JavaScript

Francesco Marcelloni

and Alessio Vecchio

Dipartimento di Ingegneria dell'Informazione
Università di Pisa
ITALY

Where to insert the JavaScript Code

Where to put the JavaScript? Head solution

- When in the head, JavaScripts in a page will be executed immediately while the page loads into the browser.
- Functions are executed when called

```
<html>
<head>
<script>
function message() {
  alert("This alert box was called with the onload event");
}
</script>
</head>
<body onload="message()">
</body>
</html>
```

Where to put the JavaScript? Body Solution (not recommended)

- You can place an unlimited number of scripts in your document, so you can have scripts in both the body and the head section.

```
<html>
<head>
</head>
<body>
<script>
document.write("This message is written by JavaScript");
</script>
</body>
</html>
```

Where to Put the JavaScript? External File

- If you want to run the same JavaScript on several pages, without having to write the same script on every page, you can write a JavaScript in an external file.
- Save the external JavaScript file with a .js file extension.
- Note: The external script cannot contain any HTML tags (in particular, <script></script> tags)

To use the external script, point to the .js file in the "src" attribute of the <script> tag:

```
<html>
<head>
  <script src=”myscript.js">
  </script>
</head>
  <body>
  </body>
</html>
```

When is a script executed?

- Global code (not in the body of functions):
  - is executed when it is met during the rendering of the page.
  - The global code can be in the HTML page or in an external file.
- Code in functions
  - is executed only if the function is called
- Events
  - The code of event handlers is executed when the event is fired

When is a script executed?

Modern browsers:

- once they encounter a JavaScript file they pause the rendering of HTML
- run though the entire JavaScript file before they resume the HTML rendering
- in this example, **document.write** does run first, but we do not see the result before execution is completed
- the alert dialog pauses that processing.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
<title>Execution order</title>
<script>
```

When is a script executed?

```
  window.alert("Not in function");
  document.write("<h1> This is a heading. </h1>");
  document.write("<p> This is a paragraph. </p>");
  document.write("<p> This is another paragraph. </p>");
  function loading() { window.alert("onLoad event");}
</script>
</head>
<body onload="loading()" >
<p>Text in the body<\p>
<script>
document.write("<p>Paragraph written in the body.</p>");
window.alert("Code in Body"); </script>
</body>
</html>
```

let vs var

```
let x = 10;

if (x === 10) {
  let x = 20;

  console.log(x);
  // prints 20
}

console.log(x);
// prints 10
```

```
function varExample() {
  var x = 10;
  {
    var x = 20;  // it is the same variable
    console.log(x);  // prints 20
  }
  console.log(x);  // prints 20
}

function letExample() {
  let x = 10;
  {
    let x = 20;  //it is a different variable
    console.log(x);  // prints 20
  }
  console.log(x);  // prints 10
}
```

the scope of a **var** is the whole function where it is declared

let vs var

- When used <u>outside functions</u>, var creates a property in the global object

```
var a = 'ABC';
let b = 'XYZ';
console.log(this.a); // "ABC"
console.log(this.b); // undefined
```

```
if (x) {
  let v;
  let v; // error, the same
         // variable defined twice
}
```

```
var a = 1;
var b = 2;
if (a === 1) {
  var a = 10; // the scope is global because of the other a
  let b = 20; // the scope is this block
  console.log(a); // prints 10
  console.log(b); // prints 20
}
console.log(a); // prints 10
console.log(b); // prints 2
```

const

- scope works similarly to let, must be initialized

```
const PI = 3.14;

// this will throw an error - Uncaught
// TypeError: Assignment to constant variable.
PI = 5;

console.log('The value of PI is ' + PI);

// trying to redeclare a constant throws an error
// Uncaught SyntaxError: Identifier 'PI' has already
// been declared
const PI = 9;

// Error: the name PI is reserved
var PI = 20;

// this throws an error since it is already defined
let PI = 20;

// Error, must be initialized, it is a const
const DIM;
```

Var (bad behaviors)

- Variables can be initialized

```
var x=5;
var carname="Volvo";
```

- If you assign values to variables that have not been declared yet, the variables will automatically be declared.
  - and they will be global variables... (error-prone)
- If you redeclare a JavaScript variable, it will not lose its original value.

```
var x=5;
var x;
```

- NOTE: the variable x will still have the value of 5. The value of x is not reset when you redeclare it.

Use of undeclared variables

- Esempio relativo a all'uso (**sbagliato**) di variabili non dichiarate.
- Vengono dichiarate automaticamente e sono global.

```
<!DOCTYPE html>
<html>
<head>
<script>
var abc = 10;
if (abc===10) {
  def = 20;
}
console.log(window.abc); // Stampa 10
console.log(window.def); // Stampa 20
// def automaticamente creata e globale
</script>
</head>
<body>
Esempio relativo a all'uso (sbagliato) di variabili non dichiarate.
Vengono dichiarate automaticamente e sono global.
</body>
</html>
```

Var Scope

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
<title>Scope</title>
<script>
var cc = 0 ;          //global
var dd = scr();     // global
document.writeln("global: " + cc); // print value of cc
document.writeln("local: " + dd);  // print value of dd
function scr() {
    var cc = 3;     //local variable hides the global variable cc
    // without var, it would be an assignment to global variable cc
    return cc;
}
</script>
</head>

<body>
<p>Scope</p>
</body>
</html>
```

String Values

- String

  contains zero or more characters enclosed in single or double quotes

  - NOTE: the empty string is distinct from the null value
  - The backslash (\) is used to insert apostrophes, new lines, quotes, and other special characters into a text string

    - \'
    - \"
    - \\

Comparison Operators

- Given x=5, the table below explains the comparison operators

Operator	Description	Example
==	is equal to	x==8 is false x=='5' is true
===	is exactly equal to (value and type)	x===5 is true x==='5' is false
!=	is not equal (it attempts conversion)	x!=8 is true x!=5 is false
!==	is not equal and/or not of the same type	x!=='5' is true
>	is greater than	x>8 is false
<	is less than	x<8 is true
>=	is greater than or equal to	x>=8 is false
<=	is less than or equal to	x<=8 is true

Comparison Operators

- Comparison operators
  - If either or both values are **NaN**, then they are **not** equal.
  - Objects, arrays, and functions are compared by reference. This means that two variables are equal only if they refer to the same object.
  - If both are **null**, or both **undefined**, they are equal.
  - If one value is **null** and one **undefined**, they are equal.

- Two separate arrays are never equal by the definition of the == operator, even if they contain identical elements.

Bitwise Operators

Operator	Description	Example
&	and	a & b
|	or	a | b
!	xor	a ^ b
~	not	~a
<<	Left shift	a<<b
>>	Sign-propagating right shift	a>>b
>>>	Zero-fill right shift	a>>>b

JavaScript Operators

- If the types of the two values differ, attempt to convert them into the same type so they can be compared:
  - If one value is a number and the other is a string
    - convert the string to a number and try the comparison again, using the converted value.
  - If either value is true
    - convert it to 1 and try the comparison again.
  - If either value is false
    - convert it to 0 and try the comparison again.
  - If one value is an object and the other is a number or string
    - convert the object to a primitive value by either its *toString()* method or its *valueOf()* method. Native JavaScript classes attempt *valueOf()* conversions before *toString()* conversion.
  - Any other combinations of types are not equal.

typeof operator

- **typeof** operator

  Two ways:

  1. typeof operand
  2. typeof (operand)

- The typeof operator returns a string indicating the type of the operand. The parentheses are optional.

```
let num1 = 3;
let num2 = 3.0;
let b=true;
let shape="round";
typeof num1;        // returns 'number'
typeof num2;        // returns 'number'
typeof b;           // returns 'boolean'
typeof shape;       // returns 'string'
```

void operator

```
> let x;
<· undefined
> x = 10;
<· 10
> void(x=20);
<· undefined
>
```

- void operator

  Two ways:

  1. void (expression)
  2. void expression

- The void operator specifies a JavaScript expression to be evaluated without returning a value. The parentheses surrounding the expression are optional, but it is good style to use them.

- The following code creates a hypertext link that changes the background color

```
<a
  href="javascript:void(document.body.style.backgroundColor='red')"
  >
Change background to red</a>
```

Functions

- The arguments of a function are maintained in the array "arguments".
- Within a function, you can address the parameters passed to it by:

  arguments[i]

  functionName.arguments[i]

  where i is the ordinal number of the argument, starting at zero.

  arguments[0] -> the first argument passed to a function.

- The total number of arguments is indicated by arguments.length.

Functions

- Using the arguments array, you can call a function with more arguments than it is formally declared to accept.
  - This is often useful if you do not know in advance how many arguments will be passed to the function.
  - You can use arguments.length to determine the number of arguments actually passed to the function, and then treat each argument using the arguments array.

Functions

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Example</title>
<script>
function myConcat(separator) {
  let result=""; // initialize list
  // iterate through arguments
  for (let i=1; i<arguments.length; i++) {
    result += arguments[i] + separator;
  }
  result += "<br>";
  return result
}
</script>
</head>

<body>
<script>
// returns "red, orange, blue, "
document.write(myConcat(", ","red","orange","blue"));
// returns "elephant; giraffe; lion; cheetah;"
document.write(myConcat("; ","elephant","giraffe","lion", "cheetah"));
// returns "sage. basil. oregano. pepper. parsley. "
document.write(myConcat(". ","sage","basil","oregano", "pepper", "parsley"));
</script>
</body>
</html>
```

Predefined functions

- JavaScript has several top-level predefined functions:
  - eval(string) - Evaluates a string and executes it as if it was script code
    eval("x=10;y=20;document.write(x*y)"); //200

  - isFinite - Determines whether a value is a finite, legal number
    document.write(isFinite(123)+ "<br>"); //true
    document.write(isFinite("2005/12/12")+ "<br>"); //false

  - isNaN -The isNaN() function determines whether a value is an illegal number (Not-a-Number).
    This function returns true if the value is NaN, and false if not.
    - document.write(isNaN(123)+ "<br>"); //false
    - document.write(isNaN("2005/12/12")+ "<br>"); //true

Predefined functions

- parseInt(string,radix)

  Parses a string and returns an integer of the specified radix (base).

  radix - a number that represents the numeral system to be used

- parseFloat(string)

  Parse a string and returns a float number.

  If the first character cannot be converted to a number, the two functions return NaN.

- Number(object) and String(object)

  Converts the object argument to a number or to a string that represent the object's value.

  If the value cannot be converted to a legal number, NaN is returned.

Predefined functions

```
<script>
eval("x=10;y=20;document.write(x*y)");
document.write("<br>" + isFinite(123)+ "<br>");
document.write(isFinite("2005/12/12")+ "<br>");
document.write(isNaN(123)+ "<br>");
document.write(isNaN("2005/12/12")+ "<br>");
document.write(parseInt("His age is 40 years")+ "<br>");
document.write(parseInt("40 years")+ "<br>");
</script>
```

Alert Box

- An alert box is often used if you want to make sure information comes through to the user.
- When an alert box pops up, the user will have to click "OK" to proceed.

```
alert("sometext");
```

Alert Box

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Example</title>
<script>
function showAlert()
{ alert("I am an alert box!");}
</script>
</head>

<body>
<p><input type="button" onclick="showAlert()" value="Show an alert
box">
</p>
</body>
</html>
```

Confirm Box

- A confirm box is often used if you want the user to verify or accept something.
- When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed.
- If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.

```
confirm("sometext");
```

Confirm Box

```
<head>
<meta charset="utf-8">
<title>Example</title>
<script>
function showConfirm() {
  let r = confirm("Press a button!");
  if (r) {
    alert("You pressed OK!");
  } else {
    alert("You pressed Cancel!");
  }
}
</script>
</head>
<body>
<p><input type="button" onclick="showConfirm()" value="Show confirm box">
</p>
</body>
```

Prompt Box

- A prompt box is often used if you want the user to input a value before entering a page.
- When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value.
- If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

```
prompt("sometext","default value");
```

Prompt Box

```
<head>
<meta charset="utf-8">
<title>Example</title>
<script>
function showPrompt() {
  let name = prompt("Please enter your name", "Harry Potter");
  if (name != null && name != "") {
    document.write("Hello " + name + "! How are you today?");
  }
}
</script>
</head>
<body>
<p>
<input type="button" onclick="showPrompt()" value="Show prompt box">
</p>
</body>
```

try...catch statement

```
try
 {
 //Run some code here
 }
catch(err if expression)
 {
 //Handle errors here
 }
```

- The try...catch statement allows you to test a block of code for errors.
- The try block contains the code to be run, and the catch block contains the code to be executed if an error occurs.
- *err* is initialized with the exception object
- *expression* is a test expression

try…catch statement

```
<script>
function message() {
  try {
    addlert("Welcome guest!");
  } catch(err) {
    let txt="There was an error on this page.\n\n";
    txt += err;
    txt += "\n\nClick OK to continue viewing this page,\n";
    txt += "or Cancel to return to the home page.\n";
    if(!confirm(txt)) {
      document.location.href= "http://www.example.com";
    }
  }
}
</script>
</head>
<body>
<input type="button" value="View message" onclick="message()">
</body>
```

Throw statement

- The throw statement allows you to create an exception.

  throw(exception)

- The exception can be a string, integer, Boolean or an object.

Throw statement

```
<body>
<script>
let x = prompt("Enter a number between 0 and 10:","");
try {
  if(x>10) { throw "Err1"; }
  else if(x<0) { throw "Err2"; }
  else if(isNaN(x)) { throw "Err3"; }
  // Other statements
  console.log("Here!");
} catch(er) {
  if(er=="Err1") {
    alert("Error! The value is too high");
  }
  if(er=="Err2") {
    alert("Error! The value is too low");
  }
  if(er=="Err3") {
    alert("Error! The value is not a number");
  }
}
</script>
</body>
```

JavaScript Objects

Francesco Marcelloni

and Alessio Vecchio

Dipartimento di Ingegneria dell'Informazione

Università di Pisa

ITALY

Creating new objects

- Two ways for creating objects
  - use an **object initializer**
  - create **a constructor function** and then instantiate an object using that function and the **new** operator
- The first way is useful when you need to create a unique instance of the object; otherwise use the second way.
- Recent versions of JavaScript: objects can be created as instances of **classes**.

Object Initializer

- Object initializer

  objectName = {property1:value1,… , propertyN:valueN}

  objectName: name of the new object

  propertyX: identifier (either a name, a number, or a string literal),
  valueX: expression whose value is assigned to the *propertyX*.

Object initializer: examples

```
const stud1 = {nome: 'Mario',
               cognome: 'Rossi',
               matricola: 1234};
console.log(stud1);

const o1 = {};
console.log(o1);

const o2 = {
  nome_composto: 'questo va bene',
  //altro-nome-composto: 'questo non va bene'
};
console.log(o2);
```

```
{ nome: 'Mario', cognome: 'Rossi', matricola: 1234 }
{}
{ nome_composto: 'questo va bene' }
```

Object initializer: examples

- Whe using variables, if the property name is omitted then the variable name is used:

```
const c1 = 22;
let v1 = "ABC";

const o3 = {c1, v1};
console.log(o3);
const o3bis = {c1: c1, v1: v1};
console.log(o3bis);
```

```
{ c1: 22, v1: 'ABC' }
{ c1: 22, v1: 'ABC' }
```

Accessing properties

```
// getting setting
let p1 = {x: 0, y:0};
const p2 = {x: 1, y: 2};
console.log(p1.x);
console.log(p2.y);
p1.x = 10;
p2.y = 20;
console.log(p1);
console.log(p2);

// if a property does not exist, undefined is returned
console.log(p1.z);

// A new property is added in case of write
p2.z = 33;
console.log(p2);
```

- dot notation

```
0
2
{ x: 10, y: 0 }
{ x: 1, y: 20 }
undefined
{ x: 1, y: 20, z: 33 }
```

Object Initializer

- The property of an object can be an object
- Object literals can be passed as parameters

```
const cont1 = {tel: 555123456,
               email: 'mario.rossi@unipi.it'};
const s1 = {nome: "Mario",
            cognome: "Rossi",
            contatto: cont1};
console.log(s1);
function f(o) {
  console.log("Coordinates: " +
               o.x + ", " + o.y);
}
f(p1);
f({x: 100, y: 200});
```

```
{
  nome: 'Mario',
  cognome: 'Rossi',
  contatto: { tel: 555123456, email: 'mario.rossi@unipi.it' }
}
Coordinates: 10, 0
Coordinates: 100, 200
```

Objects: associative arrays notation

- There is another notation that can be used
- Properties are not limited anymore to identifier rules

```
const o4 = {
  'Una stringa e non un identificatore': 1234,
  'altra chiave': 'XYZ' };
console.log(o4);
// also mixed
const o5 = {
  'This is a string': 666,
  this_is_an_identifier: 999,
  k1: 'v1'
};
console.log(o5);
// To access:
o4['Una stringa e non un identificatore'] = 555;
o4['k1'] = 'new value';
console.log(o4);
```

```
{ 'Una stringa e non un identificatore': 1234, 'altra chiave': 'XYZ' }
{ 'This is a string': 666, this_is_an_identifier: 999, k1: 'v1' }
{
  'Una stringa e non un identificatore': 555,
  'altra chiave': 'XYZ',
  k1: 'new value'
}
```

Objects: associative arrays notation

- Now properties names can be dynamically generated

```
let a = "ABC";
let b = "DEF";
let c = Math.random();
const o6 = {
  [a+b]: "XYZ",
  [c]: Math.random()
}
console.log(o6);

const o7 = {};
for(let i=0; i<5; i++) {
  o7['k'+i]= 'v'+i;
}
console.log(o7);
```

```
{ ABCDEF: 'XYZ', '0.32245611236406035': 0.5164861542942805 }
{ k0: 'v0', k1: 'v1', k2: 'v2', k3: 'v3', k4: 'v4' }
```

for .. in

- The for...in statement loops through the enumerable properties of an object

*for (*variable *in* object*) {*

code to be executed

}

```
const studente = {nome: "Mario",
                  cognome: "Rossi",
                  matricola: 1234,
                  voti: [23, 28, 29]};
// Può anche essere const p, se non varia nel corpo del ciclo
for(let p in studente) {
  console.log(p + ' di tipo ' + typeof p);
  console.log('con valore di tipo ' + typeof studente[p]
            + ': ' + studente[p]);
}
```

nome di tipo string
con valore di tipo string: Mario
cognome di tipo string
con valore di tipo string: Rossi
matricola di tipo string
con valore di tipo number: 1234
voti di tipo string
con valore di tipo object: 23,28,29

Object Initializer

- Object initializer (example)

```
<head>
<meta charset="utf-8">
<title>Example</title>
<script>
let guy = { name: "Mario",
        age: 40,
        country: "Italy",
        auto: { trademark: "Ferrari", colour: "rosso" }}
</script>
</head>
<body>
<pre>
<script>
for (let a in guy)
 if (typeof(guy[a])=="object")
  for (let i in guy[a])
   document.writeln(a + '.' + i + ':' + guy[a][i]);
 else document.writeln(a + ':' + guy[a]);
</script>
</pre>
</body>
```

- Associative array notation (example)

```
<script>
function showProperties(obj, objectName) {
  let result = "";
  // i assumes as values all the property names
  for (let i in obj)
    result += objectName + "." + i + " = " + obj[i] + "\n";
  return result;
}
</script>
</head>
<body>
<pre>
<script>
let myCar = {
  make: "Fiat",
  model: "500",
  year: 2020
}
document.write(showProperties(myCar,"myCar"));
</script>
</pre>
</body>
```

Constructor Function

- **Constructor Function**

  1. Define the object type by writing a constructor function.

  2. Create an instance of the object with **new**.

To define an object type, create a function for the object type that specifies its name, properties, and methods.

```
<script>
function Bike(make, model, year) {
  this.make = make
  this.model = model
  this.year = year
}

function displayBike(bike) {
  for (let a in bike)
    document.writeln(a + ':' + bike[a]);
  document.writeln();
}
</script>
```

this keyword!

```
<body>
<pre>
<script>
  let mybike = new Bike("Giant", "Reign", 2013);
  let mariobike = new Bike("Specialized", "Enduro", 2018);
  let luigibike = new Bike("Transition", "Scout", 2020);
  displayBike(mybike);
  displayBike(mariobike);
  displayBike(luigibike);
</script>
</pre>
</body>
```

Constructor Function

```
<head>
<meta charset="utf-8">
<title>Example</title>
<script>
function Car(make, model, year, owner) {
 this.make = make;
 this.model = model;
 this.year = year;
 this.owner = owner;
}
function Person(name, age, sex) {
 this.name = name;
 this.age = age;
 this.sex = sex;
}
function displayCar(car) {
 for (let a in car)
  if (typeof(car[a])=="object")
   for (let i in car[a])
    document.writeln(a + '.' + i + ':' + car[a][i]);
  else document.writeln(a + ':' + car[a]);
 document.writeln();
}
</script>
</head>
```

- Constructor Function

  An object can have a property that is itself another object.

```
<body>
<pre>
<script>
let owner = new Person("Gino Bianchi",45,"M");
let mycar = new Car("Fiat", "Topolino", 1956, owner);
let luigicar = new Car("Lancia", "Stratos", 1977);
let mariocar = new Car("Ford", "Model T", 1910);
displayCar(mycar);
displayCar(luigicar);
displayCar(mariocar);
</script>
</pre>
</body>
```

new operator

- **new** operator

  You can use the new operator to create an instance of a user-defined object type or of one of the predefined object types such as **Array**, **Boolean**, **Date**, **Function**, **Image**, **Number**, **Object**, **Option**, **RegExp**, or **String**.

  objectName = new ObjectType(param1 [,param2] ...[,paramN]);

Adding a new property

- Properties can be added to an object at run time.
- To add a property to a specific object you have to assign a value to the object:

```
car1.enginepower = 100
```

- Note: only the *car1* object will have the property *enginepower*

Adding a new property

- You can add a property to a previously defined object type by using the **prototype** property.
- This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object.
- The following code adds a *color* property to all objects of type *Car*, and then assigns a value to the *color* property of the object *car1*:

```
Car.prototype.color = null;
car1.color="black";
```

Adding a new property

```
<head>
<meta charset="utf-8">
<title>Example</title>
<script>
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
function displayCar(car) {
  for (let a in car)
    if (typeof(car[a])=="object")
      for (let i in car[a])
        document.writeln(a + '.' + i + ':' + car[a][i]);
    else document.writeln(a + ':' + car[a]);
  document.writeln();
}
</script>
</head>
```

Adding a new property

```
<body>
<pre>
<script>
let owner = new Person("Mario Rossi",45,"M");
let mycar1 = new Car("Eagle", "Talon TSi", 1993, owner);
displayCar(mycar1);

Car.prototype.color = "red";

let mycar2 = new Car("Nissan", "300ZX", 1992, owner);
let mycar3 = new Car("Mazda", "Miata", 1990, owner);
mycar3.color = "black";

displayCar(mycar1);
displayCar(mycar2);
displayCar(mycar3);
</script>
</pre>
</body>
```

Defining methods

- The following syntax associates a **function** with an existing object:

  object.methodname = function_name

- You can then call the method in the context of the object as follows:

  object.methodname(params);

- You can define methods for an object type by including a method definition in the object constructor function.

Defining methods

```
<head>
<meta charset="utf-8">
<title>Example</title>
<script>
function Car(make, model, year, owner) {
 this.make = make
 this.model = model
 this.year = year
 this.owner = owner
 this.displayCar = displayCar;
}
function Person(name, age, sex) {
 this.name = name
 this.age = age
 this.sex = sex
}
function displayCar(){
 for (let a in this)
  if (typeof(this[a])=="object")
   for (let i in this[a])
    document.writeln(a + '.' + i + ':' + this[a][i]);
  else
   if (typeof(this[a])!="function")
    document.writeln(a + ':' + this[a]);
 document.writeln();
}
</script>
</head>
```

Defining methods

```
<body>
<pre>
<script>
let owner = new Person("Frankie Black",45,"M")
let mycar1 = new Car("Eagle", "Talon TSi", 1993, owner)
mycar1.displayCar();
Car.prototype.color = "red"
let mycar2 = new Car("Nissan", "300ZX", 1992,owner)
let mycar3 = new Car("Mazda", "Miata", 1990,owner)
mycar3.color = "black"
mycar1.displayCar();
mycar2.displayCar();
mycar3.displayCar();
</script>
</pre>
</body>
```

Defining methods

Alternatively:

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}

Car.prototype.displayCar = function () {
  for (let a in this)
    if (typeof(this[a])=="object")
      for (let i in this[a])
                        document.writeln(a + '.'    + i + ':' + this[a][i]);
    else if (typeof(this[a])!="function")
                        document.writeln(a + ':' + this[a]);
  document.writeln();
}
```

Objects: methods

- Another notation

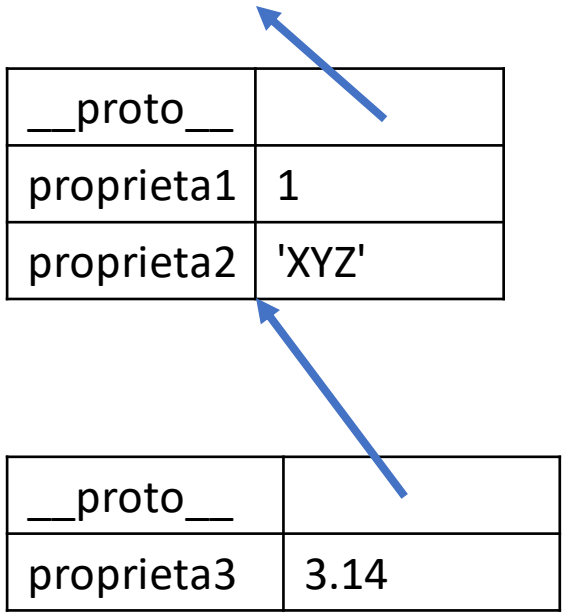
```
stud2.migliore = function () {
	let m = 0;
	for(let i=0; i<this.voti.length; i++)
		if (this.voti[i]>m)
	m = this.voti[i];
	return m;
}

console.log(stud2.migliore());
```

Inheritance

Prototype-based inheritance

- JavaScript inheritance is based on prototype objects

```
const sopra = {
  proprieta1: 1,
  proprieta2: 'XYZ'
};

const sotto = {
  __proto__: sopra,
  proprieta3: 3.14
};

console.log(sotto);
console.log(sotto.proprieta3);
console.log(sotto.proprieta1);
console.log(sotto.proprieta2);
sotto.proprieta1 = 100;
console.log(sotto);
console.log(sopra);
```




```
{ proprieta3: 3.14 }
3.14
1
XYZ
{ proprieta3: 3.14, proprieta1: 100 }
{ proprieta1: 1, proprieta2: 'XYZ' }
```

Prototype-based inheritance

- There can be a chain of prototype objects
  - sopra can have its own prototype
- Each object has its **own** properties and **inherited** properties
  - own properties of sopra: proprieta1 and proprieta2
  - own properties of sotto: proprieta3
  - inherited properties of sotto: proprieta1 and proprieta2
  - some methods consider just own properties, other consider also the inherited ones
- When reading the property of an object
  - if the object does not have such a property the prototype chain is followed looking for such a property
  - if not found: undefined
- When writing a value in an inherited property, the object is given such a property with the new value
  - the new property hides the corresponding one in the prototype object
  - the value of the property in the prototype remains unchanged

Prototype-based inheritance

```
const Persona = {
 stampaGeneralita() {
  console.log(this.nome + " " + this.cognome);
 }
};
const studenteA = {
 __proto__: Persona,
 nome: "Sara",
 cognome: "Verdi",
 media: 23.5,
 stampaStudente() {
  this.stampaGeneralita();
  console.log("media: " + this.media);
 }
}
const docenteX = {
 __proto__: Persona,
 nome: "Paolo",
 cognome: "Bianchi",
 email: "paolo.bianchi@unipi.it",
 stampaDocente() {
  this.stampaGeneralita();
  console.log("email: " + this.email);
 }
}
studenteA.stampaStudente();
docenteX.stampaDocente();
```

- It works also for methods

Sara Verdi
media: 23.5
Paolo Bianchi
email: paolo.bianchi@unipi.it

__proto__	...
stampaGeneralita	function () {...}

__proto__	
nome	Paolo
cognome	Bianchi
email	paolo.bianchi@unipi.it
stampaDocente	function () {...}

__proto__	
nome	Sara
cognome	Verdi
media	23.5
stampaStudente	function () {...}

Constructor functions

- The __*proto*__ property is generally not set by hand
- Constructor functions allows the programmer to create instances of classes using the **new** operator
- The function automatically set the __*proto*__ property for all created object so that they all inherit from a common prototype object
- The prototype object contains the method inherited by all class instances
- In JavaScript, two objects are instances of the same class if they have the same prototype object

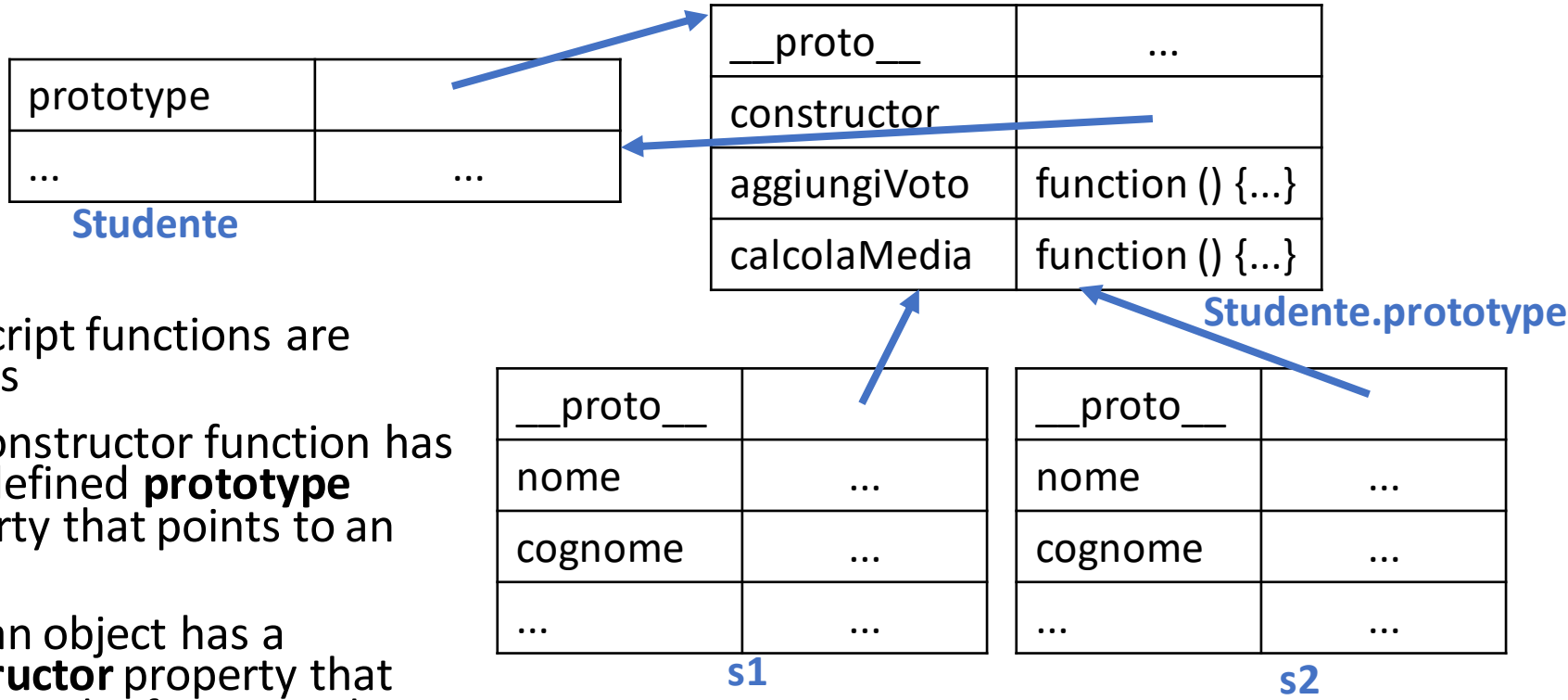
Constructor functions

```
function Studente(n, c, m) {
  this.nome = n;
  this.cognome = c;
  this.matricola = m;
  this.voti = [];
}

Studente.prototype.aggiungiVoto = function(v) {
  this.voti[this.voti.length] = v;
}

Studente.prototype.calcolaMedia = function() {
  let s = 0;
  for(let i=0; i<this.voti.length; i++)
    s += this.voti[i];
  return s / this.voti.length;
}
```

Constructor functions

```
let s1 = new Studente("Mario", "Rossi", 1234);
s1.aggiungiVoto(20);
s1.aggiungiVoto(22);
let media = s1.calcolaMedia();
console.log(media);
const s2 = new Studente("Sara", "Verdi", 8899);
s2.aggiungiVoto(30);
console.log(s2.calcolaMedia());
```

1. creates a new object and sets its **__proto__** property to the **Studente.prototype** object
2. the body of the **Studente()** function is executed, **this** is the newly created object
3. the **Studente.prototyp**e object has been equipped the **aggiungiVoto()** and **aggiungiMedia()** methods, that are hence inherited



- JavaScript functions are objects
- The constructor function has a predefined **prototype** property that points to an object
- Such an object has a **constructor** property that points to the function object
- The **instanceof** operator:

  s instanceof Studente

  returns true if **Studente.prototype** is in the inheritance chain for object **s**

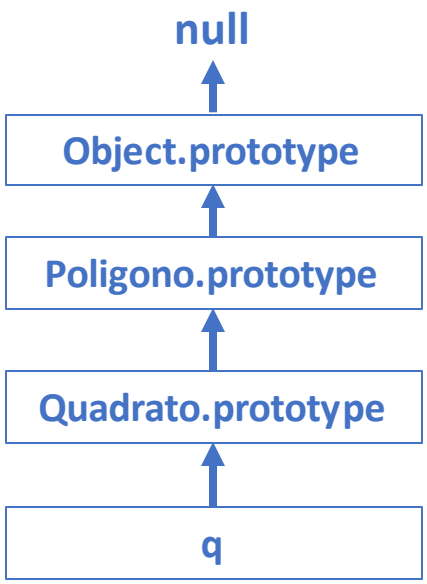
```
console.log(s1.__proto__ === s2.__proto__);
// true
console.log(s1.__proto__ === Studente.prototype);
// true
console.log(Studente.prototype.constructor === Studente);
// true
```

Setting/getting prototype

- *__proto__* is also indicated as *[[Prototype]]*
  - more precisely: *__proto__* is the accessor of *[[Prototype]]*
- Direct manipulation should be avoided, use instead
  - *Object.getPrototypeOf(x)* to obtain the *__proto__* of *x*
  - *Object.setPrototypeOf(x, p)* sets the *__proto__* of *x* to *p*
- The *Object.create(p)* method creates a new object that inherits from *p*

```
const myp = {a: 1, b: 2};
const mysub = Object.create(myp);
console.log(mysub.a);
// Prints 1
const newp = {c: 3};
Object.setPrototypeOf(mysub, newp);
console.log(mysub.a);
// Prints undefined
console.log(mysub.c);
// Prints 3
```

Inheritance




```
function Poligono(n, l) {
  this.lunghezzaLato = l;
  this.numeroLati = n;
}

function Quadrato(l) {
  Poligono.call(this, 4, l);
}

Object.setPrototypeOf(Quadrato.prototype,
                      Poligono.prototype);

Poligono.prototype.perimetro = function () {
  return this.lunghezzaLato * this.numeroLati;
}

Quadrato.prototype.area = function () {
  return this.lunghezzaLato *
        this.lunghezzaLato;
}

const q = new Quadrato(10);
console.log(q.area());          // 100
console.log(q.perimetro());     // 40
```

Built-in Objects

Objects

- Predefined built-in objects

  Array, Boolean, Date, Function, Math, Number, RegExp, and String

Array

- An array is an ordered set of values that you refer to with an index
- Array elements may belong to **different** types
- The Array object has methods for manipulating arrays in various ways, such as joining, reversing, and sorting them
- Methods are inherited (*Array.prototype*)
- Each array has a property for determining the array length

Array

- Create an Array

  Three forms

  - let myCars = new Array(); // (add an optional integer
    myCars[0] = "Saab"; // argument to control array's size)
    myCars[1] = "Volvo";
    myCars[2] = "BMW";

  - let myCars = new Array("Saab", "Volvo", "BMW"); //condensed

  - let myCars = ["Saab", "Volvo", "BMW"]; //literal array
    - It is also possible to avoid to specify all the values of the elements of the array

      let myCars = ["Saab", ,"BMW"];

Array

- Access an Array

```
document.write(myCars[0]);
```

- Modify values in an Array

```
myCars[0] = "Ferrari";
```

- An array can dynamically be extended

```
myCars[6] = "Fiat";
```

The size of the array is extended to 7.

The elements with indexes 3, 4 and 5 are undefined.

Array

- The length property specifies the size of the array

```
let v = [10, 20, 30];
console.log(v.length);
v.length = 6;
console.log(v);
v[8] = 999;
console.log(v);
```

```
3
[ 10, 20, 30, <3 empty items> ]
[ 10, 20, 30, <5 empty items>, 999 ]
```

Array

Method	Description
concat()	Joins two or more arrays, and returns a copy of the joined arrays
	array.concat(array2, array3, ..., arrayX)
join()	Joins all elements of an array into a string
	array.join(separator)
	If the separator is omitted, the elements are separated with a comma
pop()	Removes the last element of an array, and returns that element
	array.pop()
push()	Adds new elements to the end of an array, and returns the new length
	array.push(element1, element2, ..., elementX)

Array

Method	Description
reverse()	Reverses the order of the elements in an array
	array.reverse()
shift()	Removes the first element of an array, and returns that element
	array.shift()
slice()	Selects a part of an array, and returns the new array
	array.slice(start, end)
	start Required. An integer that specifies where to start the selection. You can also use negative numbers to select from the end of an array
	end Optional. An integer that specifies where to end the selection. If omitted, slice() selects all elements from the start position and to the end of the array

Array

Method	Description
sort()	Sorts the elements of an array
	array.sort(sortfunc)
	sortfunc Optional. A function that defines the sort order Default: sorts the elements alphabetically and ascending. However, numbers will not be sorted correctly (40 comes before 5). To sort numbers, you must add a function that compare numbers
splice()	Adds/Removes elements from an array
	array.splice(index,howmany,element1,.....,elementX)
	index Required. An integer that specifies at what position to add/remove elements
	howmany Required. The number of elements to be removed. If set to 0, no elements will be removed
	element1, ..., elementX Optional. The new element(s) to be added to the array

Array

Method	Description
toString()	Converts an array to a string, and returns the result
	array.toString()
unshift()	Adds new elements to the beginning of an array, and returns the new length
	array.unshift(element1,element2, ..., elementX)
	element1,element2, ..., elementX Required. The element(s) to add to the beginning of the array
valueOf()	Returns the primitive values of an array
	array.valueOf()

Array

- Sort numbers

```
<script>
function compare(a, b) {
        return a - b;      //(numerically and ascending)
        //return b - a;   (numerically and descending)
}
let n = [10, 5, 40, 25, 100, 1];
document.write(n.sort(compare));
</script>
```

If compare(a, b) < 0, a is placed before b;

If compare(a, b) == 0, no change;

compare(a, b) > 0, b is placed before a;

Boolean

Used in two ways:

- **new Boolean(v)** creates a Boolean wrapper object, it is not a primitive value
- **Boolean(v)** convert *v* to a boolean primitive value

The second form much more common than the first one

new Boolean()

```
// Some wrapper objects
const b1 = new Boolean(0);
const b2 = new Boolean(null);
const b3 = new Boolean(false);
const b4 = new Boolean("ABC");
const b5 = new Boolean(39);

// valueOf() retrieves the primitive
// value (true/false) from the wrapper object
console.log(b1.valueOf());
console.log(b2.valueOf());
console.log(b3.valueOf());
console.log(b4.valueOf());
console.log(b5.valueOf());

// The type of b1, ..., b5 is object
console.log(typeof b1);

// A non null reference is considered true, also
// when it is a Boolean contaning a false value
if(b3) {
  console.log("I'm here: then");
} else {
  console.log("I'm here: else");
}
// Now the primitive value
if(b3.valueOf()) {
  console.log("I'm here: then");
} else {
  console.log("I'm here: else");
}
```

```
false
false
false
true
true
object
I'm here: then
I'm here: else
```

Boolean

- undefined is converted to false
- null is converted to false
- 0, -0, and NaN are converted to false; other numbers converted to true
- All objects converted to true
- Empty string converted to false

```
const cb1 = Boolean(undefined);
const cb2 = Boolean(null);
const cb3 = Boolean(0);
const cb4 = Boolean([]);
const cb5 = Boolean('');
const cb6 = Boolean({a: 1});
const cb7 = Boolean([1, 2, 3]);
const cb8 = Boolean('ABC');
const cb9 = Boolean(
        new Boolean(false));
console.log(cb1);
...
console.log(cb9);
```

```
false
false
false
true
false
true
true
true
true
```

Boolean

- Values that are converted to true are said *truthy*
- Values that are converted to false are said *falsy*
- Things can be weird…

```
// Empty array is truthy
let a = [];
if(a) {
  // all objects are truthy and an empty array is still an object
  console.log("I'm here: then");
} else {
  console.log("I'm here: else");
}
// but it is loosely equal to false
if(a == false) {
  // Multiple conversions
  // a is converted to a primitive using Array.prototype.toString()
  // the result is ""
  // When converting string and boolean both are converted to numbers
  // and they are both 0
  console.log("I'm here: then");
} else {
  console.log("I'm here: else");
}
```

```
I'm here: then
I'm here: then
```

Date

- JavaScript stores dates as the number of milliseconds since January 1, 1970, 00:00:00
- Four forms to create a Date object
  - new Date () *// Date object with current date and time*
  - new Date (milliseconds) *//milliseconds since 1970/01/01*
  - new Date (dateString) *//A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, xmas23 = new Date("December 25, 2023 19:15:00"). If you omit hours, minutes, or seconds, the value will be set to zero.*
  - new Date (year, month, day, hours, minutes, seconds, milliseconds) *//parameters are integer values*
- Most parameters above are optional. Not specifying them, causes 0 to be passed in.

Date

Method	Description
getDate()	Returns the day of the month (from 1-31)
getDay()	Returns the day of the week (from 0-6)
getFullYear()	Returns the year (four digits)
getHours()	Returns the hour (from 0-23)
getMilliseconds()	Returns the milliseconds (from 0-999)
getMinutes()	Returns the minutes (from 0-59)
getMonth()	Returns the month (from 0-11)
getSeconds()	Returns the seconds (from 0-59)
getTime()	Returns the number of milliseconds since midnight Jan 1, 1970

Date

Method	Description
parse()	Parses a date string and returns the number of milliseconds since midnight of January 1, 1970
	Date.parse(datestring)
	Datestring Required. A string representing a date
setDate()	Sets the day of the month (from 1-31)
setFullYear()	Sets the year (four digits)
setHours()	Sets the hour (from 0-23)
setMilliseconds()	Sets the milliseconds (from 0-999)
setMinutes()	Set the minutes (from 0-59)
setMonth()	Sets the month (from 0-11)
setSeconds()	Sets the seconds (from 0-59)
setTime()	Sets a date and time by adding or subtracting a specified number of milliseconds to/from midnight January 1, 1970

Date

Method	Description
toDateString()	Converts the date portion of a Date object into a readable string
toLocaleDateString()	Returns the date portion of a Date object as a string, using locale conventions
toLocaleTimeString()	Returns the time portion of a Date object as a string, using locale conventions
toLocaleString()	Converts a Date object to a string, using locale conventions
toString()	Converts a Date object to a string
toTimeString()	Converts the time portion of a Date object to a string
toUTCString()	Converts a Date object to a string, according to universal time
UTC()	Returns the number of milliseconds in a date string since midnight of January 1, 1970, according to universal time
valueOf()	Returns the primitive value of a Date object

Date

```
<script type="text/javascript">
        let d = new Date();
        document.write("Original form: ");
        document.write(d + "<br>");
        document.write("Formatted form: ");
        document.write(d.toLocaleDateString());
</script>
```

Output

Original form: Sun Oct 24 2021 23:18:24 GMT+0200 (Ora legale dell'Europa centrale)

Formatted form: 24/10/2021

Math

- The Math object allows performing mathematical tasks.
- The Math object includes several mathematical constants and methods.

- Math is not a constructor. All properties/methods of Math can be called by using Math as an object, without creating it.
- Example:

```
let pi_value = Math.PI;
let sqrt_value = Math.sqrt(16);
```

Math

Property	Description
E	Returns Euler's number
LN2	Returns the natural logarithm of 2
LN10	Returns the natural logarithm of 10
LOG2E	Returns the base-2 logarithm of E
LOG10E	Returns the base-10 logarithm of E
PI	Returns PI
SQRT1_2	Returns the square root of 1/2
SQRT2	Returns the square root of 2

Math

Method	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x, in radians
asin(x)	Returns the arcsine of x, in radians
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians
atan2(y,x)	Returns the arctangent of the quotient of its arguments
ceil(x)	Returns x, rounded upwards to the nearest integer
cos(x)	Returns the cosine of x (x is in radians)
exp(x)	Returns the value of E^x

Math

Method	Description
floor(x)	Returns x, rounded downwards to the nearest integer
log(x)	Returns the natural logarithm (base E) of x
max(x,y,z,...,n)	Returns the number with the highest value
min(x,y,z,...,n)	Returns the number with the lowest value
pow(x,y)	Returns the value of x to the power of y
random()	Returns a random number between 0 and 1
round(x)	Rounds x to the nearest integer
sin(x)	Returns the sine of x (x is in radians)
sqrt(x)	Returns the square root of x
tan(x)	Returns the tangent of an angle

Number

Used in two ways:

- **new Number(v)** creates a Number wrapper object, it is not a primitive value
- **Number(v)** convert v to a number primitive value

The second form much more common than the first one

Number

- The Number object has properties for numerical constants, such as maximum value, not-a-number, and infinity.
- You cannot change the values of these properties and you use them as follows:

```
biggestNum = Number.MAX_VALUE
smallestNum = Number.MIN_VALUE
```

Property	Description
MAX_VALUE	Returns the largest number possible in JavaScript
MIN_VALUE	Returns the smallest number possible in JavaScript
NEGATIVE_INFINITY	Represents negative infinity (returned on overflow)
POSITIVE_INFINITY	Represents infinity (returned on overflow)

Number

Method	Description
toExponential(x)	Converts a number into an exponential notation
	number.toExponential(x)
	x Optional. An integer between 0 and 20 representing the number of digits in the notation after the decimal point. If omitted, it is set to as many digits as necessary to represent the value
toFixed(x)	Formats a number with x numbers of digits after the decimal point
	number.toFixed(x)
	x Optional. The number of digits after the decimal point. Default is 0 (no digits after the decimal point)
toPrecision(x)	Formats a number to x length
	number.toPrecision(x)
	x Optional. The number of digits. If omitted, it returns the entire number (without any formatting)
toString()	Converts a Number object to a string
valueOf()	Returns the primitive value of a Number object

Number

```
<script>
let num = new Number(13.3714);
document.write(Number.MAX_VALUE+"<br>");
document.write(Number.MIN_VALUE+"<br>");
document.write(num.toExponential(4)+"<br>");
document.write(num.toFixed(2)+"<br>");
document.write(num.toPrecision(3)+"<br>");
document.write(num.toString()+"<br>");
document.write(num.valueOf());
</script>
```

Number

- Number(value), used as a function, converts a value to a number. If the value cannot be converted, it returns NaN
  - null is converted to 0
  - undefined is converted to NaN
  - true and false are converted to 1 and 0
  - string are parsed, in case of failure NaN
  - objects are converted to primitive values
- Number.parseInt(string)/ Number.parseFloat(string): similar, but just for strings

```
const n1 = Number('ABC');
const n2 = Number('123');
const n3 = Number('');
const n4 = Number([]);
const n5 = Number([33]);
const n6 = Number([33, 44]);
const n7 = Number(undefined);
const n8 = Number(null);
const n9 = Number(true);
const n10 = Number(false);
const n11 = Number({valore: 4,
                valueOf() {
          return this.valore;
         }});
```

```
NaN
123
0
0
33
NaN
NaN
0
1
0
4
```

String

- The String object is a wrapper around the string primitive data type. Do not confuse a string literal with the String object.
- String wrapper objects are created with new String().

```
s1 = "Hi"                    //creates a string literal value
s2 = new String("Hi")        //creates a String object
```

- Similarly to Boolean and Number, String is more frequently used without new, as a conversion function. But mainly, it is a container for methods.

String

- Conversions:
  - undefined is converted to "undefined"
  - null is converted to "null"
  - true and false are converted to "true" and "false"
  - numbers are converted to base 10
  - objects are converted to a primitive (toString() and then valueOf())

```
const o = {valore: 4,
 toString() {
  return "pippo"
 },
 valueOf() {
  return this.valore;
 }};
const s1 = String(undefined);
const s2 = String(null);
const s3 = String(1234);
const s4 = String(true);
const s5 = String(o);
const s6 = "" + o;
```

```
undefined
null
1234
true
pippo
4
```

"" + o should not be used for string coercion as the primitive value of o is obtained differently (priority to valueOf)

String

- You can call any of the methods of the String object on a string literal value
  - JavaScript automatically converts the string literal to a temporary String object, calls the method, then discards the temporary String object.
- You can also use the String.length property with a string literal.

String

Method	Description
charAt()	Returns the character at the specified index
concat()	Joins two or more strings, and returns a copy of the joined strings
	string.concat(string2, string3, ..., stringX)
indexOf()	Returns the position of the first found occurrence of a specified value in a string string.indexOf(searchstring) string.indexOf(searchstring, start)
lastIndexOf()	Returns the position of the last found occurrence of a specified value in a string (-1 if the value to search for never occurs) *string.*lastIndexOf*(searchstring)* *string*.lastIndexOf(searchstring, start)

String

Method	Description
match()	Searches for a match between a regular expression and a string, and returns the matches
	string.match(regexp)
	regexp Required. A regular expression.
replace()	Searches for a match between a substring (or regular expression) and a string, and replaces the matched substring with a new substring
	string.replace(*regexp/substr,newstring*)
	regexp/substr Required. A substring or a regular expression. *newstring* Required. The string to replace the found value in parameter 1
search()	Searches for a match between a regular expression and a string, and returns the position of the match

String

Method	Description
slice()	Extracts a part of a string and returns a new string *string*.slice(begin,end) *begin* Required. The index where to begin the extraction. First character is at index 0 *end* Optional. Where to end the extraction. If omitted, slice() selects all characters from the begin position to the end of the string
split()	Splits a string into an array of substrings
	string.split(separator, limit)
	separator Optional. Specifies the character to use for splitting the string. If omitted, the entire string will be returned *limit* Optional. An integer that specifies the number of splits

String

Method	Description
substr()	Extracts the characters from a string, beginning at a specified start position, and through the specified number of character
substring()	Extracts the characters from a string, between two specified indices
toLowerCase()	Converts a string to lowercase letters
toUpperCase()	Converts a string to uppercase letters
valueOf()	Returns the primitive value of a String object

String

```
<script>
let str="Hello world!";
document.write(str +"<br>");
document.write(str.substring(1) +"<br>");
document.write(str.substring(3,7)+"<br>");
document.write(str.split('o',2)+"<br>");
document.write(str.toUpperCase()+"<br>");
document.write(str.toLowerCase());
</script>
```

RegExp

- A regular expression is an object that describes a pattern of characters.
  - A simple pattern can be one single character.
  - A more complicated pattern can consist of more characters, and can be used for parsing, format checking, substitution and more.

- Regular expressions are used to perform powerful pattern-matching and "search-and-replace" functions on text.

```
let txt = new RegExp(pattern, modifiers);
or more simply
let txt = /pattern/modifiers;
```

pattern specifies the pattern of an expression

modifiers specify if a search should be global, case-sensitive, etc.

RegExp

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)

Method	Description
exec()	Tests for a match in a string. Returns the first match regexp.exec(str) regexp(str)
test()	Tests for a match in a string. Returns true or false regexp.test(str)

RegExp

Expression	Description
[abc]	Find any character between the brackets
[^abc]	Find any character not between the brackets
[0-9]	Find any digit from 0 to 9
[a-z]	Find any character from lowercase a to lowercase z
[A-Z]	Find any character from uppercase A to uppercase Z
[a-Z]	Find any character from lowercase a to uppercase Z
adgk	Find the sequence of characters
(red|blue|green)	Find any of the alternatives specified

RegExp

Metachar	Description
.	Find a single character, except newline or line terminator
\w	Find a word character
\W	Find a non-word character
\d	Find a digit
\D	Find a non-digit character
\s	Find a whitespace character
\S	Find a non-whitespace character
\b	Find a match at the beginning/end of a word
\B	Find a match not at the beginning/end of a word

RegExp

Metachar	Description
\0	Find a NUL character
\n	Find a new line character
\f	Find a form feed character
\r	Find a carriage return character
\t	Find a tab character
\v	Find a vertical tab character
\xxx	Find the character specified by an octal number xxx
\xdd	Find the character specified by a hexadecimal number dd
\uxxxx	Find the Unicode character specified by a hexadecimal number xxxx

RegExp

Quantifier	Description
n+	Matches any string that contains at least one n
n*	Matches any string that contains zero or more occurrences of n
n?	Matches any string that contains zero or one occurrences of n
n{X}	Matches any string that contains a sequence of X n's
n{X,Y}	Matches any string that contains a sequence of at least X and less than or equal to Y n's
n{X,}	Matches any string that contains a sequence of at least X n's
n$	Matches any string with n at the end of it
^n	Matches any string with n at the beginning of it
?=n	Matches any string that is followed by a specific string n
?!n	Matches any string that is not followed by a specific string n

RegExp

```
<body>
<script>
let s1 = "10, 200, 2000, 10000 or 3000000?";
let p1 = /\d{3,4}/g;
let v1 = s1.match(p1);
document.write(v1);
document.write("<br>");
let s2 = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, " +
  "sed do eiusmod tempor incididunt ut labore et dolore a magna aliqua.";
let p2 = /[abc]\w+/g;
let v2 = s2.match(p2);
document.write(v2);
</script>
</body>
```

Global Properties and Functions

- The JavaScript global properties and functions can be used with all the built-in JavaScript objects.

Property	Description
Infinity	A numeric value that represents positive/negative infinity
NaN	"Not-a-Number" value
undefined	Indicates that a variable has not been assigned a value

Function	Description
decodeURI()	Decodes a URI
decodeURIComponent()	Decodes a URI component
encodeURI()	Encodes a URI
encodeURIComponent()	Encodes a URI component

Global Properties and Functions

Function	Description
eval()	Evaluates a string and executes it as if it was script code. First, eval() determines if the argument is a valid string, then eval() parses the string looking for JavaScript code. If it finds any JavaScript code, it will be executed.
isFinite()	Determines whether a value is a finite, legal number
isNaN()	Determines whether a value is an illegal number
parseFloat()	Parses a string and returns a floating point number
parseInt()	Parses a string and returns an integer

Global Properties and Functions

```
<script type="text/javascript">
    eval("x=10;y=20;document.write(x*y)");
    document.write("<br>" + eval("2+2"));
    document.write("<br>" + eval(x+17));
</script>
```

Output:

200

4

27

Elements of modern JavaScript

Alessio Vecchio

University of Pisa

for of

- Introduced in ES6
- It can be used with iterable objects
- Iterable objects: strings, arrays, sets, maps

```
let mya = [11, 2, 3, 55, 4, 12, 32];

for(let x of mya) {
  console.log(x);
}
```

```
11
2
3
55
4
12
32
```

for of

- Objects are generally not iterable, generates *TypeError* at runtime
- You can iterate through the properties of an object using ***for in*** or obtain an array of properties and then use *for of*

```
let o1 = {"a": "one",
          "b": "two",
           c: 3};

for(let x of Object.keys(o1)) {
  console.log(x + ": " + o1[x]);
}
```

```
a: one
b: two
c: 3
```

Object.keys() returns an array with the property names

for of

- Same for object values. *Object.values()* returns the properties' values.

```
for(let x of Object.values(o1)) {
  console.log(typeof x + " " + x);
}
```

```
string one
string two
number 3
```

- or for key-value pairs

```
for(let [kk, vv] of Object.entries(o1)) {
  console.log(kk + ": " + vv);
}
```

```
a: one
b: two
c: 3
```

Object.entries() returns an array of arrays. Each internal array is a key-value pair.

for of

- An example with a string

```
let q = {};
for(let c of "Progettazione Web") {
  if(q[c]) q[c]++;
  else q[c] = 1;
}
console.log(q);
```

```
{
  P: 1,
  r: 1,
  o: 2,
  g: 1,
  e: 3,
  t: 2,
  a: 1,
  z: 1,
  i: 1,
  n: 1,
  ' ': 1,
  W: 1,
  b: 1
}
```

Destructuring assignment

- Introduced in ES6
- Syntax:

  one or more variables = a compound object, generally an array

```
let [a, b, c] = [10, 20, 30];
console.log(a + " " + b + " " + c);
// Output:
// 10 20 30
```

```
// Exchanges the value of a, b
[a, b] = [b, a];
console.log(a + " " + b);
// Output:
// 20 10
```

Destructuring assignment

```
// Not all elements must be used
[a, b] = ['ABC', 'DEF', 'GHI'];
console.log(a + " " + b + " " + c);
// Output:
// ABC DEF 30

// If too many variables, last ones are undefined
let [x, y, z] = [1, 2];
console.log(x + " " + y + " " + z);
//Output:
// 1 2 undefined

// Some values can be skipped using commas
[,x,,y,,z] = ['a', 'b', 'c', 'd', 'e', 'f'];
console.log(x + " " + y + " " + z);
// Output:
// b d f
```

Destructuring assignment

```
// Can be used with functions
function myfun(){
  let p = 100, q = 200;
  // Some code
  return [p, q];
}
[x,y] = myfun();
console.log(x + " " + y);
// Output:
// 100 200

// The set of remaining values can
// be collected in a single one
[x, ...y] = [10, 20, 30, 40, 50, 60];
console.log("x=" + x + ", y=" + y);
// Output:
// x=10 y=20,30,40,50,60
```

```
// Works also with objects
// but less used
let o1 = {k1: 42, k2: 24};
let {k1, mk2} = o1;
console.log(k1 + " " + mk2);
// Output:
// 42 undefined
```

Template string literals

- Introduced in ES6
- Delimited by **backticks**
- Everything within **${** and **}** is considered as a JS expression

```
let topic = "Progettazione Web";
let s1 = `My favourite topic is ${topic}`;
console.log(s1);
// Output:
// My favourite topic is
// Progettazione Web

let x = "X";
let y = 42;
let s2 = `Some text, ${x} and ${y*2}`;
console.log(s2);
// Output:
// Some text, X and 84
```

Default parameters

- Introduced in ES6
- Arguments without a value are generally initialized to undefined
- It is now possible to have default values that are used when a value is not provided during call
- Parameters with default values must be at the end of the list

```
function f1(a1, a2 = 10, a3 = 'X'){
  // Some code, here just print
  console.log(a1 + " " + a2 + " " + a3);
}

f1(1, 2, 'Z');
// Output:
// 1 2 Z
f1(1, 2);
// Output:
// 1 2 X
f1(1);
// Output:
// 1 10 X
```

Classes

- Introduced in ES6
- Classes can be defined according to a style that is similar to the one of other OO programming languages
- Classes can be declared using the **class** keyword
- The class body **{ }** contains the definition of the constructor and methods
- constructor can be omitted if not needed
- In methods the *function* keyword is not used

Classes

```
class Student {
  constructor(n, d) {
    this.name = n;
    this.degree = d;
    this.marks = [];
  }
  toString() {
    return `${this.name}, ${this.degree}`;
  }
  addMarks(m) {
    this.marks.push(m);
  }
  getAverage(){
    let s = 0;
    for(let m of this.marks) {
      s += m;
    }
    return s/this.marks.length;
  }
}
```

Classes

```
let s1 = new Student("Mario",
                "Ing. Informatica");
console.log("s1=" + s1);

let s2 = new Student("Luigi",
              "Computer Engineering");
console.log("s2=" + s2);

console.log(s1.getAverage());
s1.addMarks(27);
s1.addMarks(28);
console.log(s1.getAverage());
```

```
// Output:
s1=Mario, Ing. Informatica
s2=Luigi, Computer Engineering
NaN
27.5
```

Classes

- A class must be declared before being used
- Syntactic sugar, underlying model still the same
- Single inheritance is supported
- super to call superclass constructor and methods

```
class WorkingStudent extends Student {
 constructor(n, d, c, f=true) {
  super(n, d);
  this.company = c;
  this.fulltime = f;
 }
 toString(){
  let s = super.toString();
  return s +
    `, company=${this.company}, fulltime=${this.fulltime}`;
 }
 applyBonus(b){
  if(this.fulltime)
   for(let i=0; i<this.marks.length; i++)
    this.marks[i] = this.marks[i]*b;
 }
}
```

Classes

```
let w1 = new WorkingStudent("Gino",
                            "Ing. Informatica", "Google");
w1.addMarks(30);
w1.applyBonus(1.05);
console.log("w1=" + w1);
console.log(w1.getAverage());

// Output:
// w1=Gino, Ing. Informatica, company=Google, fulltime=true
// 31.5
```

Strict mode

- Introduced in ES5
- A directive that is applied by inserting the special **"use strict";** string expression statement at the beginning of a file
- Code in that file is executed in strict mode
  - all variables must be declared
  - *with* can't be used
  - duplicate parameters generate error
  - additional checks
- Classes are automatically in strict mode

Strict mode

```
"use strict";

x = 21;
// x = 21;
// ^
// ReferenceError: x is not defined

o1 = {p1: 10, p2: 20};
with(o1) {
  console.log(p1);
  console.log(p2);
}
// with(o1) {
// ^^^^
//
// SyntaxError: Strict mode code may not include a with statement

function f(a1, a1) {
  console.log(a1);
}
f(10, 20);
// SyntaxError: Duplicate parameter name not allowed in this context
```

Arrow functions

- Introduced in ES6
- Syntax: **parameters => body of the function**
- The *function* keyword is not used
- Useful when you have to pass a function with limited complexity

```
// No arrow function yet
const f1 = function (a, b) {
  return a + b;
}
console.log(f1(10, 20));
// Output:
// 30

// Now arrow function
const f2 = (a, b) => {return a + b};
console.log(f2(10, 20));
// Output:
// 30
```

Arrow functions

```
// If the body just a return statement then
// return keyword and {} can be omitted
const f3 = (x, y, z) => x+y+z;
let r1 = f3(5, 6, 7);
console.log(r1);
// Output:
// 18
```

```
// If a single parameter can omit ()
const square = x => x*x;
console.log(square(11));
// Output:
// 121
```

```
// If no parameters, still have to use ()
const f4 = () => {return Math.random()*10};
let r2 = f4();
console.log(r2);
// Output:
// 7.4960047952343
```

Arrow functions

- Passing as argument

```
let ar1 = [10, 20, 30];
ar1.forEach(function (e) {
        let t = Math.random() * e;
        console.log(t);
       });
// Output:
// 2.88029933302546
// 5.757679342468269
// 27.80074927780763

// with arrow function
ar1.forEach((e) => {
        let t = Math.random()*e;
        console.log(t);
        });
// Output:
// 9.58533802031983
// 8.773334995700527
// 12.840542468303616
```

Spread operator

- The spread operator ... can be used with arrays (ES6)

```
let a1 = [10, 20, 30, 50];
// a1 is spread and its elements become elements of a2
let a2 = [1, ...a1, 100];
console.log(a2, "\n");

// can be used to create a shallow copy
let v = ['first', 'second', 'third'];
let c = [...v];
console.log(c, "\n");

// the copy is shallow
let x1 = [{AAA: 1, BBB: 2}, {CCC: 3, DDD: 4}];
let x2 = [...x1];
x1[0].AAA = 999;
console.log(x2, "\n");

// strings can be spread into an array of strings
let s = "This is a sentence";
let charactersOfS = [...s];
console.log(charactersOfS, "\n");
```

```
[ 1, 10, 20, 30, 50, 100 ]


[ 'first', 'second', 'third' ]


[ { AAA: 999, BBB: 2 },
  { CCC: 3, DDD: 4 } ]


[
  'T', 'h', 'i', 's', ' ',
  'i', 's', ' ', 'a', ' ',
  's', 'e', 'n', 't', 'e',
  'n', 'c', 'e'
]
```

Spread operator

- can be used with objects as well (ES2018)

```
// objects can be spread as well
let stud1 = {name: "Mario", averageGrade: 25};
let r1 = {...stud1, course: "Programmazione Web"};
console.log(r1, "\n");

// if property name is the same, last one wins
let r2 = {...stud1, name: "Gino"};
console.log(r2, "\n");

// inherited properties are not spread
let p1 = {myproperty: "ABC"};
let p2 = Object.create(p1);
console.log(p2.myproperty);
let p3 = {...p2};
console.log(p3.myproperty);
```

```
{ name: 'Mario', averageGrade: 25, course:
'Programmazione Web' }


{ name: 'Gino', averageGrade: 25 }


ABC
undefined
```