JavaScript

André Restivo

Index

Introduction

JavaScript

- *JavaScript* is a **dynamic**, **imperative** and **functional** (ish) language.
- In *JavaScript*, functions are considered **first-class** citizens.
- It is also **object-oriented**, but **prototype-based** (not class-based).
- Most commonly used as a **client-side** scripting language (in browsers).
- But can also be used as a **general purpose** language.

History

- Originally developed by **Brendan Eich** at **Netscape**.
- Developed under the name **Mocha** but later named **LiveScript**.
- Changed name from LiveScript to **JavaScript**, in **1995**, when Netscape added support for Java.
- Microsoft introduced JavaScript support in Internet Explorer in August **1996** (called JScript).
- Submitted to **Ecma** International for consideration as an industry-standard in 1996 (**ECMAScript**).
- Ecma International released the first version of the specification in **1997**.
- Nowadays, JavaScript is a trademark of the **Oracle** Corporation.
- But JavaScript is officially managed by the **Mozilla** Foundation.

Console

- Modern browsers all have a *JavaScript* console that can log messages from within web pages.
- It can also inspect variables, evaluate expressions, and just plain experimentation.
- The specifics of how it works vary from browser to browser, but there is a *de facto* set of typically provided features.
- The **console.log(msg)** function outputs a message to the console:

```
console.log('Hello World')
```

- Other debug level are possible:
  - **console.info(msg)**, **console.warn(msg)** and **console.error(msg)**.
  - Browsers allow filtering messages depending on their level.

Strict Mode

ECMAScript 5 brought some significant changes.

To opt-in for those changes, scripts (or functions) must start with:

```
'use strict'
```

Some of those changes:

- **No more** global undeclared variables.
- **No more** declaring variables with **var**.
- Some warnings are now errors.

Automatic Semicolons

Statements are separated by semicolons:

```
console.log(123); console.log('abc');
```

But if a line break separates them:

```
console.log(123);
console.log('abc');
```

The semicolon can be omitted:

```
console.log(123)
console.log('abc')
```

This is not always true!

Resources

- Reference:
  - MDN JavaScript Reference
  - ECMAScript Reference
  - MDN DOM Reference
- Resources:
  - MDN JavaScript Resources
  - JS Fiddle
- Tutorials:
  - The Modern JavaScript Tutorial
  - JavaScript Style Guide

Variables

Variables

- JavaScript is a *loosely/weakly* and *dynamically* typed language.
- That means that:
  - values have types; variables do not.
  - types are checked at runtime.
- Variables are declared using the **let** command.
- Variable names must contain only letters, digits, $, and _ (and not start with a digit).

```
let bar = 10         // bar initialized with a numb
bar = 'John Doe'     // bar now has a string
bar = true           // and now a boolean
```

```
let foo = 10, bar    // declaring two variables at
bar = 'John Doe'     // bar was undefined
```

Constants

- Constants behave precisely the same way as variables.
- Except they can't be changed.
- Constants are declared using the **const** command.

```
const bar = 10
bar = 20          // TypeError: invalid assignment
```

Always prefer **const**; only use **let** if you need to reassign the variable.

Var

In older scripts, you might find variables declared using **var** instead of **let**.

- They have no block scope (only function scope).
- Are processed when a function starts.
- **And should not be used!**

```
if (true) {
  var bar = '1234'
  console.log(bar)      // 1234
}

console.log(bar)        // 1234
```

```
function foo() {
  bar = '1234'
  console.log(bar)     //1234
  var bar
}
```

Not declaring variables

- Declaring variables in *JavaScript* might seem *optional*, but that is **not the case**.
- When you use a variable without declaring it, that variable will bubble up until it finds a variable declared with the same name.
- If it doesn't, it attaches itself to the *window* or *global* object.
- This might have unforeseen and complex to debug consequences:

```
function foo() {
  bar = 1234
}

let bar = 10
foo()
console.log(bar) // 1234
```

Primitive Data Types

The standard defines the following data types:

- Number (**double**-precision 64-bit, *e.g.*, 10)
- String (**text**ual data — single or double quoted, *e.g.*, 'foo')
- Boolean (**true** or **false**)
- BigInt (**numbers** of arbitrary length, *e.g.*, 123456789n)
- Null (only one possible value: case sensitive **null**)
- Undefined (has **not** been **assigned** a value)

Strings

Strings can be defined equally using single or double quotes:

```
const firstname = 'John'
const lastname = "Doe"
```

We can also use *backticks*. With *backticks*, expressions inside *${...}* are evaluated, and the result becomes a part of the string.

```
console.log(`Hello, ${firstname} ${lastname}!`)
// Hello, John Doe!

console.log(`The result is ${1 + 2}`)
// The result is 3
```

The + Operator

The plus (+) operator sums numbers, but if one of the operands is a string, it converts the other one into a string and concatenates the two:

```
console.log(11 + 31)   // 42
console.log('11' + 31) // '1131'
console.log(11 + '31') // '1131'
```

Type Conversions

Most of the time, operators and functions automatically convert a value to the right type (type conversion).

You can still use the *String*, *Number* and *Boolean* functions to manually convert a value:

```
const a = 0
const b = Boolean(a) // false
const c = String(a)  // '0'
const d = String(b)  // 'false'
```

To convert from a string to a number, we can use the **parseInt** and **parseFloat** functions. Don't forget to specify the base:

```
  console.log(parseFloat('123.4')) // 123.4
  console.log(parseInt('123', 10)) // 123
  console.log(parseInt('123', 8))  // 83
  console.log(parseInt('0123'))    // 123 or 83 i
```

Comparison

When comparing values belonging to different types, they are converted to numbers:

Examples:

```
1 == '1'     // 1 == 1 -> true
0 == false   // 0 == 0 -> true
'0' == true // 0 == 1 -> false
'' == false // 0 == 0 -> true
Boolean('0') == false // 1 == 0 -> false
Boolean('0') == true  // 1 == 1 -> true
```

Primitives are compared by their value; objects (*e.g.*, arrays) are compared by their reference. This means [1, 2, 3] != [1, 2, 3]

Boolean Evaluation

The following values all evaluate to **false**:

- false
- undefined
- null
- 0
- NaN (not a number)
- the empty string

All other values, including objects, evaluate to **true**.

Be careful with the Boolean object:

```
const foo = new Boolean(false)
const bar = Boolean(false)
if (foo) // evaluates to true
if (bar) // evaluates to false
```

Strict Equality

- Strict equality compares two values for equality.
- Neither value is implicitly converted to some other value before being compared.
- If the values have different types, the values are considered unequal.

```
0 === 0     // true
0 === '0'   // false
0 === false // false
```

Comparing anything with **null** and **undefined** returns false. Comparisons between them have the following results:

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

Type Of

We can use the **typeof** function to check the type of a variable:

```
console.log(typeof undefined)         // "undefin
console.log(typeof 0)                 // "number"
console.log(typeof 10n)               // "bigint"
console.log(typeof true)              // "boolean
console.log(typeof 'foo')             // "string"
console.log(typeof new Boolean(true)) // "object"
console.log(typeof Boolean(true))     // "boolean
console.log(typeof null)              // "object"
console.log(typeof console.log)       // "functio
```

Nullish Coalescing

A common way to assign a **default value** is to use the **or** operator (||):

```
const bar = foo || some_default_value
```

This works, but it assigns the default value for any **falsy** value.

The nullish coalescing operator (??) returns the second argument if the first is *undefined* or *null*.

```
const bar = foo ?? some_default_value
```

Control Structures

If … else

- Use the **if** statement to execute a statement if a logical condition is **true**.
- Use the optional **else** clause to execute a statement if the condition is **false**.

```
if (condition) {
  //do domething
} else {
  //something else
}
```

Switch

- A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label.
- If a match is found, the program executes the associated statement.

```
switch (expression) {
   case label_1:
      statements_1
      break
   case label_2:
      statements_2
      break
   //...
   default:
      statements_def
      break
}
```

Loops

JavaScript supports the **for**, **do while**, and **while** loop statements:

```
for (let i = 0; i <= 10; i++) {
  console.log(i)
} // 0 1 2 3 4 5 6 7 8 9 10
```

```
let i = 0
do {
   console.log(i)
   i++
} while (i <= 10) // 0 1 2 3 4 5 6 7 8 9 10
```

```
let i = 0
while (i <= 10) {
   console.log(i)
   i++
} // 0 1 2 3 4 5 6 7 8 9 10
```

Ternary Operator

Like in other languages, we can use the conditional ternary operator:

```
const best = value > best ? value : best
```

- This operator takes a condition and two values.
- It returns the first value if the condition is true and the second if it is false.

Break and Continue

- The break statement finishes the current loop prematurely.
- The continue statement finishes the current iteration and continues with the next.

```
for (let i = 0; i < 10; i++) {
  if (i == 8) break
  if (i % 2 == 0) continue
  console.log(i)
} // 1 3 5 7
```

Functions

Defining functions

A function is defined using the **function** keyword.

```
function add(num1, num2) {
  console.log(num1 + num2)
}

add(1, 2) // 3
```

- **Primitive** parameters are passed to functions by **value**.
- **Non-primitive** parameters (objects) are passed by **reference**.

Return

Functions can also return values.

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

console.log(add(1, 2)) // 3
```

A function with an empty *return* or no *return* at all, returns **undefined**.

Default Values

- If a parameter expected by a function is not passed, it becomes **undefined**.
- Unless we declare a default value for that parameter.
- Default values can be complex expressions and are only calculated when needed.

```
let count = 1

function bar() {
  return count++
}

function foo(var1, var2 = 1234, var3 = bar()) {
  console.log(var1, var2, var3)
}

foo(10, 20, 30) // 10 20 30
foo(10, 20)     // 10 20 1
foo(10)         // 10 1234 2
foo()           // undefined 1234 3
```

Function Expressions

Another way to declare a function is the following:

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

This has the same effect as:

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

Functions are just another datatype stored in variables. We can even copy them or display them in the console:

```
const bar = foo
bar()
console.log(foo)
```

Functions as Parameters

We can pass functions as parameters to other functions.

```
function foo(i) {
  console.log('bar = ' + i)
}

function executeNTimes(f, n) { // Executes functi
  for (let i = 0; i < n; i++)
    f(i)
}

executeNTimes(foo, 3)   // bar = 0 bar = 1 bar =
executeNTimes(foo(), 3) // this is a common mista
```

Arrow Functions

A more compact way of declaring functions:

```
const foo = function(var1, var2) {
  return var1 + var2
}
```

Is the same as:

```
const foo = (var1, var2) => var1 + var2
```

Using the function from the previous slide:

```
executeNTimes((i) => console.log(i * i), 3)  // 0
executeNTimes(i => console.log(i * i), 3)    // E
```

Multi-line arrow functions are also possible using a code-block **{...}**.

Arrow Function Limitations

- Should not be used as methods (no *super* and no *this* binding, more on this later).
- Can not be used as constructors.
- Not ideal to use with *call*, *apply* and *bind* (more on this later).
- Cannot use *yield*.
- Multi-line arrow functions must have a return statement:

```
const sum = (a, b) => {
  const result = a + b
  return result
}
```

Objects

Objects

- JavaScript is designed on a simple **object-based** paradigm.
- An object is a collection of **properties**.
- A property is just an association between a **name** and a **value**.
- A property's value can be a function, in which case the property is known as a **method**.
- JavaScript is a **prototype-based** language and **does not** have a class statement (or does it?).

```
const person = { name: 'John Doe', age: 45 }

person.job = 'Driver'

console.log(person)
// Object { name: 'John Doe', age: 45, job: 'Driv
```

Methods

- Methods are properties of an object that happen to be functions.
- Methods are defined the way normal functions are defined, except that they are assigned as the property of an object.
- You can use the **this** keyword to refer to the current object within a method.

```
const person =
  {
    name: 'John Doe',
    age: 45,
    car: {make: 'Honda', model: 'Civic'},
    print: function() {
      console.log(`${this.name} is ${this.age} ye
    }
  }
person.print() // John Doe is 45 years old!
```

Assigning Methods

We can also assign a method to an object:

```
const person =
  { name: 'John Doe',
    age: 45,
    car: {make: 'Honda', model: 'Civic'},
  }

person.print = function() {
  console.log(`${this.name} is ${this.age} years
}

person.print() // John Doe is 45 years old!
```

Did we just change a constant???

Constant Objects

Like other types, constant objects cannot be reassigned:

```
const person = { name: 'John Doe' }
person = { name: 'Jane Doe' } // Error!
```

But we can change what's inside them:

```
const person = { name: 'John Doe' }
person.name = 'Jane Doe'
```

Getter and Setters

Setter and *getters*, accessor properties, are functions that execute on getting and setting a value, but behave like regular properties.

```
const person = {
    firstName: 'John',
    lastName: 'Doe',
    get fullName() {
      return `${this.firstName} ${this.lastName}`
    },
    set fullName (name) {
      const words = name.split(' ')
      this.firstName = words[0]
      this.lastName = words[1]
    }
}

person.fullName = 'John Doe'
console.log(person.firstName)  // John
console.log(person.lastName)   // Doe
console.log(person.fullName)   // John Doe

person.firstName = 'Jane'
console.log(person.fullName)   // Jane Doe
```

For ... in

The **for ... in** statement iterates over all properties of an object.

```
const person = { name: 'John Doe', age: 45 }

for (let key in person)
  console.log(`${key} = ${person[key]}`)

// name = John Doe
// age = 45
```

Objects as Arrays

- Properties of objects can be accessed or set using a bracket notation.
- Objects can be seen as **associative arrays** since each property is associated with a string value that can be used to access it.

```
const person = {}

person['name'] = "John Doe"
person['age'] = 45

console.log(person.age)    // 45
console.log(person['age']) // 45
```

Almost Everything is an Object

- In JavaScript, almost everything is an object.
- Even primitive types, except *null* and *undefined*, are temporarily *casted* into objects when treated as such.

```
const num = 10
console.log(num.toExponential()) // 1e+1

const name = "John Doe"
console.log(name.substring(0,4)) // John
```

In this example, the primitive types are *cast* temporarily into Number and String objects and discarded afterward.

Even Functions are Objects

They really are:

```
function foo() { console.log("Hello") }
const bar = function() { console.log("Hello") }
const baz = () => console.log("Hello")

foo(); bar(); baz() // Hello Hello Hello

foo.info = "This function says hello!"
bar.info = "This function says hello!"
baz.info = "This function says hello!"

console.log(foo.info)  // This function says hell

foo.goodBye = function() { console.log("Goodbye")
foo.goodBye() //Goodbye
```

This

This

In *JavaScript*, the **this** keyword (current context) behaves unlike in almost any other language.

- In the global execution context, **this** refers to the *global object*:
    - Or *window* if in a browser.
- Inside a function it depends on how the function was called:
    - Simple function call (**undefined** if in strict mode).
    - Arrow functions (**retain** the enclosing context).
    - Using *apply* or *call* (*this* is the **first** argument).
    - Object method (the object the method was **called** from).
    - Browser Events (the object that **fired** the event, more on this later).

This in Functions

Using **this** in simple functions:

```
'use strict'

function bar(var1, var2) {
  console.log(var1)
  console.log(var2)
  console.log(this)
}

bar(10, 20)                // 10 20 undefined
bar.call('foo', 10, 20)    // 10 20 foo
bar.apply('foo', [10, 20]) // 10 20 foo
```

- **Call** and **apply** are alternative ways to call functions that allow us to change the calling context (*this*).
- Both receive the **context** as the **first** argument.
- The remaining parameters are sent as **regular parameters** in *call* and as an **array** in *apply*.

Bind

The *bind* method allows us to fixate the *this* (or context of a function):

It receives a *context*, and returns a new function where *this* is **that** *context*.

```
'use strict'

function bar(var1, var2) {
  console.log(var1)
  console.log(var2)
  console.log(this)
}
bar(1, 2) // 1 2 undefined

const foo = bar.bind(3)
foo(1, 2) // 1 2 3
```

It can also be used to bind **parameters**. In this case it returns a function with only one parameter:

```
const baz = bar.bind('this', 'first')
baz('second')    // 'first' 'second' 'this'
```

This in Methods

In methods, *this* contains the object the method was called from:

```
const foo = {}

foo.bar = function() { console.log(this) }
foo.baz = () => console.log(this)

foo.bar()     // Object { bar: f, baz: f }
foo.baz()     // Window or Global

const bar = foo.bar
const baz = foo.baz

bar()         // Window or Global
baz()         // Window or Global
```

Arrow functions **do not have** a *this*, instead, they inherit it from the parent scope (lexical scoping).

Prototypes

Constructor Functions

Functions (but not *arrow* functions) can be used to create new objects using the **new** keyword.

```
function Person(name, age) {
  this.name = name
  this.age = age
  this.print = function() {
    console.log(`${this.name} is ${this.age} year
  }
}

const john = new Person("John Doe", 45)
john.print() // John Doe is 45 years old!
```

Cool! So, how does this work?

Prototype

- Each *JavaScript* function has an internal **prototype** property initialized as a nearly empty object.
- When the **new** operator is used on a constructor function, a new object derived from its prototype is created.
- The function is then executed, having the new object as its context.
- The new object is returned.

```
function Person(name, age) {
  this.name = name // this receives a nearly empt
  this.age = age   // based on the function's pro
  this.print = function() {
    console.log(`${this.name} is ${this.age} year
  }
}

const john = new Person("John Doe", 45)
john.print() // John Doe is 45 years old!
```

Changing the Prototype

We can inspect and change the prototype of a function:

```
function Person(name) {
  this.name = name
}

console.log(Person.prototype)       // {constructor: f}

const john = new Person("John Doe")
Person.age = 45                     // Only changes the Person functi
                                    // not its prototype.
const jane = new Person("Jane Doe")
console.log(jane.age)               // undefined

Person.prototype.age = 45           // Changes the prototype.

const mary = new Person("Mary Doe") // All objects constructed using
console.log(mary.age)  //45            person constructor now have an
console.log(jane.age)  //45            Even if created before the cha
```

What? How does THIS work?

Prototype of Objects

Every object has a **prototype of the function** that created them.

It can be accesses with **Object.getPrototypeOf(obj)** and modified using **Object.setPrototypeOf(obj, pro)**.

```
function Person(name) {
  this.name = name
}

const john = new Person('John Doe')

console.log(Object.getPrototypeOf(john) === Perso
// returns true

Object.setPrototypeOf(john, {})
// changes the prototype of john to {}
```

The Prototype Chain

When we read a property from an object, and it's missing, JavaScript will try taking it from the prototype of that object.

And then from the prototype of that prototype, until it reaches *null*.

```
function Person(name) {
  this.name = name
}

const john = new Person("John")

Person.prototype.age = 45
console.log(Object.getPrototypeOf(john))// Object
console.log(john.age)                   // 45

Object.setPrototypeOf(john, {})         // Change
console.log(john.age)                   // undefi
```

Because **john.age** does not exist, but **Object.getPrototypeOf(john).age** does.

Prototype Inheritance

Inheritance can be emulated with prototypes by changing the prototype chain.

```
function Person(name) { this.name = name }

Person.prototype.print = function() { console.log

function Worker(name, job) {
  this.job = job
  Person.call(this, name)  // super constructor w
}

Worker.prototype = new Person
Worker.prototype.print =
  function() { console.log(`${this.name} is a ${t

const mary = new Person("Mary")
mary.print() // Mary

const john = new Worker("John", "Builder")
john.print() // John is a Builder
```

Classes

Classes

- *Prototype-based* objects have many advantages (and disadvantages) over *class-based* objects.
- For example, we can do complicated meta-programming by manipulating the prototype chain.
- The original decision to use prototypes instead of classes in JavaScript as to do mainly with performance.

But why choose **one** when we can have **both**?

```
class Person {
  constructor(name) {
    this.name = name
  }

  print() {
    console.log(this.name)
  }
}
```

Syntatic Sugar

The *class* keyword is (almost) just *syntactic sugar* for *prototype-based* objects:

```
class Person {
  constructor(name) { this.name = name }
  print() { console.log(this.name) }
}
```

What's happening:

- A function named Person is being created.
- The function code is taken from the constructor method.
- Class methods, such as *print*, are stored in **Person.prototype**.

We can then use the **new** operator on that function just as we did before:

```
const john = new Person('John Doe')
```

Classes and the Prototype Chain

Inheritance is also just prototype chain manipulation:

```
class Person {
  constructor(name) { this.name = name }
  print() { console.log(this.name) }
}

class Worker extends Person {
  constructor(name, job) {
    super(name)
    this.job = job
  }
  print() { console.log(`${this.name} is a ${this
}

const john = new Worker("John", "Builder")
console.log(Object.getPrototypeOf(Worker) === Per
console.log(Object.getPrototypeOf(john) === Worke
// both return true
```

Classes Basic Syntax

- Classes can have **fields** (only very recently), **methods**, and a single **constructor**.
- The *this* keyword refers to the object that has called the method.

```
class Person {
  name     // undefined field
  age = 45 // a field initialized to a value

  constructor(name) { this.name = name } // a sin

  print() { console.log(this.name) }     // a met
}
```

The **new** operator creates a new instance of a class.

```
const john = new Person('John Doe')
```

Inheritance

Classes can extend other classes using the **extends** keyword.

```
class Person {
  constructor(name) { this.name = name }
  print() { console.log(`My name is ${this.name}`
}

class Worker extends Person {
  constructor(name, job) {
    super(name)
    this.job = job
  }
  print() {
    super.print()
    console.log(`And I'm a ${this.job}`)
  }
}
```

The **super** keyword allows calling the super-class constructor and methods.

Static

The static keyword allows declaring fields and methods as being part of the class (not the object).

- They must be accessed using the class and not an object.
- Inside a **static** method, **this** refers to the class.

```
class Person {
  static maxAge = 100

  constructor(name, age) {
    this.name = name
    this.age = age < Person.maxAge ? age : Person.maxAge
  }

  static compare(p1, p2) { return p1.name === p2.name && p1.age === p2
}

const john1 = new Person('John Doe', 120)
const john2 = new Person('John Doe', 100)

console.log(Person.compare(john1, john2)) // true
console.log(john1.maxAge, Person.maxAge) // undefined 100
```

Protected Fields and Methods

- There is no *language-level* way to create protected fields.
- But there is a *well-established* convention that fields/methods starting with an *underscore* should not be accessed directly.
- We can use this convention and *getters/setters* to create a *read-only* property:

```
class Person {
  constructor(name, age) {
    this.name = name
    this._age = age
  }

  get age() {
    return this._age
  }
}

const john = new Person('John Doe', 45)

console.log(john.age) // 45
john.age = 50         // no error, but no effect
console.log(john.age) // 45
```

Private Fields and Methods

- Unlike protected fields, there is a *language-level* way to create private fields.
- Like other field declaration aspects, this is still very recent and may not work everywhere.
- Private fields/methods are marked with a hash sign.

```
class Person {
  name
  #age

  constructor(name, age) {
    this.name = name
    this.#age = age
  }

  #compare(other) { return this.name === other.name && this.age === ot
}

const john1 = new Person('John Doe', 45)
const john2 = new Person('John Doe', 55)

console.log(john1.age)              // undefined
console.log(john1.#age)             // error
console.log(john1['#age'])          // undefined
console.log(john1.#compare(john2)) // error
```

Instance Of

You can use the **instanceof** operator to check if an object belongs to a specific class:

```
class Person {
  constructor(name) { this.name = name }
}

class Worker extends Person {
  constructor(name, job) {
    super(name)
    this.job = job
  }
}

const john = new Person('John Doe')
const jane = new Worker('Jane Doe', 'Builder')

console.log(john instanceof Person) // true
console.log(john instanceof Worker) // false
console.log(jane instanceof Person) // true
console.log(jane instanceof Worker) // true
```

Arrays

Arrays

- Arrays are **list-like objects** whose prototype has methods to perform traversal and mutation operations.
- *JavaScript* arrays are zero-indexed
- Arrays can be initialized using a bracket notation:

```
const years = [1990, 1991, 1992, 1993]
console.log(years[0])      // 1990
years.info = "Nice array" // Arrays are objects
console.log(years.info)    // Nice array
```

Array elements are object properties, but they cannot be accessed using the **dot** notation because their names are invalid.

```
const years = [1990, 1991, 1992, 1993]
console.log(years[0]) // 1990
console.log(years.0)  // Syntax error
```

Array Looping

You can use a **for** loop to iterate over array elements:

```
const years = [1990, 1991, 1992, 1993]
for (let i = 0; i < years.length; i++)
  console.log(years[i])
```

Or you can use a **for ... of** loop:

```
const years = [1990, 1991, 1992, 1993]
for (const year of years)
  console.log(year)
```

> ❗ Do not use a **for ... in** loop! Those are for object properties.

Array Prototype

These are some of the methods defined by the Array prototype:

- Properties: prototype, length
- Mutators: fill, pop, push, reverse, shift, sort, splice, unshift
- Accessor: concat, contains, join, slice, indexOf, lastIndexOf
- Iterator: forEach, entries, every, some, filter

Some examples:

```
const years = [1990, 1991, 1992, 1993]
years.push(1994)
console.log(years.length)   // 5

years.reverse()
console.log(years)          // [1994, 1993, 1992, 1991, 1990]

let sum = 0
years.forEach(e => sum += e)
console.log(sum)            // 9960

years.every(e => e >= 1990) // true
years.some(e => e % 2 == 0) // true
```

Playing with the Array Prototype

We can add methods and properties to all arrays by changing the Array prototype:

```
const years = [1990, 1991, 1992, 1993]

Array.prototype.print = function() {
  console.log("This array has length " + this.len
}

years.print() // This array has length 4
```

forEach()

The forEach() method **executes** a function once **for each** array element:

```
const numbers = [4, 8, 15, 16, 23, 42]
numbers.forEach(function(value, index){
    console.log('Element #' + index + ' is ' + va
})
```

The result would be:

```
Element #0 is 4
Element #1 is 8
Element #2 is 15
Element #3 is 16
Element #4 is 23
Element #5 is 42
```

filter()

The filter() method returns a **new array** with all elements that **pass a test**.

```
const numbers = [4, 8, 15, 16, 23, 42]
const even = numbers.filter(function(n) { return
console.log(even) // [ 4, 8, 16, 42 ]
```

Or using arrow functions:

```
const numbers = [4, 8, 15, 16, 23, 42]
const even = numbers.filter(n => n % 2 == 0)
console.log(even) // [ 4, 8, 16, 42 ]
```

An alternative would be:

```
const numbers = [4, 8, 15, 16, 23, 42]
const even = []
for (let i = 0; i < numbers.length; i++)
  if (numbers[i] % 2 == 0) even.push(numbers[i])
console.log(even) // [ 4, 8, 16, 42 ]
```

map()

The map() method creates a **new array** by **applying a function** to every element in the original array.

```
const numbers = [4, 8, 15, 16, 23, 42]
const doubled = numbers.map(function(n) { return
console.log(doubled) // 8, 16, 30, 32, 46, 84
```

Or using **arrow functions**:

```
const numbers = [4, 8, 15, 16, 23, 42]
const doubled = numbers.map(n => n * 2)
console.log(doubled) // 8, 16, 30, 32, 46, 84
```

Generic use of map()

The map() method can be used on **other types** of *array-like* objects:

```
const ascii = Array.prototype.map.call('John', l
console.log(ascii) // [74, 111, 104, 110]
```

Simpler:

```
const ascii = [].map.call('John', letter => lette
console.log(ascii) // [74, 111, 104, 110]
```

A more useful example:

```
const inputs = document.querySelectorAll('input[t
const values = [].map.call(inputs, input => input
console.log(values) // an array with all the numb
```

reduce()

The reduce() method **applies a function** to each element in the array:

The result is passed to the next iteration as an **accumulator** (starting at 0 by default). The objective is to **reduce** the array to a **single value** in the end.

```
const numbers = [4, 8, 15, 16, 23, 42]
const total = numbers.reduce(function(accumulator
  return accumulator + number
})
console.log(total) // 108
```

Or with **arrow functions**:

```
[4, 8, 15, 16, 23, 42].reduce( (acc, num) => acc
```

We can **initialize** the accumulator by adding a second parameter:

```
[4, 8, 15, 16, 23, 42].reduce( (acc, num) => acc
```

Spread Operator

The spread operator allows an iterable, such as an array or string, to be **expanded** in places where zero or more arguments are expected:

```
function sum(x, y, z) {
  return x + y + z
}

const numbers = [1, 2, 3]

console.log( sum(...numbers) ) // 6
```

Other example:

```
function sum(...args) { // sum any number of args
  let sum = 0
  for (let i = 0; i < args.length; i++)
    sum += args[i]
  return sum
}
console.log( sum(1, 2, 3) ) // 6
```

Destructuring

Array Destructuring

Destructuring assignment allows us to split an array (or any iterable) into separate variables:

```
const names = ['John', 'Doe']
const [first, last] = names
console.log(first) // John
```

It also works with fields (first, we split a string into an array):

```
const person = {}
[person.first, person.last] = 'John Doe'.split('
console.log(person) // {first: 'John', last: 'Doe
```

Swap and much more:

```
let a = 10, b = 5
[b, a] = [a, b]
console.log(a, b) // 5 10
```

The Rest

After all elements are assigned, the remaining (or "the rest") can be assigned too, using the *spread operator* (...):

```
const numbers = [1, 2, 3, 5, 8]
const [a, b, ...r] = numbers
console.log(a)  // 1
console.log(b)  // 2
console.log(r)  // [3, 5, 8]
```

Destructuring Objects

Destructuring also works with objects:

```
const person = { first: 'John', last: 'Doe', age:
const {first, last} = person
console.log(first) // John
```

The order does not matter, and we can assign to variables with different names:

```
const person = { first: 'John', last: 'Doe', age:
const {age: a, first: f, last: l} = person
console.log(a) // 45
console.log(f) // John
console.log(l) // Doe
```

Destructuring in Functions

We can use destructuring when defining functions:

```
function sum(...numbers) {
  let sum = 0
  for (const n of numbers)
    sum += n
  return sum
}

console.log(sum(1, 2, 3)) // 6
```

And even with objects:

```
function print({first, last}) {
  console.log(`${first} ${last}`)
}

print({first: 'John', last: 'Doe', age: 45}) // J
```

Map and Set

Map

- A *Map* is a collection of key-value pairs that allows keys of any type (even objects).
- You can **get**, **set**, and **delete** values from a Map.
- You can also check (**has**) if a key exists in the Map.
- And **clear** all values.

```
const map = new Map()

map.set('name', 'John Doe')
map.set('age', 45)
map.set(10, 'it is a number')

map.delete(10)

console.log(map.has('name'))  // true
console.log(map.has(10))      // false
console.log(map.get('age'))   // 45

map.clear()
```

Map Looping

There are three ways to access all elements of a Map:

- **.keys()** – returns an iterable for keys
- **.values()** – returns an iterable for values
- **.entries()** – returns an iterable for entries

The **.entries()** method is the *default* when using **for ... of** loops:

```
const map = new Map([['name', 'John Doe'], ['age'
for (const [key, value] of map)
  console.log(`${key} = ${value}`)
```

We can also initialize a *Map* with an *iterable* of *key-value* pairs (like a nested *Array*).

Set

- A *Set* is a collection of values (of any type) that cannot contain repeated values.
- You can **add**, and **delete** values from a Set.
- You can also check (**has**) if a value exists in the Set.
- And **clear** all values.

```
const set = new Set()

set.add('John Doe')
set.add('Jane Doe')

console.log(set.size) // 2
set.add('John Doe')
console.log(set.size) // still 2

set.delete('Jane Doe')

console.log(set.has('John Doe'))  // true
console.log(set.has('Jane Doe'))  // false
```

Set Looping

We can loop over the elements in a Set using **for … of** loops:

```
const set = new Map(['John Doe', 'Jane Doe'])
for (const element of set)
  console.log(element)
```

We can also initialize a *Set* with an *Array*.

Error Handling

Try … Catch … Finally

- The **try** block contains statements to *try*.
- The **catch** block contains code to deal with any exception thrown inside the **try** block.
- The **finally** block executes regardless of whether an exception is thrown. Useful for cleanup operations (e.g., closing a connection).

```
try {
  doesThisFunctionExist() // it doesn't
  console.log('I will not print')
} catch (e) {
  console.log(e)          // prints the not defin
  throw new Error('burp') // uncaught exception
} finally {
  console.log('I always print')
}

console.log('I might not print')
```

Throw

You can throw exceptions using the **throw** statement. You can throw any expression.

```
try {
  throw 'Whoops!'
} catch (e) {
  console.error(`e`) // Whoops!
}
```

If you are throwing your own exceptions, to take advantage of the name and message properties, you can use the **Error** constructor.

```
try {
  throw new Error('Whoops!')
} catch (e) {
  console.error(`${e.name}: ${e.message}`) // Err
}
```

Or extend the **Error** class.

Dealing with different Exceptions

To distinguish between different types of exceptions, we can use **instanceof**:

```
try {
  // code to try
}
catch (e) {
  if (e instanceof DatabaseError) {
    // statements to handle DatabaseError excepti
  }
  if (e instanceof SomethingElseError) {
    // statements to handle SomethingElseError ex
  }
}
```

Scope

Code Blocks

If a variable is defined inside a **code block**, it is only visible inside that code block:

```
{
  const name = 'John Doe'
  console.log(name)        // John Doe
}
console.log(name)          // undefined
```

We can use this to create **nested functions** (functions are like any other type):

```
function equal(a, b) {
  function difference(a, b) { return b - a }
  return difference(a, b) === 0
}
console.log(equal(10, 10)) // true
difference(10, 10)         // error
```

Lexical Environments

When we have nested blocks, each one has a **Lexical Environment** where local variables are stored.

Each one of these environments has a **pointer** to the lexical environment where it was created.

```
function equal(a, b) {
  function difference(a, b) { return b - a }
  return difference(a, b) === 0
}
console.log(equal(10, 10)) // true
difference(10, 10)         // error
```

Like this: **difference** → **equal** → *global*

Scope

- When we reference a variable, it is **first** searched in the current lexical environment.
- If it isn't found, it is searched in the **outer** lexical environment. This goes on until the global environment is reached.
- That's why using variables that have not been declared is a **bad idea**. They will bubble up until the global lexical environment and become **global variables**.

Closures

When a function is created, it **retains** the lexical environment in which it was **created**.

That's why this code works:

```
function createCounter() {
  let counter = 0
  return function() {
    return ++counter
  }
}

const counter = createCounter()
console.log(counter()) // 1
console.log(counter()) // 2
console.log(counter()) // 3
```

A **closure** is the combination of a function bundled together with its surrounding lexical environment.

Closures

A **new closure** is created everytime a function is created:

```
function createCounter() {
  let counter = 0
  return function() {
    return ++counter
  }
}

const counter1 = createCounter()
const counter2 = createCounter()

console.log(counter1()) // 1
console.log(counter1()) // 2
console.log(counter2()) // 1
console.log(counter1()) // 3
console.log(counter2()) // 2
```

Asynchronous Code

Callbacks and Promises

JavaScript Engines

- JavaScript code is executed by a **JavaScript Engine**.
- Some notable examples: V8 (Chrome, Node.js), SpiderMonkey (Firefox), and JavaScriptCore Safari.
- They provide a heap, a single call stack, and a way to run JavaScript code.

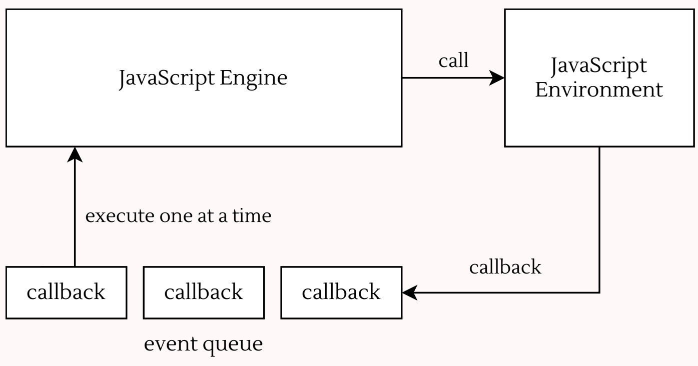
However:

- JavaScript is a **single-threaded** language!
  An *engine* does not provide a way to start new threads.
- There is also no way to do input/output.
  e.g., networking, storage, graphics.

> So how can we get asynchronous code?

JavaScript Environments

- **JavaScript Runtime Environment**s provide the necessary APIs to do I/O.
  For example, both Chrome and Node.js use the same engine (V8) but provide very **different** environments.
- These environments also allow us to schedule **asynchronous** actions (*e.g.*, timers, events, network).
  Actions that are independent of the main program flow.
- These actions run in **separate** and **independent** threads).
- When they finish, they put a callback function on an *event queue*, waiting to be executed.



The Event Loop

Consider the following code where **readFile** is an **asynchronous** function provided by some *runtime environment*:

```
const path = '/some/large/file/we/want/to/read.txt'

readFile(path, function(error, content) {
  if (error) handleError(error) // if there is an error, we handle it
  else console.log(content)     // when the file is read, this is exe
})
```

- The **readFile** function asks the *environment* to read a file.
  The environment returns imediately and starts reading the file in a separate process.
- When the *environment* finishes reading the file, the *callback* function is placed in an *event queue*.
- Tasks in this *queue* are executed only when the *call stack* becomes empty.
  In a FIFO order.

ℹ The **event loop** is an endless loop where the JavaScript *engine* **waits** for tasks, **executes** them, and then **waits** for more tasks.

Callback Hell

What happens if we need to read a series of files, one after the other?

We will end up with code like this:

```
readFile('file1.txt', function(error, content1) {
  if (error) handleError(error)
  else readFile('file2.txt', function(error, cont
    if (error) handleError(error)
    else readFile('file3.txt', function(error, co
      if (error) handleError(error)
      else readFile('file4.txt', function(error,
        if (error) handleError(error)
        else console.log(content1, content2, cont
      })
    })
  })
})
```

This is called *callback hell* or the *pyramid of doom*!

Why don't we use synchronous code?

Imagine we had a different version of the **readFile** function that worked **synchronously**:

```
const content1 = readFileSync('file1.txt')
const content2 = readFileSync('file2.txt')
const content3 = readFileSync('file3.txt')
const content4 = readFileSync('file4.txt')
```

- This would be much nicer, but JavaScript is **single-threaded**.
- If these operations take a lot of time, the code will **hang** for the whole duration.

Promises

Promises solve this problem in a very elegant way.

- A promise represents the **eventual result** of an **asynchronous** operation.
- A promise may be in one of 3 possible states: **fulfilled**, **rejected**, or **pending**.
- A Promise is an *object* that takes a **function** with two parameters, functions **resolve** and **reject**:

```
const promise = new Promise((resolve, reject) =>
  readFile('file.txt', (err, data) => {
    if (err) reject(err)
    else resolve(data)
  })
})
```

Consuming

When the *promise resolves* or is *rejected*, we can use **.then** and **.catch** to consume it:

```
promise.then(function(content) {
  console.log(content)
}).catch(function(error) {
  handleError(error)
})
```

This might not seem much better, but *promises* still have some tricks left!

Returning Promises

The idea behind *promises* is that **instead** of using *callbacks* to transform *synchronous* into *asynchronous* code, *asynchronous* functions should return *promises* instead:

```
function promiseFile(filename) {
  return new Promise((resolve, reject) => {
    readFile(filename, (err, data) => {
      if (err) reject(err)
      else resolve(data)
    })
  })
}
```

This could then be used like this:

```
promiseFile('file.txt')
  .then(content => console.log(content))
  .catch(error => console.error(error))
```

Promise Chaining

If we return a *promise* from a **.then** handler, we can chain *promises*:

```
promiseFile('file1.txt')
  .then(content => {
    console.log(content)
    return promiseFile('file2.txt')
  })
  .then(content => {
    console.log(content)
    return promiseFile('file3.txt')
  })
  .then(console.log)    // this is not magic!
  .catch(console.error) // one catch for all the
```

- In fact, **.then** and **.catch** handlers always return *promises*.
- If the code inside them returns something else, the result is wrapped in an automatically fulfilled *promise*.
- This simplifies *promise chaining* (no more *callback hell*).

Error Handling

Promises have an implicit **try ... catch** block around their code.

So, if **readFileSync** throws an error, we don't even need to call **reject**:

```
function promiseFile(filename) {
  return new Promise(function(resolve, reject)) {
    const content = readFileSync('file.txt') // throws an error if it
    resolve(content)
  }
}
```

It also happens in promise handlers (**.then** and **.catch**).

If we throw inside a **.then** handler, the control jumps to the nearest **.catch**.

```
promiseFile('file1.txt')
  .then(console.log)
  .catch(console.error)                 // error reading file1.text
  .then(_ => promiseFile('file2.txt')) // needs to be a function
  .then(console.log)
  .catch(console.error)                 // error reading file2.text
  .then(_ => promiseFile('file3.txt'))
  .then(console.log)
  .catch(console.error)                 // error reading file3.text
```

Promise.all

There is also an easy way to run several *promises* in parallel and **wait** for them **all**:

```
Promise.all([promiseFile('file1.txt'),
             promiseFile('file2.txt'),
             promiseFile('file3.txt')])
  .then(([c1, c2, c3]) => console.log(c1, c2, c3))
  .catch(console.error)
```

- **Promise.all** receives an array of *promises*.
- The **.then** handler is called when they all resolve.
- If any of them *throw* an *error* (or call *reject*), then **.catch** is called.
- We are using destructuring to receive all the results in separate variables.

Async

When we add the **async** keyword before a function declaration then that function always returns a promise:

```
async function getName() {
  return 'John Doe'
}
```

So this would be possible:

```
getName().then(console.log) // John Doe
```

And this would be our read function:

```
async function promiseFile(filename) {
  return new Promise((resolve, reject) => {
    readFile(filename, (err, data) => {
      if (err) reject(err)
      else resolve(data)
    })
  })
}
```

Await

The keyword *await* makes *JavaScript* wait until a promise settles and returns its result:

- *Await* only works inside *async* functions.
- *Await* uses the event loop mechanism; the code is suspended until the promise settles and a new *callback* is added to the *event queue*. This way, no CPU resources are wasted.

It's just a more elegant way to use sequential promises:

```
async function foo() {
  const name = await readFile('file.txt')
}
```

If an error is thrown, the *promise* returned by the *async* function is **rejected**.

Async/Await

Putting it all together, we can write:

```
async function foo() {
  const c1 = await promiseFile('file1.txt')
  const c2 = await promiseFile('file2.txt')
  const c3 = await promiseFile('file3.txt')
  console.log(c1, c2, c3)
}

foo().catch(console.error)
```

And we get *synchronous-like* code that behaves in a **non-blocking** manner.

JSON

JSON

- JSON (**J**ava**S**cript **O**bject **N**otation) is a *lightweight data-interchange format*. Some alternatives are YAML and TOML.
- It is easy for humans to **read** and **write**.
- It is easy for machines to **parse** and **generate**.

```
const posts = [
  {
   "id":"1",
   "title":"Mauris...",
   "introduction":"Sed eu...",
   "fulltext":"Donec feugiat..."
  }, {
   "id":"2",
   "title":"Etiam efficitur...",
   "introduction":"Cum sociis ...",
   "fulltext":"Donec feugiat..."
  }
]
```

JSON

The **JSON.stringify** and **JSON.parse** functions can be used to encode from and to JSON easily.

```
const encoded = JSON.stringify(posts)  // retur
const decoded = JSON.parse(encoded)    // same
```