
```

```
function iExecuteYourCallback(callback) {
    // callback contains the anonymous function passed to it
    // Similar to doing:
    // var callback = function () { };
    callback();
}
iExecuteYourCallback(function() {
    console.log('Im a callback function!');
});
```

```
const calc = function(calcFunction,num1,num2){
              return calcFunction(num1,num2);
             }

let result = calc(function(a,b){
                  return a**b;
                },2,3);


console.log(result);
```

Why do we need Callbacks?

Why do we need Callbacks?

For one very important reason — JavaScript is an event driven language. This means that instead of waiting for a response before moving on, JavaScript will keep executing while listening for other events. Callbacks are widely used in map, filter and reduce methods which are applied on arrays.

```
function first(){
console.log(1);
}

function second(){
console.log(2);
}

first();

second();
```

Callbacks are a way to make sure certain code doesn't execute until other code has already finished execution.

But what if function first contains some sort of code that can't be executed immediately? For example, an API request where we have to send the request then wait for a response?

```
function first(){
  // Simulate a code delay
   setTimeout( function(){
    console.log(1);
     }, 500 );
}

function second(){
  console.log(2);
}

first();
second();
```

JavaScript didn't wait for a response from first() before moving on to execute second()

Callback Examples

```
//Part1: Operate on 1 to n numbers based on the callback function

const add = function(a,b){
  return a+b;
}

const calc = function(n=1,cb){
  if(n<1){
    return 0;
  }
  if(n===1){
    return 1;
  }
  let total =1;
  for(let i=2;i<=n;i++){
    total = cb(total,i);
  }
  return total;
}
console.log(calc(3,add));
```

```
//Another Example

const mult = function(a,b){
  return a*b;
}

const calc = function(cb, arr=[]){
  if(arr.length === 0){
    return 0;
  }
  let total =arr[0];
  for(let i=1;i<arr.length;i++){
    total = cb(total,arr[i]);
  }
  console.log(total);
  return total;
}

const a =[100,200];
console.log(calc(mult,a));
```

Function that returns function

```
function a() {

    alert('A!');

    function b(){
        alert('B!');
    }

    return b();
}

var s = a();
alert('break');
s();
```

```
function d() {
    function e() {
        alert('E');
    }
    return e;
}


d()();

// same as

let a = d();
a();
```

The function count can keep the variables that were defined outside of it. This is called a closure

```
function counter() {
    var count = 0;
    return function() {
        console.log(count++);
    }
}
var count = counter();
count();
count();
count();
```

Map, Reduce and Filter methods

Map

Often, we find ourselves needing to take an array and modify every element in it in exactly the same way. Typical examples of this are squaring every element in an array of numbers, retrieving the name from a list of users, or running a regex against an array of strings.

map is a method built to do exactly that. It's defined on Array.prototype, so you can call it on any array, and it accepts a callback as its first argument.

When you call map on an array, it executes that callback on every element within it, returning a *new* array with all of the values that the callback returned.

Under the hood, map passes three arguments to your callback:
1.The *current item* in the array
2.The *array index* of the current item
3.The *entire array* you called map on

Map method

```
var tasks = [
 {
   'name'     : 'Write for Envato Tuts+',
   'duration' : 120
 },
 {
   'name'     : 'Work out',
   'duration' : 60
 },
 {
   'name'     : 'Procrastinate on Duolingo',
   'duration' : 240
 }
];
```

Using for loop

```
var task_names = [];
for (var i = 0, max = tasks.length; i < max; i += 1) {
    task_names.push(tasks[i].name);
}
```

Using forEach

```
var task_names = [];
tasks.forEach(function (task) {
    task_names.push(task.name);
});
```

```
var task_names = [];
var task_names = tasks.map(function (task) {
    return task.name;
});

console.log(task_names);
```

Using map

```
var task_names = [];
var task_names = tasks.map(function (task, index, array) {
    return task.name;
});
```

Using ES6 arrow function

```
var task_names = tasks.map((task) => task.name );
```

Filter method

It takes an array, and filters out unwanted elements.

Like map, filter is defined on Array.prototype. It's available on any array, and you pass it a callback as its first argument. filter executes that callback on each element of the array, and spits out a *new* array containing *only* the elements for which the callback returned true.

Also like map, filter passes your callback three arguments:
1.The *current item*
2.The *current index*
3.The *array* you called filter on

Filter method

Using forEach

```
var difficult_tasks = [];

tasks.forEach(function (task) {
    if (task.duration >= 120) {
        difficult_tasks.push(task);
    }
});
```

```
const numArray = [{num:1},{num:2},{num:3}];

const cb=function(e){
  if(!(e.num%2 === 0)){
    return e.num*2;
  }
}
const newArray = numArray.filter(cb);
console.log(newArray);
```

With filter

```
var difficult_tasks = tasks.filter(function (task) {
    return task.duration >= 120;
});
```

Using ES6

```
var difficult_tasks = tasks.filter((task) => task.duration >= 120 );
```

Reducing Arrays

map creates a new array by transforming every element in an array, individually. filter creates a new array by removing elements that don't belong. reduce, on the other hand, takes all of the elements in an array, and *reduces* them into a single value.

Just like map and filter, reduce is defined on Array.prototype and so available on any array, and you pass a callback as its first argument. But it also takes an optional second argument: the value to start combining all your array elements into.

reduce passes your callback four arguments:

1.The *current value*
2.The *previous value*
3.The *current index*
4.The *array* you called reduce on

Reducing Arrays

```
var numbers = [1, 2, 3, 4, 5],
    total = 0;

numbers.forEach(function (number) {
    total += number;
});
```

```
let total_time1 = tasks.map((task) => task.duration )
  .reduce(function (accumulator, current) {
    return accumulator + current;
}, 0);

console.log(total_time1);
```

With reduce

Provide 0 as second argument to reduce method so that it will have the initial value for ***accumulator***

```
var total = [1, 2, 3, 4, 5].reduce(function (previous, current) {
    return previous + current;
}, 0);
```

```
const numArray = [{num:1},{num:2},{num:3}];

const cb = function(acc,e){
    return acc + e.num;
}
console.log(numArray.reduce(cb,0);
```

```
// Using arrow functions
let total_time2 = tasks.map((task) => task.duration )
  .reduce((previous, current)=> previous + current,0 );

console.log(total_time2);
```

Putting It Together: Map, Filter, Reduce, and Chainability

1. Collect two days' worth of tasks.
2. Convert the task durations to hours, instead of minutes.
3. Filter out everything that took two hours or more.
4. Sum it all up.
5. Multiply the result by a per-hour rate for billing.
6. Output a formatted dollar amount.

```
var result = tasks.reduce(function (accumulator, current) {
                return accumulator.concat(current);
            }).map(function (task) {
                return (task.duration / 60);
            }).filter(function (duration) {
                return duration >= 2;
            }).map(function (duration) {
                return duration * 25;
            }).reduce(function (accumulator, current) {
                return [(+accumulator) + (+current)];
            }).map(function (dollar_amount) {
                return '$' + dollar_amount.toFixed(2);
            }).reduce(function (formatted_dollar_amount) {
                return formatted_dollar_amount;
            });
```

```
var monday = [
        {
            'name'     : 'Write a tutorial',
            'duration' : 180
        },
        {
            'name'     : 'Some web development',
            'duration' : 120
        }
    ];

var tuesday = [
        {
            'name'     : 'Keep writing that tutorial',
            'duration' : 240
        },
        {
            'name'     : 'Some more web development',
            'duration' : 180
        },
        {
            'name'     : 'A whole lot of nothing',
            'duration'  : 240
        }
    ];

var tasks = [monday, tuesday];
```

Sorting Arrays using callback functions

```
const arr = [1,9,7,2,11];
//sort() treats numbers as strings
arr.sort();
console.log(arr);;
```

```
const arr = [1,9,7,2,11];

//callback, compares 2 numbers
const cb = function(a,b){
  return a-b;
}
//sorting in ascending order
arr.sort(cb);
console.log(arr);

//callback, compares 2 numbers
//sorting in descending order
const cb = function(a,b){
  return b-a;
}
```

```
const arr = [{name:"john",age:30},
             {name:"peter",age:20},
             {name:"jill",age:90}
            ];

const cb = function(a,b){
  return a.name>b.name?-1:1;
 // return a.name>b.name?-1:1;
}
arr.sort(cb);
console.log(arr);
```

Array Methods: some() and every()

```
const arr = [1,9,7,2];
const arr1 = [1,9,7];

const cb = function(element){
  return element === 2;
}

let hasSomeTwo= arr.some(cb);
console.log(hasSomeTwo);
console.log(arr1.some(cb));
```

```
const arr = [{name:"john",age:30},
             {name:"peter",age:20},
             {name:"jill",age:90}
            ];

const cb = function(element){
  return element.age === 20;
}

console.log(arr.some(cb));

const cb1 = function(element){
  return element.age > 25;
}

console.log(arr.every(cb1));
```

Functional Programming

About this

JavaScript has a number of built in function/methods that can apply to various data types. Just as .split(“”) can apply to a string, .call(), .apply() and .bind() are methods of functions.

These 3 functions are all designed to set CONTEXT, or more specifically, what “this” refers to.

```
const profile = {
  name: "john",
  children: [
    {
      name: "peter",
      getName() {
        return this.name;
      }
    },
    { name: "jill" }
  ],
  getName() {
    return this.name;
  }
};

console.log(profile.getName());
console.log(profile.children[0].getName());
const getNameLoose = profile.getName;
console.log(getNameLoose());
```

Functional Programming

Call, Apply and Bind Methods

```
const profile = {
  name: "john",
  age:20
};

const setName = function(name) {
  this.name = name;
};

//ad-hoc calling
setName.call(profile, 'peter');

console.log(profile);
```

```
const profile = {
  name: "john",
  age:20
};

const setProfile = function(name, age) {
  this.name = name;
  this.age= age;
};

//ad-hoc calling
setProfile.call(profile, 'peter', 30);

console.log(profile);
```

.call essentially ***forces*** the value in ***this*** in whichever function .call is applied on.
In this case, we invoke the sayProfile function.
.call(profile, …) forces the ***this*** of sayProfile() to be the profile object.

```
const profile = {
  name: "john",
  age:20
};

const setProfile = function(name, age) {
  this.name = name;
  this.age= age;
};

let params = ['Peter', 30];
setProfile.apply(profile, params);

console.log(profile);
```

Using .apply() is very similar to .call. The only difference is that .call() takes arg1 and arg2 just one after another.
For .apply() to work, arguments 1 and 2 need to be entered as an array

Functional Programming

Call, Apply and Bind

```
const profile = {
  name: "john",
  age:20,
 //loosely binded, not to used
  setProfile(){

 }

};
```

Using .bind()

.bind(), unlike .apply() and .call(), returns a ***function*** instead of a value.
.bind() sets the value of ***this*** and changes the function to a new function, but it doesn't invoke the function.

```
const profile = {
  name: "john",
  age:20
};

const setProfile = function(name, age) {
  this.name = name;
  this.age= age;
};

let params = ['Peter', 30];
const boundProfile = setProfile.bind(profile);
//console.dir(boundProfile);
boundProfile('Romeo', 21);
console.log(profile);
```

Closures

If you declare a function within another function, you create what's called a **scope chain**. The outer parent cannot see the variables of its children, but the child can see the variables of its parent(s).

```
//this is closure's global scope

 function outie(){
 // this is closure's first and only outer scope

 function closure(){
   // this is closure's local scope
 }

}
```

Closures

Closures

Example 1

```
const counter = () => {
  let i=0;

  const add = (val) =>{
    i += val
  };

   const subtract = (val) =>{
    i -= val
  }

  const print = () =>{
    return i;
  }

  return {
    add,
    subtract,
    print
  }
}
```

Example 2

```
function sayHello() {
  const say = () => { console.log(hello); }
  // Local variable that ends up within the closure
  let hello = 'Hello, world!';
  return say;
}
const sayHelloClosure = sayHello();
sayHelloClosure(); // 'Hello, world!'
```

Notice how the variable **hello** is defined *after* the anonymous function—but can still access the **hello** variable. This is because the **hello** variable has already been defined in the function sayHello(), making it available when the anonymous function is finally executed.

```
const outerCounter =counter();
console.log(outerCounter);
outerCounter.add(2);
console.log(outerCounter.print());
outerCounter.subtract(1);
console.log(outerCounter.print());
```

Closures

Example 3

```
var x = 10;
function foo(a) {
  var b = 20;

  function bar(c) {
    var d = 30;
    return boop(x + a + b + c + d);
  }

  function boop(e) {
    return e * -1;
  }

  return bar;
}

var moar = foo(5); // Closure
console.log(moar(15));
```

Closures : Function Chaining

```
const counter = () => {
  let i = 0;

  return {
    add(val) {
      i += val;
      return this;
    },
    subtract(val) {
      i -= val;
      return this;
    },
    print() {
      return i;
    }
  };
};
```

```
const counterObj = counter();

console.log(
  counterObj
    .add(1)
     .add(5)
    .subtract(4)
    .print()
);
```

Javascript Blocking and Non-Blocking IO

Blocking methods:

When a thread invokes a read() or write() operation, that thread is blocked until there is some data to read, or the data is fully written. The thread can do nothing else in the meantime. Some operations, e.g. reading a file, used to be single threaded operations, and the operation had to be completed before another could be executed. This used to cause bottlenecks, or, like explained in the opening sentence, blocks or deadlocks. This was just how things used to work. There was nothing else until about 2002 for server-side languages.

Non-Blocking Methods:

It is 2009. A miracle occurs: Node.js changes the way front-ends are developed forever. The initial release is supported only by Linux and Mac OSX, and finally, in June 2011, Microsoft implement a native Windows version.

Non-Blocking I/O enables a thread to request reading data from a channel, or in the case of front-end technologies from "the Event Loop," gets what is currently available, or nothing at all if no data is available. Rather than remain blocked until data becomes available for reading, the thread can go on with something else. Revolutionary: speeding up processing time and read/write time quite dramatically, with improved error handling:

```
// reading a file asynchronously
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
if (err) throw err; });

// reading a file synchronously
const fs = require('fs');
const data = fs.readFileSync('/file.md');
// blocks here until file is read
```

So what do threads spend their idle time doing when not blocked in IO calls? Usually performing IO on other channels or on other requests in the event loop.

That is, a single thread can now manage multiple requests of input and output. Blocking methods execute synchronously (at once — the full process from start to finish) and non-blocking methods execute asynchronously — sending and receiving "call-backs" as tokens to and from where their operations are performed.

Java vs Node

In order to understand how Node beats Java, you have to consider 3 factors: IO, Concurrency, and Computation. All 3 factors contribute the overall speed and throughput of an application.

1. IO

While Java introduced nio several years ago, take a look at the servlet spec, Java's de facto standard for writing backend server code. All throughout the spec, the API's make the assumption that code can and will block at every level. While improvements to the spec over the last few years now allow non-blocking request handling, the fundamental paradigm for the servlet spec still fully embraces IO blocking. That means that operations that write to disk in a non-blocking way, or write to a database using JDBC, block the processing thread from doing anything else. While java is capable of working in a non-blocking way, it remains impractical for Java applications that are based on servlet technology to successfully implement a non-blocking server. This is a major misfortune for the Java community.

JavaScript, on the other hand, fully embraces non-blocking IO through and through. Its API's don't provide an option for blocking, which forces non-blocking API specs at every level. That means that Node will never be blocked while IO operations are being performed, and it will always be free to process other work while waiting for IO. This allows Node to use the server's processing and memory resources incredibly efficiently. It can process and issue thousands of IO operations at once with only a single thread.

2. Concurrency

Java and Node achieve concurrency in the web frameworks in two very different ways.

Java dedicates a thread per each incoming request that is processed. As the number of concurrent requests grows, the number of threads also has to grow. If you plan on processing up to 100 simultaneous requests, then you will need to a pool of 100 threads available for processing.

Node, on the other hand, uses a single thread to process all incoming requests. Because Node is non-blocking, it has plenty of time to do all the computations and transformations for all the requests while it is waiting for any IO operations.

3. Computation

While Google's V8 JavaScript engine compiles JavaScript into machine code to achieve impressive results, Java still has superior performance over Node. If you were to pit Java against Node in a contest of pure computation doing something such as computing the first 5000 prime numbers, Java would win quite easily every time.

Web Applications are IO Heavy

Web Applications do a lot of IO. First, a request is received from the browser; that's IO. Second, the application typically fetches the requested data from a database; that's IO. Once it has all the response data computed, it then sends it back to the browser; that's IO. All the while, the application is probably maintaining an application log, and that's IO too. Web applications, in fact, spend most of their time doing IO, not computation.

If servicing each request were a race between the two technologies, every time more IO is performed, Node would win a little bit more. What this amounts to is that Node is incredibly well suited for web applications.

Web Applications Require High Concurrency

How many web applications can you think of that are only accessed by one person at a time? Since Java must dedicate a thread for the life of each request, it demands a lot of threads to achieve concurrency. This doesn't seem like a big deal, and in fact it isn't if you only have a few users, but if your user load increases, time slicing between all those threads quickly becomes very costly and inefficient. Eventually as the number of threads gets too high, the operating system becomes so busy time slicing between all the threads that it doesn't have any CPU time left to do any actual work.

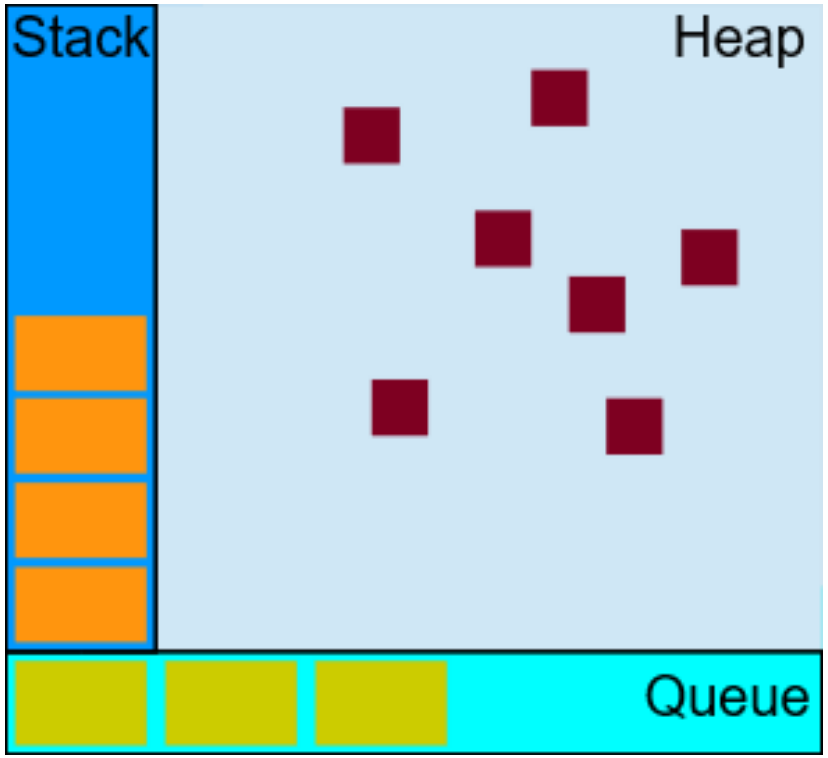
The efficiency of Node becomes really apparent at scale. When Node is running at max load, although the CPU will be very busy, the operating system won't even break a sweat. All of the CPU time will be spent doing exactly what you want, servicing requests.

Concurrency Model and Event Loop

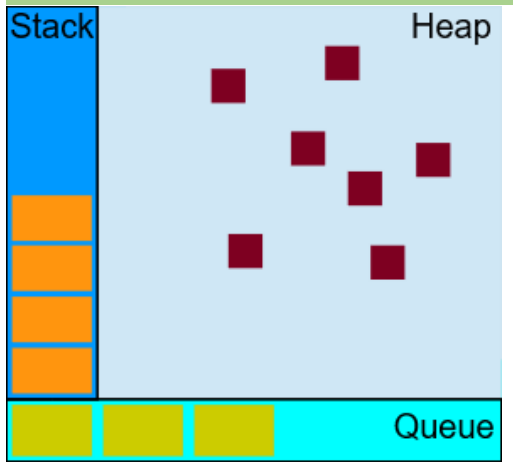
JavaScript has a concurrency model based on an "event loop". This model is quite different from models in other languages like C and Java.

Remember that JavaScript is single-threaded programming language. Hence concurrency cannot be achieved through multithreading like in Java, C++ etc.

Visual representation



Concurrency Model and Event Loop



Stack

Function calls form a stack of *frames*

```
function f(b) {
        var a = 12;
        return a + b + 35;
}

function g(x) {
        var m = 4;
        return f(m * x);
}

g(21);
```

When calling g, a first frame is created containing g arguments and local variables.

When g calls f, a second frame is created and pushed on top of the first one containing f arguments and local variables.

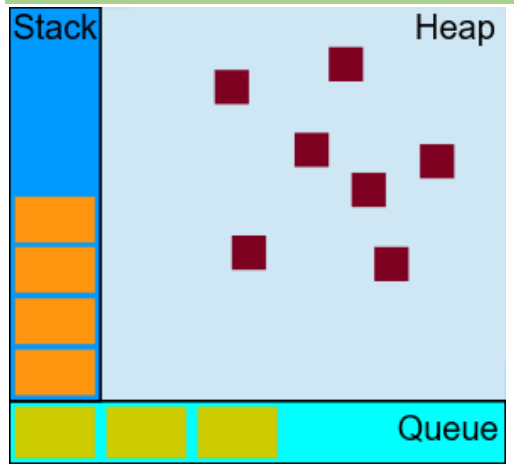
When f returns, the top frame element is popped out of the stack (leaving only g call frame).

When g returns, the stack is empty.

Heap

Objects are allocated in a heap which is just a name to denote a large mostly unstructured region of memory

Concurrency Model and Event Loop



Queue

A JavaScript runtime contains a message queue, which is a list of messages to be processed.

A function is associated with each message. When the stack has enough capacity, a message is taken out of the queue and processed.

The processing consists of calling the associated function (and thus creating an initial stack frame).

The message processing ends when the stack becomes empty again.

Concurrency Model and Event Loop

Event Loop

The **Event Loop** is a queue of callback functions. When an async function executes, the callback function is pushed into the queue.

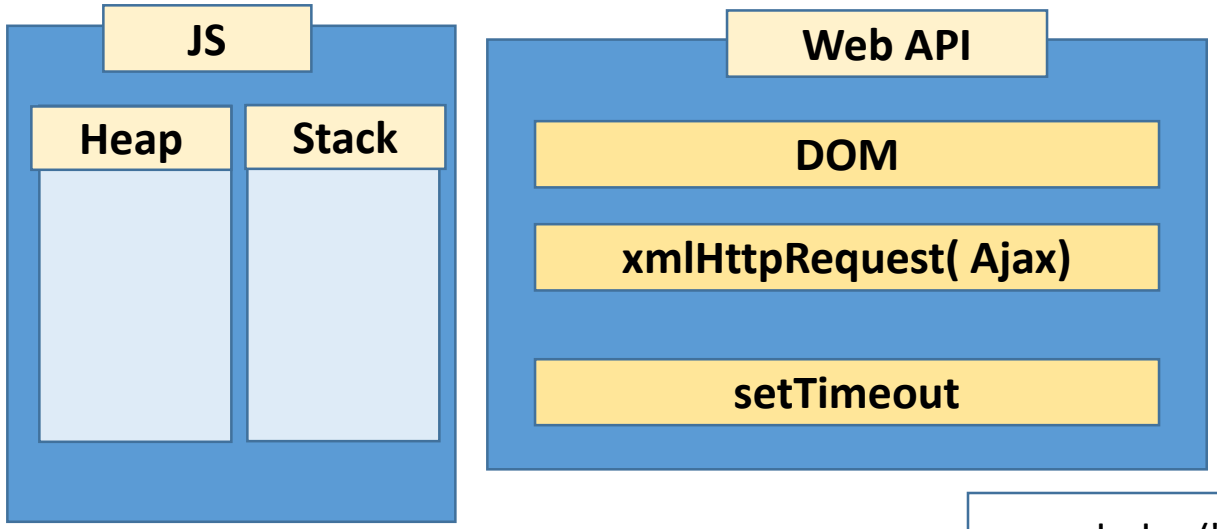
The **JavaScript** engine doesn't start processing the **event loop** until the code after an async function has executed i.e. the current stack frame is cleared.

The event loop got its name because of how it's usually implemented, which usually resembles:

```
while(queue.waitForMessage()) {
      queue.processNextMessage();
}
```

queue.waitForMessage waits synchronously for a message to arrive if there is none currently.

Concurrency Model and Event Loop




```
console.log("Hi");
setTimeout(function cb(){
        console.log("Welcome");
},0);
console.log("Hello");
```

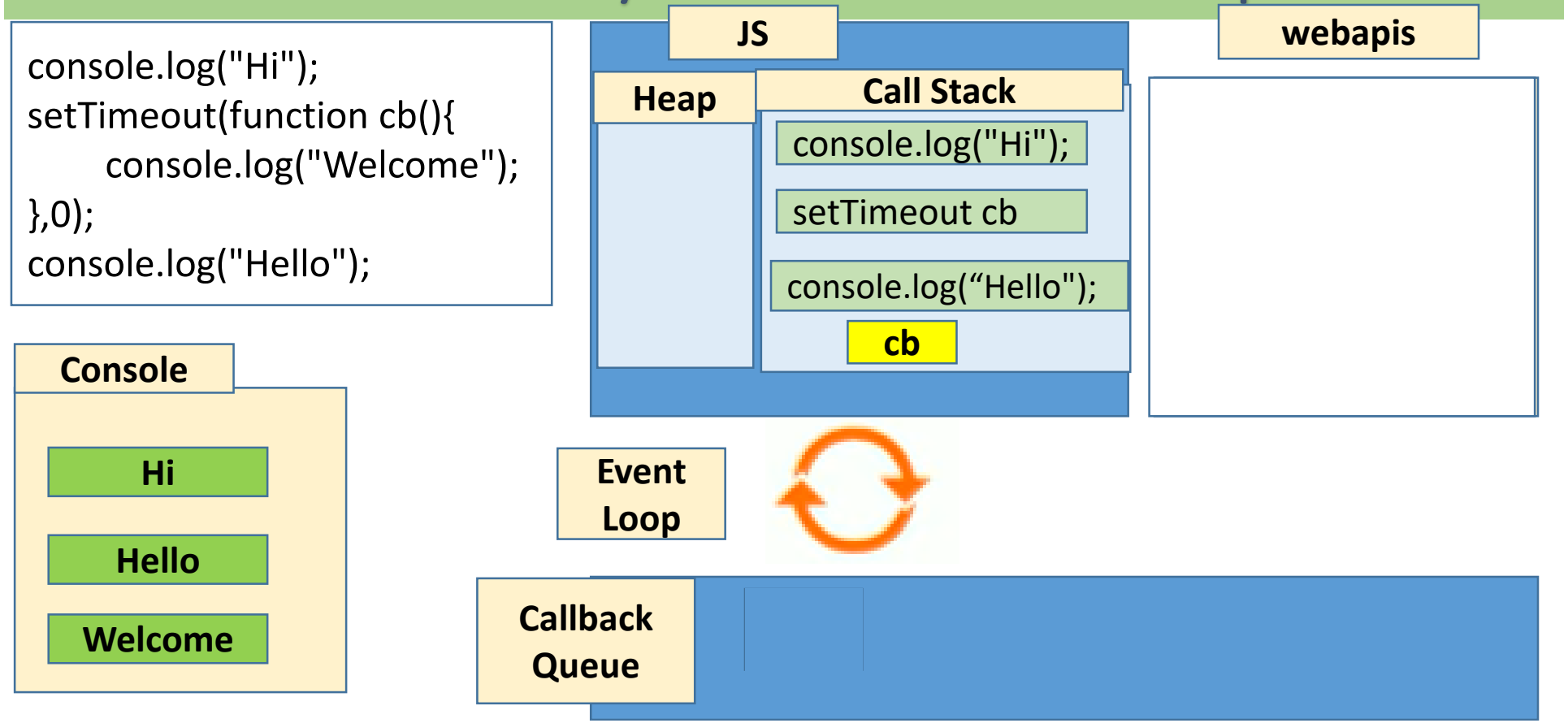
Event Loop

Callback Queue

Note: setTimeout() is an asynchronous function.

Concurrency Model and Event Loop

```
console.log("Hi");
setTimeout(function cb(){
      console.log("Welcome");
},0);
console.log("Hello");
```



When setTimeout() is called, we are passing callback function and a delay to the setTimeout call. setTimeout is an API provided to us by the browser, it doesn’t live in the V8 source, It is the extra stuff we get when we are running Javascript. The browser kicks of the timer indicating that the callback function will be executed with a minimum delay time of 5 seconds. As the timer is ticking, console.log("Hello") gets executed. Once the setTimeout() times out, web API pushes the callback function to the callback queue. The event loop job is to look at the stack and if it is empty, it takes the first message(cb) from the queue and pushes it on to the stack which effectively runs it.

setTimeout() : Zero delay

Zero delays

Zero delay doesn't actually mean the call back will fire-off after zero milliseconds.

Calling setTimeout with a delay of 0 (zero) milliseconds doesn't execute the callback function after the given interval. The execution depends on the number of awaiting tasks in the queue

```
(function () {
    console.log('this is the start');
    setTimeout(function cb() {
        console.log('this is a message from call back1');
    },1);
    console.log('this is just a message');
    setTimeout(function cb1() {
        console.log('this is a message from call back2');
    }, 0);
    console.log('this is the end');
})();
```

JavaScript Event Loop

```
console.log("A");

setTimeout(
  ()=>{
   console.log("B");
   setTimeout(()=>{
     console.log("C")
   },50);
  },100);

setTimeout(()=>{
  console.log("D");
},200);
console.log("E");
```

```
const main = ()=>{
  console.log('A');
  setTimeout(
    function display(){ console.log('B'); }
  ,0);
console.log('C');
}
main();
```

```
(function () {
console.log('this is the start');
setTimeout(function cb1() {
console.log('this is a message from call back1');
},200);
console.log('this is just a message');
setTimeout(function cb2() {
console.log('this is a message from call back2');
}, 100);
console.log('this is the end');
})();
```

setInterval() & setTimeout() methods

```
let id1 = setInterval(
  ()=>{
   //onsole.log('id1') ;
  },1000);

let id2 = setInterval(
  ()=>{
    //console.log('id2') ;
  },1000);

let id3 = setInterval(
  ()=>{
   //onsole.log('id3');
  },1000);

console.log(id1,id2,id3);
setTimeout(()=>{},6001);
```

```
let id1 = setInterval(
  ()=>{
    console.log('id1') ;
  },1000);

//console.log(id1,id2,id3);
console.log(id1);
setTimeout(()=>{ clearInterval(id1);},2001);
```

Promises

The **Promise** object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

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.

A Promise is in one of these states:

- *pending*: initial state, neither fulfilled nor rejected.
- *fulfilled*: meaning that the operation completed successfully.
- *rejected*: meaning that the operation failed.

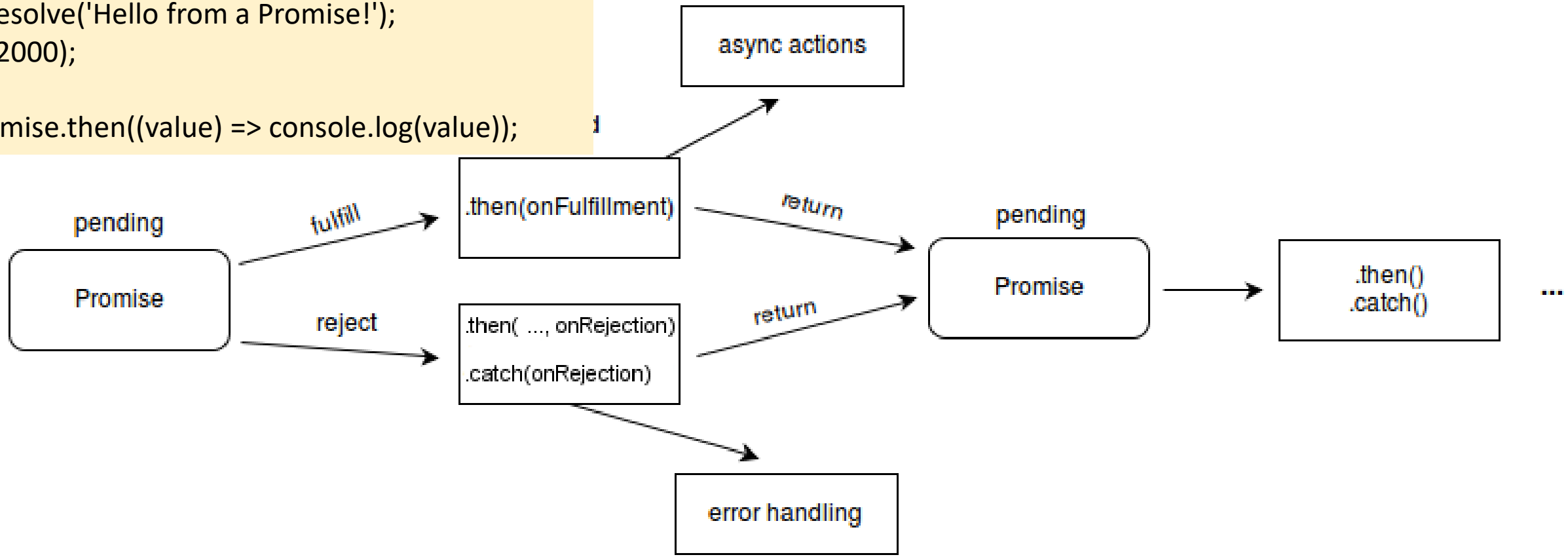
A pending promise can either be *fulfilled* with a value, or *rejected* with a reason (error).

When either of these options happens, the associated handlers queued up by a promise's **then method** are called. (If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.)

Promises

Promise.prototype.then() and Promise.prototype.catch() methods return promises, so they can be chained

```
const promise = new Promise((resolve,reject) =>
{
  setTimeout(() => {
    resolve('Hello from a Promise!');
  }, 2000);
});
promise.then((value) => console.log(value));
```



Promises

```
const willCleanTheRoom = new Promise( (resolve, reject)=>{
    //do what you told
  setTimeout(()=>{
    let cleanRoom = false;
    if(cleanRoom){
      resolve('room is clean');
    }else{
      reject('room is still dirty');
    }
  },3000);

});

willCleanTheRoom.then((resolveStatus)=>{
  console.log(resolveStatus );
}).catch((rejectStatus)=>{
  console.log(rejectStatus );
}).finally(()=>{
  console.log('I will be called regardless');
})
```

Chaining Promises

```
const willCleanTheRoom = new Promise( (resolve, reject)=>{
   //do what you told
  setTimeout(()=>{
    let cleanRoom = true;
    if(cleanRoom){
      resolve('room is clean');
    }else{
      reject('room is still dirty');
    }
  },3000);

});

const removeGarbage = new Promise(
  (resolve,reject)=>{
    let removeGarbage=true;
    if(removeGarbage){
      resolve('garbage removed');
    }else{
      reject('garbage is not removed');
    }
  }
```

```
const getReward = new Promise(
 (resolve,reject)=>{
   let getReward=true;
   if(getReward){
     resolve('received reward');
   }else{
     reject('No rewards :()');
   }
 }
);

willCleanTheRoom.then((resolveStatus)=>{
  console.log(resolveStatus);
  return removeGarbage;
}).then(resolveStatus =>{
  console.log(resolveStatus);
  return getReward;
}).then(resolveStatus=>{
  console.log(resolveStatus);
  console.log('all done');
}).catch(rejectStatus=>{
```

Executing Promises at a time

```
const willCleanTheRoom = new Promise( (resolve, reject)=>{
   //do what you told
  setTimeout(()=>{
    let cleanRoom = true;
    if(cleanRoom){
      resolve('room is clean');
    }else{
      reject('room is still dirty');
    }
  },3000);
});

const removeGarbage = new Promise(
  (resolve,reject)=>{
    let removeGarbage=true;
    if(removeGarbage){
      resolve('garbage removed');
    }else{
      reject('garbage is not removed');
    }
  }
);
```

```
const getReward = new Promise(
 (resolve,reject)=>{
   let getReward=true;
   if(getReward){
     resolve('received reward');
   }else{
     reject('No rewards :()');
   }
 }
);

Promise.all([willCleanTheRoom,removeGarbage,getReward])
  .then((messages)=>{
    console.log(messages);
})
```

```
Promise.race([willCleanTheRoom,removeGarbage,getReward])
  .then((messages)=>{
    console.log(messages);
})
```

Thank You!