JavaScript - Basics

Tecnologie Internet

a.a. 2022/2023

JavaScript: the most-used programming language

JavaScript is **the programming language of the Web**.

The majority of modern websites use JavaScript, and all modern web browsers (on desktops, game consoles, tablets and smart phones) include **JavaScript interpreters**.

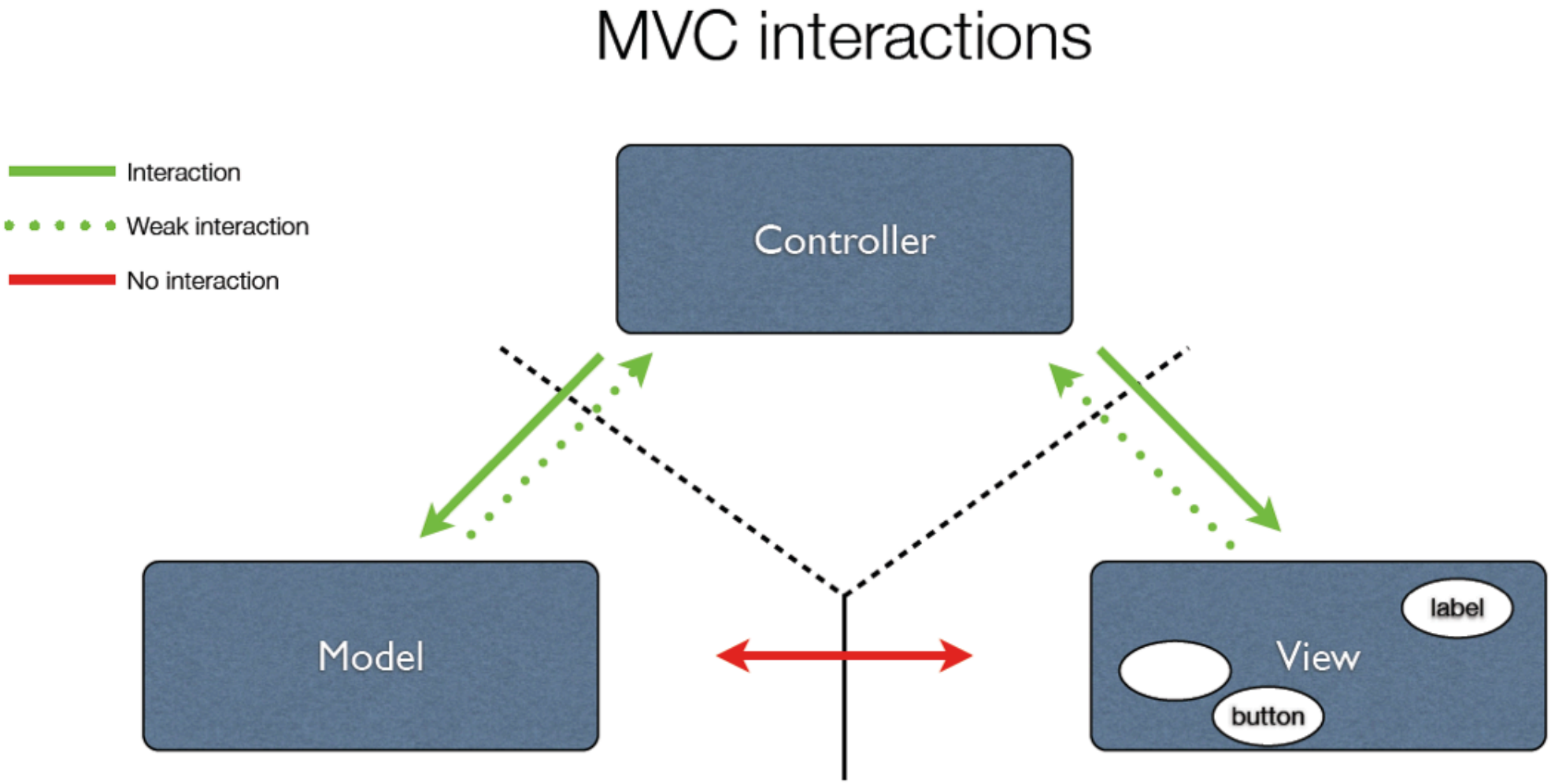
Over the last decade, **Node** has enabled JavaScript programming outside of web browsers, and the dramatic success of Node means that JavaScript is now also the most-used programming language among software developers.

Model-View-Controller (MVC)

MVC is an *architectural pattern* used to organize parts of code into clean and separate fields, according to the responsibilities and features of each of them.

- The **Model** is the representation of the data that will be used in the application; the model is independent from the View, since it does not know how data will be displayed
- The **View** is the user interface that will display the application's contents; the view is independent from the Model since it contains a bunch of graphical elements that can be used in any application (e.g. buttons, labels, sliders, ...)
- The **Controller** manages how the data in the Model should be displayed in the View; it is highly dependent from the Model and the View since it needs to know which data it will handle and which graphical elements it will need to interact with

Model-View-Controller (MVC)



Model-View-Controller (MVC)

Thus, dynamic web pages may be developed using the MVC pattern, according to the following approach:

XML, JSON --> Model

HTML + CSS --> View

JavaScript --> Controller

Note: JSON data structures map 1-to-1 onto JavaScript objects!

JavaScript: the story so far

The name "JavaScript" is quite misleading.

JavaScript is not Java (their syntax is somehow similar as they both derive from C, but that's all) and has long since outgrown its scripting-language roots to become a robust and efficient general-purpose language suitable for large software projects.

JavaScript was created at Netscape (now Mozilla) and its original name was LiveScript. When Netscape started supporting Java into its browser (Netscape Navigator), LiveScript became JavaScript (generating a lot of confusion).
Microsoft introduced a compatible language called JScript.

The need for a commonly accepted specification led to **ECMAScript**, the official standardization of the language (by the European Computer Manufacturer's Association).

JavaScript: the story so far

June 1997 - ECMAScript 1

June 1998 - ECMAScript 2

December 1999 - ECMAScript 3

(ECMAScript 4 was abandoned)

December 2009 - ECMAScript 5

June 2015 - **ECMAScript 6**

...

June 2022 - ECMAScript 13

JavaScript: the reasons of a success

The core JavaScript language defines a minimal API for working with text, arrays, dates and regular expressions, but does not include any input or output functionality.

Input and output, as well as networking, storage and graphics, are the responsibility of the "host environment" within which JavaScript is embedded:

- the **web browser** (the most obvious interpreter for client-side JavaScript)

- **Rhino** (a Java-based interpreter that gives JavaScript programs access to the entire Java API)

- **Node** (a JavaScript interpreter with low-level bindings for the POSIX API and a particular emphasis on asynchronous I/O and networking)

Embedding JavaScript in HTML

Client-side JavaScript code is embedded within HTML documents in four ways:

1) **Inline**, between a pair of `<script>` and `</script>` tags

2) From an **external file** specified by the `src` attribute of a `<script>` tag

3) In an HTML **event handler** attribute, such as `onclick` or `onmouseover`

4) **In a URL** that uses the special `javascript:` protocol

Embedding JavaScript in HTML

A programming philosophy known as **unobtrusive JavaScript** argues that content (HTML) and behavior (JavaScript code) should as much as possible be kept separate.

This approach allows for JavaScript code reuse and HTML simplification.

Thus, JavaScript is best embedded in HTML documents using `<script>` elements with `src` attributes.

```
<script src="../../scripts/util.js"></script>
```

A JavaScript file contains pure JavaScript, without any HTML.

Embedding JavaScript in HTML

When the browser loads the HTML file, it will execute the JavaScript code.

Today's web browsers all include web developer tools that are indispensable for debugging, experimenting and learning.

For testing the code in these slides, use **W3C's TryIt editor**:

http://www.w3schools.com/html/tryit.asp?filename=tryhtml_basic

Embedding JavaScript in HTML

Inline JavaScript example:

```
<!DOCTYPE html>
<html>
<body>

<h1>My Web Page</h1>

<p id="demo">A Paragraph</p>

<button type="button" onclick="myFunction()">Try it</button>

<script>
function myFunction() {
   document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>

</body>
</html>
```

Note: JavaScript code can be embedded in <head> as well!

Output

JavaScript does NOT have any built-in print or display functions.

JavaScript can "display" data in different ways:

- Writing into an alert box, using **window.alert()**
- Writing into the HTML output using **document.write()**
- Writing into an HTML element, using **innerHTML**
- Writing into the browser console, using **console.log()**

Output

Writing into an alert box, using **window.alert()**

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
window.alert(5 + 6);
</script>

</body>
</html>
```

Output

Writing into the HTML output using **document.write()**

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
document.write(5 + 6);
</script>

</body>
</html>
```

Note: using document.write() after an HTML document is fully loaded, will **delete all existing HTML**.

The document.write() method should be used only for testing.

Output

Writing into an HTML element, using **innerHTML**

The **id** attribute defines the HTML element. The **innerHTML** property defines the HTML content:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My First Paragraph</p>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>

</body>
</html>
```

Output

Writing into the browser console, using **console.log()**

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
console.log(5 + 6);
</script>

</body>
</html>
```

Statements

Statements are executed to make something happen.

JavaScript statements are composed of keywords, values, comments, operators and expressions, and they are separated by semicolons.

```
let x = 5;
let y = 6;
let z = x + y;
```

Reserved keywords

abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const	continue
debugger	default	delete	do
double	else	enum*	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected
public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

Words marked with* are new in ECMAScript 5 and 6.

Constants and variables

The JavaScript syntax defines two types of values: **constants** and **variables**.

Constants are declared with `const` and variables are declared with `let` (or with `var` in older JavaScript code).

JavaScript constants and variables are **untyped**: declarations do not specify what kind of values will be assigned.

Comments

Not all JavaScript statements are "executed".

Code after double slashes `//` or between `/*` and `*/` is treated as a **comment**.

Comments are ignored, and will not be executed:

```
let x = 5;   // I will be executed

// let x = 6;   I will NOT be executed
```

Identifiers

In JavaScript, identifiers are used to name variables, functions, classes, etc.

The first character must be a letter, an underscore (_), or a dollar sign ($).
Subsequent characters may be letters, digits, underscores, or dollar signs.

Numbers are not allowed as the first character.
This way JavaScript can easily distinguish identifiers from numbers.

All JavaScript identifiers are **case sensitive**.
The variables `lastName` and `lastname` are two different variables:

```
let lastName = "Doe";
let lastname = "Peterson";
```

Identifiers

Historically, programmers have used three ways of joining multiple words into one variable name:

Hyphens:
first-name, last-name, master-card, inter-city.

Underscore:
first_name, last_name, master_card, inter_city.

Camel Case:
FirstName, LastName, MasterCard, InterCity.

In programming languages, especially in JavaScript, camel case often starts with a lowercase letter: firstName, lastName, masterCard, interCity.

Hyphens are not allowed in JavaScript. - is reserved for subtractions.

Numbers and strings

Numbers are written with or without decimals:

`10.50` floating point literal

`1001` integer literal

Strings literals are written within double or single quotes:

```
"John Doe"
```

```
'John Doe'
```

Escape sequences:

```
'You\'re right, it can\'t be a quote'
```

```
'\n'
```

etc.

Type conversions

In programming, data types is an important concept.

Without data types, a computer cannot safely solve this:

```
let x = 16 + "Volvo";
```

JavaScript will treat the example above as:

```
let x = "16" + "Volvo";
```

If the second operand is a string, JavaScript will also treat the first operand as a string.

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

```
let x = 16 + 4 + "Volvo";      ----->    20Volvo
let x = "Volvo" + 16 + 4;      ----->    Volvo164
let x = "16" + "4";            ----->    164
```

Type conversions

Value	to String	to Number	to Boolean
`undefined`	`"undefined"`	`NaN`	`false`
`null`	`"null"`	**`0`**	`false`
`true`	`"true"`	**`1`**	
`false`	`"false"`	**`0`**	
`""` (empty string)		**`0`**	**`false`**
`"1.2"` (nonempty, numeric)		`1.2`	`true`
`"one"` (nonempty, non-numeric)		`NaN`	`true`
`0`	`"0"`		**`false`**
`-0`	`"0"`		**`false`**
`1` (finite, non-zero)	`"1"`		`true`
`Infinity`	`"Infinity"`		`true`
`-Infinity`	`"-Infinity"`		`true`
`NaN`	`"NaN"`		**`false`**
`{}` (any object)	*see §3.9.3*	*see §3.9.3*	`true`
`[]` (empty array)	`""`	**`0`**	`true`
`[9]` (one numeric element)	`"9"`	**`9`**	`true`
`['a']` (any other array)	*use join() method*	`NaN`	`true`
`function(){}` (any function)	*see §3.9.3*	`NaN`	`true`

Variable declaration and assignment

An **assignment operator** = is used to **assign values** to constant and variables.

```
const C = 74;
let x;
x = 6;
```

Variables can be numbers, strings, booleans or objects.

```
let length = 16;                                // Number
let lastName = "Johnson";                       // String
let cars = ["Saab", "Volvo", "BMW"];            // Array
let x = {firstName:"John", lastName:"Doe"};     // Object
```

`null` and `undefined` are special values that mean “no value”.

Operators

Arithmetic operators (all numbers are floating-point):

```
+   -   *   /   %   ++   --
```

Relational operators:

```
<   <=   ==   ===   !=  !==   >=   >
```

Logical operators:

```
&&   ||   !
```

Bitwise operators:

```
&   |   ^   ~   <<   >>   >>>
```

Assignment operators:

```
+=   -=   *=   /=   %=   <<=   >>=   >>>=   &=   ^=   |=
```

Operators

String concatenation:
+

Conditional operators:
(cond.) **?** (true_value) **:** (false_value)

Other operators:
new typeof void delete

Expressions

An **expression** is a combination of values, variables, and operators, which computes to a value.

The computation is called an **evaluation**. For example, 5 * 10 evaluates to 50.

Expressions can also contain variable values.

The values can be of various types, such as numbers and strings. For example, "John" + " " + "Doe", evaluates to "John Doe".

Conditional statements

- Use **`if`** to specify a block of code to be executed, if a specified condition is true
- Use **`else`** to specify a block of code to be executed, if the same condition is false
- Use **`else if`** to specify a new condition to test, if the first condition is false
- Use **`switch`** to specify many alternative blocks of code to be executed

Conditional statements

```
if (time < 10) {
    greeting = "Good morning";
} else if (time < 20) {
    greeting = "Good day";
} else {
    greeting = "Good evening";
}
```

```
switch (new Date().getDay()) {
    case 0:
        day = "Sunday";
        break;
    case 1:
        day = "Monday";
        break;
    case 2:
        day = "Tuesday";
        break;
    case 3:
        day = "Wednesday";
        break;
    case 4:
        day = "Thursday";
        break;
    case 5:
        day = "Friday";
        break;
    case 6:
        day = "Saturday";
        break;
}
```

Looping statements

- **`for`** - loops through a block of code a number of times
- **`for/in`** - loops through the properties of an object
- **`while`** - loops through a block of code while a specified condition is true
- **`do/while`** - also loops through a block of code while a specified condition is true

Looping statements

```
for (let i = 0, len = cars.length, text = ""; i < len; i++) {
    text += cars[i] + "<br>";
}
```

```
let person = {fname:"John", lname:"Doe", age:25};
let text = "";
let x;
for (x in person) {
    text += person[x];
}
```

Looping statements

```
let i = 1;
while (i < 10) {
    text += "The number is " + i;
    i++;
}
```

```
let i = 1;
do {
    text += "The number is " + i + "\n";
    i++;
}
while (i < 10);
```

Jumps

The **break statement** breaks a loop (or switch) and continues executing the code after the loop (or switch):

```
for (let i = 0; i < 10; i++) {
    if (i === 3) { break; }
    text += "The number is " + i + "<br>";
}
```

The **continue statement** breaks one iteration (in a loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
for (let i = 0; i < 10; i++) {
    if (i === 3) { continue; }
    text += "The number is " + i + "<br>";
}
```

Functions

A JavaScript function is a block of code designed to perform a particular task. It is executed when "something" invokes it (calls it).

```
function myFunction(p1, p2) {
    return p1 * p2;
}
```

A JavaScript function is defined with the **`function`** keyword, followed by a **`name`**, followed by parentheses **`()`**.
Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).
The parentheses may include parameter names separated by commas: **`(parameter1, parameter2, ...)`**

The code to be executed, by the function, is placed inside curly brackets: **`{}`**

Functions

Function **parameters** are the **names** listed in the function definition.
Function **arguments** are the real **values** received by the function when it is invoked.
Inside the function, the arguments are used as local variables.

When JavaScript reaches a **return statement**, the function will stop executing.
If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

```
let x = myFunction(4, 3);

function myFunction(a, b) {
    return a * b;
}
```

Arrow Functions

In ES6, you can define functions using a particularly compact syntax known as “arrow functions.”

The function keyword is not used, and, since arrow functions are expressions instead of statements, there is no need for a function name, either.

The general form of an arrow function is a comma-separated list of parameters in parentheses, followed by the => arrow, followed by the function body in curly braces:

```
const sum = (x, y) => { return x + y; };
```

If the body of the function is a single return statement, you can use a more compact syntax:

```
const sum = (x, y) => x + y;
```

Objects

JavaScript variables are containers for data values.

```
let car = "Fiat";
```

Objects are variables too. But objects can contain many **named values**, written as **name:value** pairs (called **properties**).

```
let car = {type:"Fiat", model:500, color:"white"};
```

Objects may contain also methods, i.e., **actions** that can be performed on objects. Methods are stored in properties as **function definitions**.

```
let person = {
    firstName:"John",
    lastName:"Doe",
    age:50,
    eyeColor:"blue",
    fullName:function() {return this.firstName + " " + this.lastName;}
};
```

Objects

We can access object properties in two ways:

```
objectName.propertyName
```

```
objectName[propertyName]
```

To access an object method, we use the following syntax:

```
objectName.methodName()
```

Objects

If you access the fullName **property**, without (), it will return the **function definition**:

```
<p id="demo"></p>

<script>
let person = {
    firstName: "John",
    lastName : "Doe",
    id       : 5566,
    fullName : function() {
       return this.firstName + " " + this.lastName;
    }
};

document.getElementById("demo").innerHTML = person.fullName;
</script>
```

Output:

```
function () { return this.firstName + " " + this.lastName; }
```

Objects

With JavaScript, you can define and create your own objects.
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 function**, and then create objects of the constructed type.
- Define a **class** and create objects of that class using the keyword **new**. [this option was introduced by ECMAScript2015, aka ES6]

Objects

Using an **object literal** is the easiest way to create a JavaScript Object.

Using an object literal, we both define and create an object in one statement.

An object literal is a list of name:value pairs (like age:50) inside curly braces {}.

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

Objects

Creating an object with the JavaScript keyword **new**:

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

For simplicity, readability and execution speed, it is better to use the object literal method than the keyword new.

Objects

The examples above are limited in many situations. They only create a single object.

Sometimes we like to have an "object type" that can be used to create many objects of one type.

The standard way to create an "object type" is to use an **object constructor function**:

```
function Person(first, last, age, eye) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eye;
}
let myFather = new Person("John", "Doe", 50, "blue");
let myMother = new Person("Sally", "Rally", 48, "green");
```

The constructor function is the **prototype** for your person objects.

Objects

All JavaScript objects inherit the properties and methods from their prototype.

Objects created using an object literal, or with new Object(), inherit from a prototype called **Object.prototype**.

Objects created with new Date() inherit the Date.prototype.

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

All JavaScript objects (Date, Array, RegExp, Function, ...) inherit from the Object.prototype.

Objects

In JavaScript, the thing called **this**, is the object that "owns" the JavaScript code.

The value of **this**, when used in a function, is the object that "owns" the function.

The value of **this**, when used in an object, is the object itself.

The **this** keyword in an object constructor does not have a value. It is only a substitute for the new object.

The value of **this** will become the new object when the constructor is used to create an object.

Objects

ES6 introduced more intuitive, OOP-style **classes**:

```
class Shape {
    constructor (id, x, y) {
        this.id = id;
        this.move(x, y);
    }
    move (x, y) {
        this.x = x;
        this.y = y;
    }
}
```

The constructor method is called automatically when the object is initialized. If you do not have a constructor method, JavaScript will add an *invisible and empty* constructor method.

Objects

Classes may have **static members** (properties and methods), which are called without instantiating their class and **cannot** be called through a class instance.

```
class Shape {
    constructor (id, x, y) {
        this.id = id;
        this.move(x, y);
    }
    static displayName = "Shape";
    move (x, y) {
        this.x = x;
        this.y = y;
    }
}

s1 = new Shape("s1", 5, 5);

console.log(Shape.displayName);
```

Objects

Class **inheritance** is an important feature:

```
class Rectangle extends Shape {
    constructor (id, x, y, width, height) {
        super(id, x, y);
        this.width  = width;
        this.height = height;
    }
}

class Circle extends Shape {
    constructor (id, x, y, radius) {
        super(id, x, y);
        this.radius = radius;
    }
}
```

The super() method refers to the parent class. By calling the super() method in the constructor method, we call the parent's constructor method and get access to the parent's properties and methods.

Objects

You must declare a class before using it!

```
//You cannot use the class yet.
//mycar = new Car("Ford")
//This would raise an error.

class Car {
  constructor(brand) {
    this.carname = brand;
  }
}

//Now you can use the class:
mycar = new Car("Ford")
```

Objects

JavaScript has built-in constructors for native objects:

```
let x1 = new Object();     // A new Object object
let x2 = new String();     // A new String object
let x3 = new Number();     // A new Number object
let x4 = new Boolean()     // A new Boolean object
let x5 = new Array();      // A new Array object
let x6 = new RegExp();     // A new RegExp object
let x7 = new Function();   // A new Function object
let x8 = new Date();       // A new Date object
```

The Math() object is not in the list. **Math is a global object.** The new keyword cannot be used on Math.

```
Math.random();        // returns a random number

Math.min(0, 150, 30, 20, -8, -200);       // returns -200

Math.PI           // returns PI
```

Objects

But there is no reason to use new Array(). Use array literals instead: []
And there is no reason to use new RegExp(). Use pattern literals instead: /()/
And there is no reason to use new Function(). Use function expressions instead: function () {}.
And there is no reason to use new Object(). Use object literals instead: {}

```
let x1 = {};              // new object
let x2 = "";              // new primitive string
let x3 = 0;               // new primitive number
let x4 = false;           // new primitive boolean
let x5 = [];              // new array object
let x6 = /()/             // new regexp object
let x7 = function(){};    // new function object
```

Objects

Objects are addressed by reference, not by value.

If y is an object, the following statement will not create a copy of y:

```
let x = y;  // This will not create a copy of y.
```

The object x is not a **copy** of y. It **is** y. Both x and y points to the same object. Any changes to y will also change x, because x and y are the same object.

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

let x = person;
x.age = 10;           // This will change both x.age and person.age
```

Objects

You can add new properties to an existing object by simply giving it a value.

Assume that the person object already exists - you can then give it new properties:

```
person.nationality = "English";
```

The **delete** keyword deletes a property from an object:

```
let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
delete person.age;   // or delete person["age"];
```

Objects

The JavaScript prototype property allows you to add new properties and methods to an existing prototype:

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
Person.prototype.nationality = "English";
Person.prototype.fullName = function() {
    return this.firstName + " " + this.lastName;
};
```

Similarly, you can delete prototype properties and methods.

Adding/deleting a prototype property or method will affect all objects inherited from the prototype.

Objects

You cannot add a new property to a class "on the fly":

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}

Car.nationality = "English";

myCar = new Car("Ford", 2019);
document.getElementById("demo").innerHTML = myCar.nationality;
```

---> undefined

Objects

You can loop through all properties of an object using the **for..in** statement.

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
let txt = "";
let person = {fname:"John", lname:"Doe", age:25};
let x;
for (x in person) {
    txt += person[x] + " ";
}
document.getElementById("demo").innerHTML = txt;
</script>

</body>
</html>
```

Objects

The syntax

```
with (object) statement
```

uses `object` as the default prefix within `statement`.

The following code fragments are equivalent:

```
with (document.myForm) {
    result.value = compute(myInput.value);
}

document.myForm.result.value = compute(document.myForm.myInput.value);
```

Arrays

JavaScript arrays are used to store multiple values in a single variable.

```
<p id="demo"></p>

<script>
let cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>
```

The first line (in the script) creates an array named cars.

The second line "finds" the element with id="demo", and displays the array in the "innerHTML" property.

Arrays

Arrays are a special kind of objects, with numbered indexes instead of named indexes.

You refer to an array element by referring to the **index number**.

```
let name = cars[0];

cars[0] = "Opel";
```

It is possible to have variables of different types in the same array.

It is also possible to have objects, functions and arrays within the same array.

```
myArray[0] = Date.now;
myArray[1] = myFunction;
myArray[2] = myCars;
```

Arrays

A common question is: How do I know if a variable is an array?

The problem is that the JavaScript operator **typeof** returns "object":

```
let fruits = ["Banana", "Orange", "Apple", "Mango"];

typeof fruits; // typeof returns object
```

To solve this problem, use the following approach:

```
function isArray(myArray) {
    return myArray.constructor.toString().indexOf("Array") > -1;
}
```

The `isArray()` function returns true if the object prototype of the argument is "[object array]".

Arrays

The real strength of JavaScript arrays are the built-in array properties and methods.

```
let x = cars.length; // The length property returns the number of
elements in cars

let y = cars.sort(); // The sort() method sort cars in alphabetical
order

let fruits = ["Banana", "Orange", "Apple", "Mango"];
let x = fruits.pop(); // the value of x is "Mango"

let fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Lemon"); // adds a new element (Lemon) to fruits

let fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[fruits.length] = "Lemon"; // adds a new element (Lemon) to
fruits
```

Complete reference:
http://www.w3schools.com/jsref/jsref_obj_array.asp

Arrays

By default, the sort() function sorts values as **strings**.
This works well for strings ("Apple" comes before "Banana").

However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".
Because of this, the sort() method will produce incorrect result when sorting numbers.

You can fix this by providing a **compare function**:

```
let points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a-b});
```

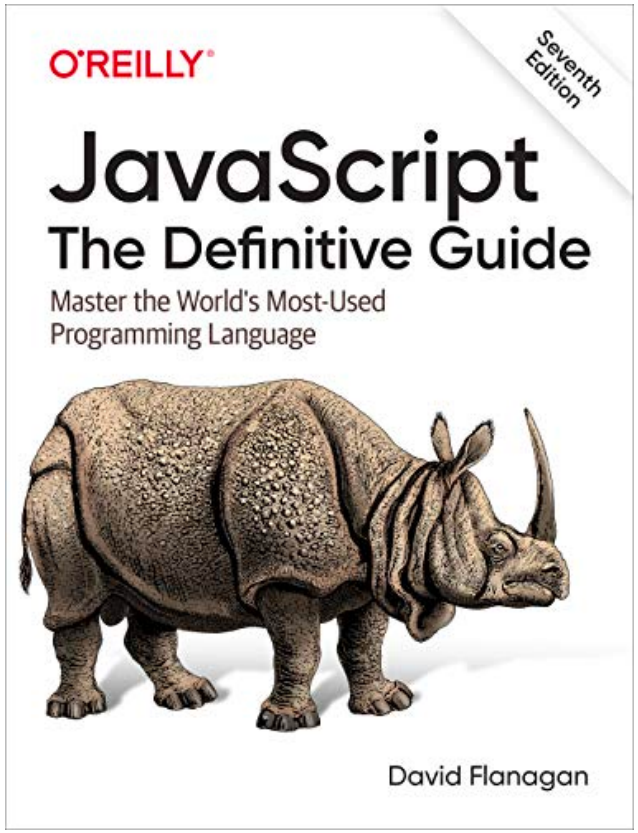
When comparing 40 and 100, the sort() method calls the compare function(40,100). The function calculates 40-100, and returns -60 (a negative value). The sort function will sort 40 as a value lower than 100. And so on..

Arrays

```
<!DOCTYPE html>
<html>
<body>

<p>Click the button to sort the array in ascending order.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
let points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points;

function myFunction() {
    points.sort(function(a, b){return a-b});
    document.getElementById("demo").innerHTML = points;
}
</script>

</body>
</html>
```

References

D. Flanagan
JavaScript - The Definitive Guide
ed. O'Reilly, 2020



http://www.w3schools.com/js/default.asp

http://es6-features.org/