



# Introduzione

## Prerequisiti

Conoscenza di base di **Javascript**

## Obiettivi

Conoscenza delle principali tecniche avanzate di Javascript, che ci consentiranno di affrontare al meglio argomenti come **Ajax** e **Angular**

# Contenuti

## Concetti avanzati di programmazione in Javascript

- Callbacks e la programmazione asincrona
- Le funzioni anonime
- Closure e First Class functions

## JavaScript Object Oriented

- I concetti principali di javascript OOP
- Classi e Oggetti, Prototype
- Proprietà e Metodi

## ECMAScript 6 (ES6) e TypeScript: nuove caratteristiche

- Introduzione a TypeScript
- Tipi di Dati (number, string, array, boolean, any) Proprietà e Metodi
- Ambiti di visibilità: public, private, protected, static
- I concetti fondamentali dell'OOP in Javascript: Incapsulamento, ereditarietà, Polimorfismo
- Le classi in TypeScript
- Ereditarietà
- Interfaces, Classe astratte

## Programmazione asincrona: nuove caratteristiche ES6/TypeScript

- Arrow function (=>)
- Promises Vs. Observables

# Sezione 1

## I TIPI DI DATO IN JAVASCRIPT

- number
- string
- boolean
- null
- undefined
- object

## Dichiarazione di variabili

- var, let, const
- Visibilità (Scope) delle variabili

## LE FUNZIONI JAVASCRIPT

- Declaration vs Expression
- Callbacks
- Funzioni anonime
- Closure function e First Class function
- IIFE (Immediately Invoked Function Expression)
- Arrow function (=>)



- Javascript a differenza di altri linguaggi ha un solo tipo di dato per rappresentare i numeri e questo è detto “number”.
- Con number rappresentiamo numeri positivi, negativi, decimali, in diverse basi
- Il numero più grande rappresentabile è Infinity, mentre quello più piccolo è -Infinity
- NaN (NOT A NUMBER) è un valore numerico speciale, che indica un valore numerico non definito

## number

```
●●●

1 //numeri in base 10
2 var interoPositivo = 2;
3 var interoNegativo = -5;
4 var decimalePositivo = 1.2;
5 var decimaleNegativo = -12.1;
6
7 // numeri in base 8 iniziano con lo 0
8 var baseOttale = 0123; // corrisponde ad 83 in base decimale
9 var altroNumeroOttale = 012; // corrisponde a 10 in base decimale
10
11 // numeri in base 16 iniziano con 0x
12 var baseEsadecimale = 0x123; // corrisponde a 291 in base decimale
13
14 /*
15 * L'insieme dei numeri rappresentabili in JavaScript cade nell'intervallo
16 * compreso tra -1.79769*10308 e 1.79769*10308, con una precisione pari a
17 * 5*10-324. Ogni valore che va al di fuori dell'intervallo rappresentabile
18 * non genera un'eccezione ma viene rappresentato da due valori speciali:
19 */
20 var infinitoPositivo = Infinity;
21 var infinitoNegativo = -Infinity;
22
23 /*
24 * Un altro valore numerico speciale è NaN, acronimo di Not a Number,
25 * che indica un valore numerico non definito. Nel seguente esempio il valore
26 * della variabile x dopo l'assegnamento è NaN:
27 */
28 var x = x + 1; // NaN
```



## string

- Una stringa in Javascript è una sequenza di caratteri delimitata da doppi apici, singoli apici e backtick(``)

```
//Le Stringhe

var singoloApice = 'ciao';
var doppioApice = "benvenuti";
var backtick = `al corso di
javascript avanzato`;
// con i backtick (Windows ALT+96, MAC ALT+9) possiamo andare a capo

/**
 * grazie al backtick `` (template literals introdotte nella versione ES6)
 * possiamo inserire delle espressioni javascript direttamente all'interno della stringa
 * come vediamo nell'esempio qui sotto
 */

var eta = 24;
var frase = `Ciao a tutti mi chiamo Ryan e ho ${eta} anni`;
```



- Il tipo di valore boolean prevede solo due valori che sono true (vero) e false (falso)
- Possiamo ottenere un valore boolean attraverso una comparazione con i simboli (`>`, `<`, `>=`, `<=`, `==`, `!=`, `====`, `!==`)
- Oppure attraverso gli operatori logici `&&(AND)` e `||(OR)`

## boolean

```
1 // i valori boolean sono solamente true e false (vero e falso);
2 var vero = true;
3 var falso = false;
4
5 // possiamo ottenere un valore booleano usando gli operatori di comparazione
6
7 // (>) maggiore e (>=) maggiore uguale
8 var maggiore = 10 > 33; // false
9 var maggioreUguale = 23 >= 12; //true
10
11 // (<) minore e (<=) minore uguale, comparano se il valore a sinistra
12 var minore = 21 < 22 // true
13 var minoreUguale = 102 <= 32 //false
14
15 /**
16 * (==) doppio uguale che controlla che il valore a sinistra sia uguale a
17 * quello a destra senza
18 * valutare il tipo di dato
19 */
20 var doppioUguale = '4' == 4; //true
21
22 /**
23 * (====) triplo uguale come l'operatore doppio uguale controlla che il valore
24 * a sinistra sia uguale a quello a destra, inoltre controlla anche il tipo
25 * di dato
26 */
27 var triploUguale = '4' === 4; //false
28
29 /**
30 * (||) operatore logico OR torna come risultato false solamente quando
31 * entrambi i valori a sinistra e destra solo false nei restanti 3 casi sarà
32 * sempre true
33 */
34 var falseOrFalse = false || false; //false
35 var trueOrFalse = true || false; //true
36 var falseOrTrue = false || true; //true
37 var trueOrTrue = true || true; //true
38
39 /**
40 * (&&) operatore logico AND torna come risultato true solamente quando
41 * entrambi i valori a sinistra e destra sono true nei restanti 3 casi sarà
42 * sempre false
43 */
44 var trueAndTrue = true && true; //true
45 var trueAndFalse = true && false; //false
46 var falseAndTrue = false && true; //false
47 var falseAndFalse = false && false; //false
```



## undefined e null

- undefined rappresenta un valore che non esiste, il valore di una variabile non inizializzata è undefined
- Il valore null è tipo di dato a che non rientra negli altri tipi di dato, non è né una stringa, né un numero, né un oggetto.

```
1 /**
2 * undefined rappresenta un valore che non esiste
3 * corrisponde a una variabile non inizializzata
4 */
5 var test; // undefined
6
7 /**
8 * Il tipo di dato null prevede il solo valore null
9 * ed è ben diverso dal valore undefined
10 * precedentemente analizzato
11 */
12 var x = null;
```

carbon  
carbon.now.sh



- Un oggetto può essere rappresentato con delle parentesi graffe {} (oggetto vuoto)
- Un oggetto tipicamente possiede proprietà e metodi
- Le proprietà possono assumere qualsiasi valore tra quelli descritti in precedenza
- Un metodo non è altro che una funzione
- Anche gli array sono degli oggetti, possiedono proprietà come .length e metodi come il .foreach()

## object

```
●●●  
1 /**
2  * un object o oggetto è una struttura dati
3  * rappresentata con parentesi graffe {}
4  * che racchiude proprietà e metodi
5 */
6
7 var persona = {
8   nome: 'Ryan Jherome', // proprietà
9   cognome: 'Burgos', // proprietà
10  saluta: function() { // metodo
11    return `Ciao a tutti sono ${this.nome} ${this.cognome}`
12  }
13}
14
15 /**
16  * Gli array sono degli object anch'essi
17  * hanno proprietà e metodi;
18 */
19 var array = [1, 2, 3, 4, 5]
20
```

## Dichiarazione di una variabile

- Per dichiarare una variabile abbiamo imparato ad usare la parola chiave **var**, con l'avvento della versione **ES6** di Javascript sono state introdotte le parole chiavi **let** e **const**
- Lo **Scope Globale** e lo **Scope Locale**
- **let** permette di dichiarare variabili limitandone la visibilità ad un blocco di codice (**Blocked-Scope**) (per blocco di codice si intendono le righe di codice racchiuse da parentesi graffe { })
- **const** come let limita la visibilità ad un blocco di codice (**Blocked-Scope**), ma questa istruzione rende il valore immagazzinato costante, ovvero, non può né essere riassegnato né ridichiarato



## Lo Scope

- **Scope globale:** accessibilità estesa all'intero script;



```
var x = 'sono una variabile nello scope globale'; // variabile nello Scope Globale
```

- **Scope locale:** accessibilità ristretta al solo codice di una funzione o di un blocco di codice.



```
function test() {  
    var x = 'sono una variabile nello scope locale'; // variabile nello Scope Locale  
}
```



- In questo esempio la variabile **var x = 'fuori'** è **una variabile globale**, e ciò la rende visibile in tutto lo script
- La variabile **var x = 'dentro'** è **una variabile locale**, interna alla funzione test(), questa variabile come vediamo nell'esempio non va a sovrascrivere la variabile x fuori dalla funzione, perché ha un ambito di visibilità racchiuso nella sola funzione
- Domanda:  
Cosa succede se nella funzione non usiamo la parola chiave **var** ?

## Lo Scope



```
var x = 'fuori'; // variabile nello Scope Globale

function test() {
    var x = 'dentro'; // variabile nello Scope Locale
    console.log( x );
}

test(); // dentro
console.log(x); // fuori
```



## Lo Scope

- All'esecuzione della funzione `test()`, **il valore della variabile globale x sarà riassegnato**, quindi passerà da 'fuori' a 'dentro'

```
● ● ●

var x = 'fuori'; //variabile nello Scope Globale

function test() {
    // all'esecuzione della funzione viene sovrascritta la variabile globale
    x = 'dentro';
    console.log(x);
}

test(); // dentro
console.log(x); //dentro
```



## Lo Scope con Javascript ES6

- Con l'avvento della versione ES6 sono stati introdotte due nuove parole chiave per la dichiarazione di variabili, **let** e **const**
- La particolarità di **let** e **const** è quella di limitare la sua visibilità ai blocchi di codice che la contengono



```
var x = 4;  
  
if (true) {  
    var x = 8;  
}  
  
console.log(x) // 8
```



```
let x = 4;  
  
if (true) {  
    let x = 8;  
}  
  
console.log(x) // 4
```



## Lo Scope con Javascript ES6

- Vediamo un esempio con un ciclo for



```
for (var x = 0; x < 4; x++) {  
    var y = 10;  
}  
  
console.log(x); // >> 4  
console.log(y); // >> 10
```



```
for (let x = 0; x < 4; x++) {  
    let y = 10;  
}  
  
console.log(x); // >> "ReferenceError: x is not defined"  
console.log(y); // >> >> "ReferenceError: y is not defined"
```

## Le Funzioni

- Una funzione si definisce con la parola chiave **function** seguito dal nome della funzione e dalle parentesi tonde  
Es: **function test() { // blocco di codice }**
- Per richiamare o invocare una funzione basta scrivere il suo nome seguito dalle parentesi tonde **test()**
- Una funzione è un insieme di istruzioni racchiuse in un blocco di codice (tra parentesi graffe {})
- Può essere anonima o contraddistinta da un nome
- Può accettare uno o più argomenti
- Può restituire un valore o eseguire semplicemente le istruzioni al suo interno

## function declaration vs function expression

- Una **function expression** non gode dell'hoisting invece, una function expression non è altro che una funzione che viene assegnata ad una variabile, chiaramente essa potrà essere invocata solo dopo che la funzione sarà assegnata alla variabile, possiamo invocare la funzione con il nome stesso della variabile seguito dalle parentesi tonde

```
/*  
 * la function expression come vediamo non si comporta come la function declaration  
 * infatti quando proveremo ad eseguire la funzione prima che essa venga assegnata alla  
 * variabile, la console ci dirà che ciò che stiamo provando ad eseguire non è una funzione  
 */  
  
expression(); // >> Uncaught TypeError: expression is not a function  
  
var expression = function() {  
    console.log('sono una function expresison');  
}  
  
// la funzione potrà essere invocata solo dopo l'assegnazione  
expression(); // >> sono una function expresison
```

## function declaration vs function expression

- Una **function declaration** o dichiarazione di funzione si dichiara con la parola chiave function seguita dal nome della funzione, grazie all'hoisting, essa può essere scritta, alla fine dello script, ma verrà comunque caricata prima di tutto il resto, perciò la funzione potrà essere utilizzata anche prima che essa venga dichiarata

```
/*  
 * come possiamo vedere, grazie all'hoisting, possiamo invocare la funzione declaration  
 * prima che essa sia dichiarata, perché all'inizio dell'esecuzione javascript porta  
 * dichiarazioni di funzioni in testa al codice  
 */  
  
declaration(); // >> sono una dichiarazione di funzione  
  
function declaration() {  
    console.log('sono una dichiarazione di funzione');  
}
```

## Funzioni con argomenti

- Una funzione può ricevere uno o più argomenti
- Un argomento è un valore passato in input alla funzione, esso può essere un qualsiasi tipo di dato che abbiamo descritto in precedenza, persino una funzione

```
/*  
 * calcola il quadrato dell'argomento che viene passato in input  
 * Funzione con un solo argomento  
 */  
function square(n) {  
    return n*n;  
}  
  
/*  
 * calcola la somma dei due argomenti che vengono passati  
 * in input  
 */  
function calcolaSomma(a, b) {  
    return a+b;  
}
```



## Funzioni con argomenti

- Può accadere che abbiamo bisogno di scrivere una funzione che accetti un numero di argomenti che indefinito, che noi non conosciamo
- È possibile inserire un numero indefinito di argomenti con lo spread operator, rappresentato con tre punti seguito dal segnaposto (es: ...args)
- args all'interno della funzione sarà visibile come un array contenente tutti gli argomenti



```
// questa funzione calcola la somma dei numeri passati come argomento nella funzione
function sommaNumeri(...numbers) {
    var somma = 0;
    for (var n of numbers) {
        somma += n;
    }
    return somma;
}
```



## Arrow function

- Con la versione ES6 abbiamo a disposizione un modo più conciso per scrivere una funzione e questa è l'arrow function
- Essa si scrive `() => { //blocco di codice }`



```
// arrow function  
let sottrazione = (a, b) => a-b;
```



```
// arrow function  
var somma = (a, b) => {  
    return a+b;  
}
```



```
// arrow function  
let quadrato = a => a*a;
```



## **Callbacks**

- Una funzione di callback è letteralmente una funzione passata come argomento ad un'altra funzione
- In Javascript ne viene fatto un uso massivo, in modo particolare nella programmazione asincrona
- Molto spesso si fa spesso di funzioni anonime come funzioni di callback



## Uso delle funzioni di callback in jquery



```
$('button').on('click', function() {  
    console.log("sono all'interno di una funzione di callback");  
})
```



## Uso delle funzioni di callback in un metodo di un array



```
let numeri = [1, 2, 3, 4, 5, 6]

/**
 * forEach accetta come argomento una funzione di callback
 * che verrà eseguita per ogni elemento presente nell'array
 */
numeri.forEach(n => {
  console.log(n)
})
```



## Una funzione custom che accetta una funzione di callback

```
/*  
 * è importante quando creiamo una funzione di questo genere  
 * controllare se ciò che stiamo passando come callback è una funzione  
 */  
function calcola(arg1, arg2, func) {  
    if (func && typeof func === 'function') {  
        return func(arg1, arg2)  
    }  
}  
  
function somma(a, b) {  
    return a + b  
}  
  
console.log(calcola(5, 6, somma)) // >> 11
```

## IIFE – IMMEDIATELY INVOCATE FUNCTION EXPRESSION

- La particolarità di questo tipo di funzione è che viene immediatamente eseguita al momento della sua creazione
- Essa è avvolta da parentesi tonde `(function (arg) { console.log(arg)})('Hello World')` seguita da due parentesi tonde che accettano gli argomenti della funzione avvolta
- Tutte le variabili al suo interno e la funzione stessa non esistono più dopo l'esecuzione
- Campi di applicazione (es <https://www.html.it/pag/48373/module-pattern/>)

```
● ● ●

/** 
 * IIFE Immediately Invoke Function Expression
 * Questa funzione verrà invocata non appena verrà creata
 */

(function(arg) {
    console.log(arg); // >> Hello World!
})('Hello World!')
```



## Funzioni annidate Closure

- Una funzione annidata, è una funzione che è all'interno di un'altra funzione
- La funzione che la contiene delimita il suo scope e per questo Una funzione può avere come valore di ritorno una funzione (Closure) che ha accesso alle variabili interne alla funzione contenitore

```
/*  
 * Closures  
 */  
  
function creaAddendo(x) {  
    return function(y) {  
        return x+y;  
    }  
}  
  
let addFive = creaAddendo(5);  
let addTen = creaAddendo(10);  
  
console.log(addFive(12)); // >> 17  
console.log(addTen(11)); // >> 21
```



# Sezione 2

## Gli Oggetti in JavaScript

- Che cos'è un oggetto in JavaScript?
- Le particolarità degli oggetti in JavaScript
- Oggetti standard e oggetti custom

## Il Browser Object Model (BOM)

- Che cosa è il BOM
- Le caratteristiche del BOM
- La struttura del BOM

## Il Document Object Model (DOM)

- Che cos'è il DOM
- Le caratteristiche del DOM'
- Il DOM e il BOM

## Introduzione al paradigma della programmazione a oggetti in Javascript

- Classi, Prototype e Oggetti
- Funzioni Costruttore
- Manipolazione



## Gli Oggetti in JavaScript

- Un oggetto può essere rappresentato con delle parentesi graffe {} (oggetto vuoto)
- Un oggetto tipicamente possiede proprietà e metodi
- Le proprietà possono assumere qualsiasi valore tra quelli descritti in precedenza
- Un metodo non è altro che una funzione
- Anche gli array sono degli oggetti, possiedono proprietà come .length e metodi come il .foreach()

```
 1 /**
 2  * un object o oggetto è una struttura dati
 3  * rappresentata con parentesi graffe {}
 4  * che racchiude proprietà e metodi
 5  */
 6
 7 var persona = {
 8   nome: 'Ryan Jherome', // proprietà
 9   cognome: 'Burgos', // proprietà
10   saluta: function() { // metodo
11     return `Ciao a tutti sono ${this.nome} ${this.cognome}`
12   }
13 }
14
15 /**
16  * Gli array sono degli object anch'essi
17  * hanno proprietà e metodi;
18 */
19 var array = [1, 2, 3, 4, 5]
20
```

## Il this in Javascript

- Il this fa riferimento all'oggetto che lo contiene
- Il this in questo caso è l'oggetto che è assegnato alla variabile persona
- Il this espresso nello spazio global afferisce all'oggetto window

```
●●●  
1 /**
2  * un object o oggetto è una struttura dati
3  * rappresentata con parentesi graffe {}
4  * che racchiude proprietà e metodi
5 */
6
7 var persona = {
8   nome: 'Ryan Jherome', // proprietà
9   cognome: 'Burgos', // proprietà
10  saluta: function() { // metodo
11    return `Ciao a tutti sono ${this.nome} ${this.cognome}`
12  }
13}
14
15 /**
16  * Gli array sono degli object anch'essi
17  * hanno proprietà e metodi;
18 */
19 var array = [1, 2, 3, 4, 5]
20
```

# Gli oggetti in javascript

Si dice che in Javascript tutto ciò che non è un valore primitivo (numeri, stringhe, booleani, NULL, undefined) è un OGGETTO, ovvero di tipo Object

Quindi, un oggetto è una variabile di tipo Object, ma può essere anche un array, oppure una funzione

Gli oggetti sono contenitori di dati eterogenei e possono essere usati per classificare dati complessi come una persona, una macchina, i dati inerenti ad un record di un database etc

Per quanto riguarda la definizione di oggetti e di classi dobbiamo distinguere tra **oggetti predefiniti** ed **oggetti custom**

Uno dei **punti di forza di JavaScript** è la sua capacità di lavorare direttamente con elementi della pagina Web, che viene gestita tramite una “**gerarchia di oggetti JavaScript**”.

Questa gerarchia contiene una notevole varietà di oggetti che rappresentano i dati nella pagina Web corrente e nella finestra del browser in esecuzione.

## Gli oggetti in Javascript

Tutti gli oggetti del Browser (o successivamente dell'interfaccia DOM) sono organizzati in una gerarchia di **oggetti genitori** e **oggetti figli**.

Un oggetto figlio non è altro che un oggetto derivato da un altro oggetto (l'oggetto genitore). Ciò significa che ciascun oggetto appartenente alla gerarchia include **proprietà** che a loro volta sono spesso altri oggetti. (Es: `window.location.href = 'http://www.google.com' - redirect`)

Un oggetto può avere dei **metodi** che servono per **eseguire funzioni su di esso**; può includere dei gestori di eventi (Es: `onclick`) che chiamano funzioni o enunciati javaScript quando si verifica un evento relativo a quell'oggetto.

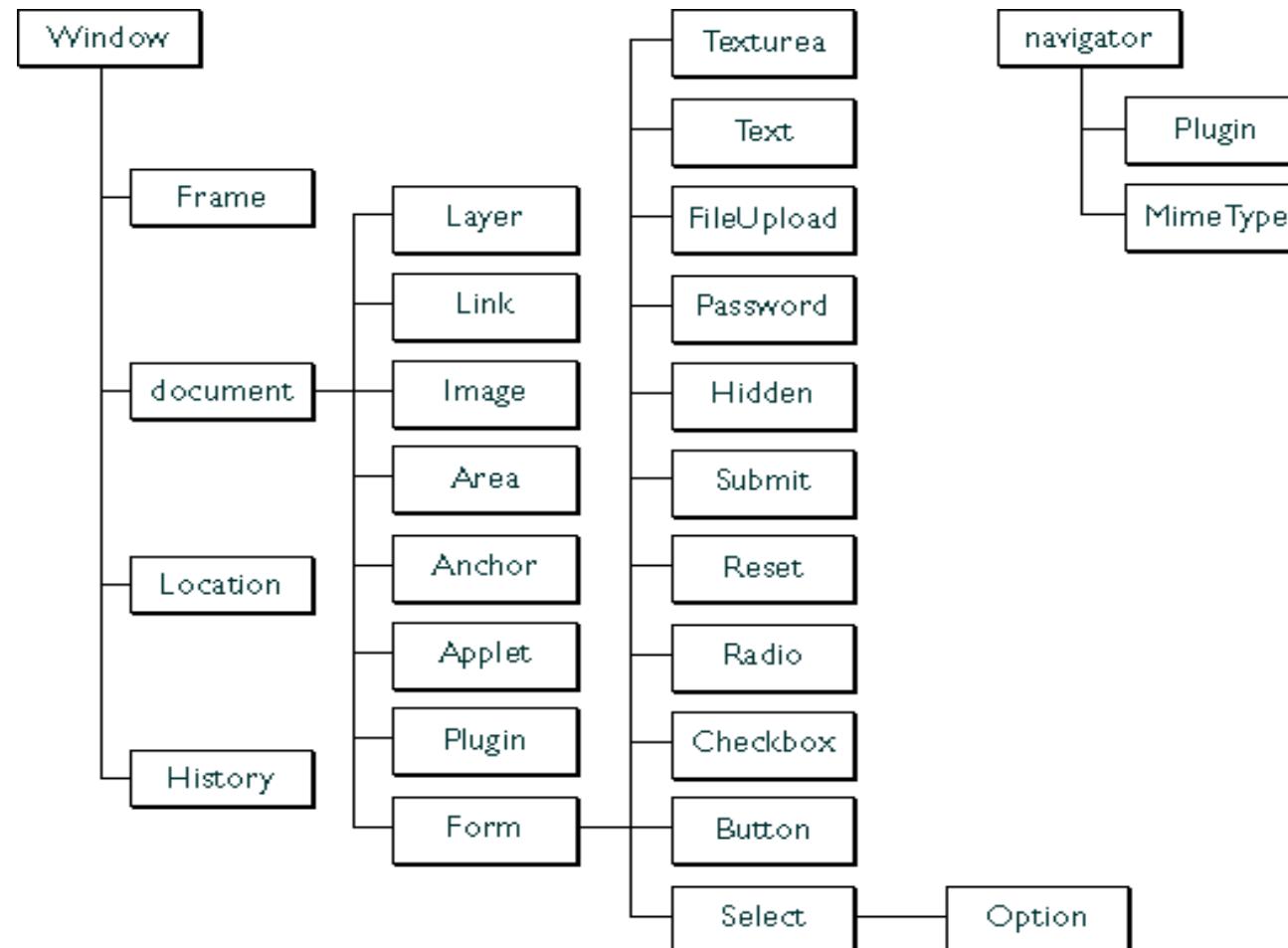
A questo punto esaminiamo l'oggetto che è al **top della gerarchia degli oggetti Javascript**: l'oggetto `window` (Es: `window.open`, `window.close`, `window.alert`). Es:  
`window.open('http://www.google.com' – redirect con target=blank – apertura di una nuova scheda del browser)`



## Il Browser Object Model

**Le caratteristiche del BOM**

# Approccio BOM – Browser Object Model





## Oggetti standard dell'oggetto window



```
// memorizza la posizione (URL) che è visualizzata nella finestra  
window.location
```



**LABFORTRAINING**

**WWW.LABFORTRAINING.IT**

## L'approccio DOM

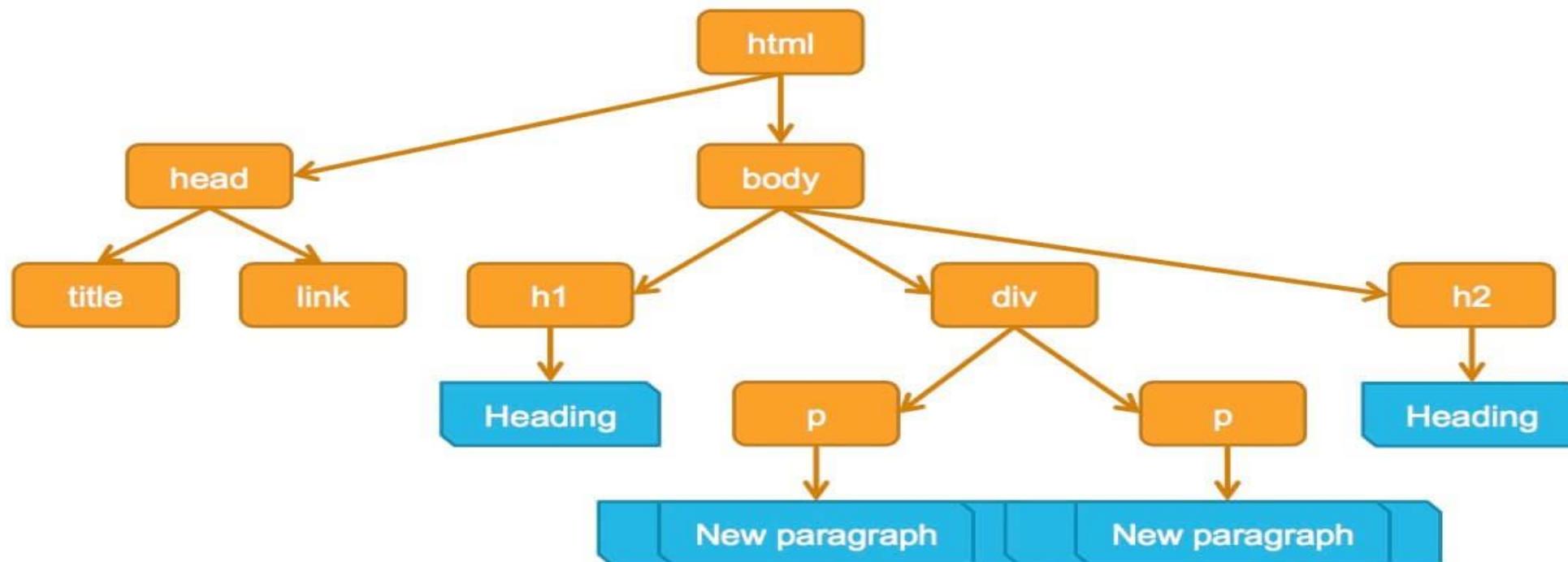
Il **DOM**, come dice la parola stessa (**Document Object Model**) è un **modello**, ovvero un modo di rappresentare il document, ovvero struttura e contenuto della pagina web.

In termini più tecnici: il **DOM** è un'interfaccia, o meglio una **API (Application Programming Interface)** ideata dal consorzio **W3C**, che **permette di accedere agli elementi di una pagina** e di farne elaborazioni (**accedere, modificare, cancellare, attraversare**)

Il **DOM** definisce una serie di funzionalità per **accedere e manipolare** un documento (sia **HTML che XML**) rappresentandolo tramite una **struttura gerarchica ad albero**. In base a questo approccio, la struttura ed il contenuto di un documento viene creata in memoria come una **gerarchia di oggetti**, ciascuno dei quali rappresenta i diversi elementi del documento.

# DOM – Document Object Model

## JavaScript and the DOM



# Le funzioni DOM

L'oggetto di base per la manipolazione del DOM è **document** il quale rappresenta **la radice** del documento nel suo complesso (cioè la pagina web). L'oggetto document è il genitore di tutti gli altri elementi della pagina web. Di seguito una rassegna delle principali funzioni DOM Javascript divise per categoria

## Funzioni per selezionare uno o più elementi

- **document.getElementById()** - identifica un solo elemento della pagina attraverso l'attributo ID (univoco);
- **document.getElementsByTagName()** - identifica un set di elementi della pagina attraverso l'indicazione dello specifico tag (ad esempio tutti i paragrafi o tutti i link);
- **document.getElementsByClassName()** - identifica un set di elementi della pagina attraverso l'indicazione di una specifica classe;
- **document.querySelector ()** - identifica il primo elemento con quell'identificatore
- **document.querySelectorAll()** - identifica un set di elementi della pagina attraverso l'indicazione di una regola CSS anche complessa

# Le funzioni per la manipolazione del documento

## Es: Modificare l'HTML del documento

```
var testo = document.getElementById('articolo');  
testo.innerHTML = 'Scrivo codice <b>HTML</b>...';
```

## Es: Modificare il valore di un attributo

```
document.getElementById('miafoto').src = 'nuovafoto.jpg';
```

## Es: Modificare lo stile CSS di un elemento

```
// imposto il colore rosso per il testo  
document.getElementById('articolo').style.color = '#FF0000';
```

# Le funzioni per la manipolazione del documento

Es: Gestire gli eventi utilizzando il DOM



```
//definisco l'evento direttamente nel codice javascript
document.getElementById('miafoto').onclick = function(){ ... };

// lo assegno una funzione esterna
foto.addEventListener('click', miafunzione);

// definisco internamente al metodo le istruzioni da eseguire
document.getElementById('miafoto').addEventListener('click', function() { alert('...'); });
```



# Le funzioni per la manipolazione del documento

```
// creo il nuovo elemento (un paragrafo)
var nuovo = document.createElement('p');

// creo un nuovo nodo di testo per "riempire" il nuovo elemento
var testo = nuovo.createTextNode('Nuovo testo da inserire');

// appendo il nodo di testo al nuovo elemento
nuovo.appendChild(testo);

// seleziono il nodo nel quale voglio aggiungere il nuovo elemento
var articolo = document.getElementById('articolo');

// ora aggiungo il nuovo elemento al nodo selezionato all'inizio
articolo.appendChild(nuovo);
```

# La Programmazione a Oggetti Javascript

- OOP
- Le Classi
- L'Oggetto (Istanza di Classe)
- Il Costruttore
- Proprietà e Metodi
- Ereditarietà
- Incapsulamento
- Polimorfismo

# Gli oggetti in Javascript: OOP

## OOP: Object Oriented Programming - Generalità

**Classe:** una struttura di dati complessa che può produrre ([istanziare](#)) **oggetti**, ognuno con stesse [proprietà](#), [eventi](#), [metodi](#)

**Oggetto:** istanza di una classe. Data una classe, questa è capace di produrre ([istanziare](#)) tanti nuovi oggetti

**Analogia per il concetto di classe** - [Catena di montaggio](#): è una serie di elementi meccanici, elettrici, elettronici in grado di produrre: [Automobili](#)

Ogni automobile rappresenta: [un oggetto](#)

Le [proprietà](#) sono caratteristiche intrinseche di un [oggetto](#)

Proprietà degli oggetti di classe [automobile](#):

[Marchio](#): “[Fiat](#)”, “[Volvo](#)” etc

[Colore](#): “[rosso](#)”, “[bianco](#)” etc

# OOP - generalità

La **OOP** è una riformulazione della programmazione procedurale strutturata e ci consente di organizzare al meglio il codice => **Riusabilità, Migliore organizzazione, Efficienza del codice**

Abbiamo già visto nei corsi di [Javascript / Jquery](#) parecchi esempi riguardo l'approccio **BOM** (Browser Object Model) e l'approccio **DOM** (Document Object Model)

Ma javascript è in grado di gestire anche la **OOP**, con la possibilità di **creare e gestire classi ed oggetti ad hoc**

## Programmazione Orientata agli Oggetti

- La programmazione orientata agli oggetti è un paradigma di programmazione che usa astrazione per creare modelli basati sul mondo reale
- Utilizza diverse tecniche di paradigmi esistenti, tra cui il polimorfismo e l'incapsulamento
- OOP ha una visione del software come un insieme di oggetti cooperanti, piuttosto che un insieme di funzioni o semplicemente un elenco di istruzioni per il computer come abbiamo visto fino ad ora con la programmazione procedurale/imperativa
- OOP promuove una maggiore flessibilità, manutenibilità e riusabilità del codice

## Le Classi

- Una classe definisce le caratteristiche dell'oggetto, essa è una definizione del modello, delle proprietà e dei metodi di un oggetto.
- Javascript nelle versioni precedenti ad ES6 utilizzava le funzioni come costruttori per le classi.
- Infatti definire una classe è facile come definire una funzione
- In ES5 per convenzione, una funzione costruttore della classe, avrà il nome con la prima lettera maiuscola, per distinguerla da una normale funzione
- In ES6 come abbiamo visto è stata introdotta la parola chiave **class** come già presente in altri linguaggi di programmazione



# Le Classi

La differenza tra ES5 ed ES6



```
/*
 * funzione costruttore della classe Persona
 * nella versione ES5 di javascript
*/
function Persona() {

}
```



```
/*
 * Classe Persona nella versione ES6 di javascript
 */
class Persona {
    constructor() {

    }
}
```

## Gli Oggetti (Istanze di Classe)

- Per creare una nuova istanza di una classe (ad esempio la classe Persona) e cioè un oggetto utilizziamo l'istruzione new Persona()
- Assegniamo il risultato dell'istanza ad una variabile per potervi accedere successivamente



## Gli Oggetti (Istanze di Classe)



```
let persona1 = new Persona();
let persona2 = new Persona();
```

## Il Costruttore

- Il costruttore è un metodo della classe
- Viene chiamato automaticamente quando viene creata l'istanza dell'oggetto
- Ha lo scopo di inizializzare il valore delle variabili di istanza(proprietà) o chiamare metodi che preparano l'oggetto per il suo uso

## Il Costruttore

- Come vediamo negli esempi nel costruttore vengono passati i valori Mario e Rossi che andranno a valorizzare le proprietà nome e cognome dell'oggetto



```
//ES5
function Persona(nome, cognome) {
    this.nome = nome
    this.cognome = cognome

    console.log("L'istanza è stata creata")
}

let persona1 = new Persona('Mario', 'Rossi')

console.log(persona1)
```



```
// ES6
class Persona {
    constructor(nome, cognome) {
        this.nome = nome;
        this.cognome = cognome;

        console.log("L'istanza è stata creata");
    }
}

let persona1 = new Persona('Mario', 'Rossi');

console.log(persona1);
```

## Le Proprietà (attributi dell'oggetto)

- Le proprietà sono variabili contenute nella classe
- Ogni istanza della classe ha queste proprietà, quindi ogni oggetto istanza di quella classe avrà quelle proprietà
- Le proprietà sono impostate nel costruttore della classe in modo che siano creati su ogni istanza.
- La parola chiave `this`, si riferisce all'oggetto corrente, ci permette di lavorare con le proprietà all'interno della classe
- Per accedere ad una proprietà in lettura o scrittura una volta che l'oggetto è stato istanziato si usa la dot notation



## Le Proprietà (attributi dell'oggetto)

- In questo esempio accediamo alla proprietà nome in lettura alla riga 12
- Alla riga 13 accediamo alla proprietà nome in scrittura e riassegniamo il valore di essa

```
● ● ●  
1 class Persona {  
2   constructor(nome, cognome) {  
3     this.nome = nome;  
4     this.cognome = cognome;  
5  
6     console.log("L'istanza è stata creata");  
7   }  
8 }  
9  
10 let persona1 = new Persona('Mario', 'Rossi');  
11  
12 console.log(persona1.nome); // >> Mario  
13 persona1.nome = 'Giorgio';  
14 console.log(persona1.nome); // >> Giorgio
```

## I metodi

- I metodi sono funzioni contenute nella classe
- Come le proprietà i metodi sono disponibili in ogni istanza, perciò ogni oggetto istanza di quella classe avrà a disposizione quei metodi
- Per accedere a un metodo come con le proprietà si usa la dot notation
- In ES5 per definire un metodo va assegnata una funzione al prototype della classe
- Mentre in ES6 basta definire la funzione all'interno della classe omettendo la parola chiave function



## I metodi

```
1 // ES5
2 function Persona(nome, cognome) {
3     this.nome = nome;
4     this.cognome = cognome;
5
6     Persona.prototype.saluta = function() {
7         console.log('Ciao');
8     }
9 }
10
11 let persona1 = new Persona('Mario', 'Rossi');
12
13 persona1.saluta(); // >> Ciao
```

```
1 // ES6
2 class Persona {
3     constructor(nome, cognome) {
4         this.nome = nome;
5         this.cognome = cognome;
6     }
7
8     saluta() {
9         console.log('Ciao');
10    }
11 }
12
13 let persona1 = new Persona('Mario', 'Rossi');
14
15 persona1.saluta(); // >> Ciao
```



## L'ereditarietà

- L'ereditarietà è un modo per creare una classe come una versione specializzata di una o più classi
- La classe specializzata viene comunemente chiamata *figlio*, e l'altra classe viene comunemente chiamato *padre*
- ES5 fornisce una modalità per creare una classe figlia, più prolissa e apparentemente più complicata
- ES6 grazie alla nuova sintassi semplifica di molto questo processo



## L'ereditarietà in ES5



```
1 function Persona(nome, cognome) {
2     this.nome = nome;
3     this.cognome = cognome;
4 }
5
6 function Docente(nome, cognome, materie) {
7     // chiamiamo il costruttore padre
8     Persona.call(this, nome, cognome);
9     this.materie = materie || [];
10}
11
12 // creiamo un Docente.prototype che erediti tutto ciò che ha Persona.prototype
13 Docente.prototype = Object.create(Persona.prototype);
14 // Assegniamo il metodo costruttore Docente
15 Docente.prototype.constructor = Docente;
16
17 var docenteJS = new Docente('Ryan', 'Burgos', ['Javascript OOP']);
18
19 console.log(docenteJS instanceof Docente) // >> true
20 console.log(docenteJS instanceof Persona) // >> true
```



## L'ereditarietà in ES6

```
1 class Persona {  
2     constructor(nome, cognome) {  
3         this.nome = nome;  
4         this.cognome = cognome;  
5     }  
6 }  
7 /*  
8 * la parola chiave extends ci permette di estendere la classe Persona  
9 * e quindi ereditare tutte le caratteristiche presenti nella classe Persona  
10 */  
11 class Docente extends Persona {  
12     constructor(nome, cognome, materie) {  
13         super(nome, cognome);  
14         this.materie = materie || [];  
15     }  
16 }  
17  
18 let docenteJS = new Docente('Ryan', 'Burgos', ['Javascript OOP']);  
19  
20 console.log(docenteJS instanceof Docente) // >> true  
21 console.log(docenteJS instanceof Persona) // >> true
```

## L'ereditarietà

- Javascript confonde un po' gli sviluppatori che hanno esperienza di linguaggi basati sulle classi (come Java o C++)
- Javascript è un linguaggio dinamico e non fornisce un'implementazione di class (la keyword **class** è introdotto in ES6, ma è zucchero sintattico, Javascript rimarrà basato sui prototipi).
- Ogni oggetto ha un link interno ad un altro oggetto chiamato **prototype**. Questo oggetto prototype ha a sua volta un suo prototype, e così via finché si raggiunge un oggetto con property null.null, per definizione, non ha un prototype, ed agisce come link finale nella **catena di prototipi**.
- Quasi tutti gli oggetti in Javascript sono istanze di Object, che risiede in cima alla catena dei prototipi.



## Incapsulamento

- **Incapsulamento** è la capacità di concentrare in un'unica entità (oggetto) dati e funzionalità
- Il nascondere informazioni (information hiding) è una caratteristica comune ad altre linguaggi, spesso in forma di metodi/proprietà private e protette

```
 1 class Persona {  
 2     constructor(nome, cognome) {  
 3         this.nome = nome;  
 4         this.cognome = cognome;  
 5     }  
 6     saluta() {  
 7         console.log(`Ciao a tutti mi chiamo ${this.nome} ${this.cognome}`);  
 8     }  
 9 }  
10 /*  
11 * la parola chiave extends ci permette di estendere la classe Persona  
12 * e quindi ereditare tutte le caratteristiche presenti nella classe Persona  
13 **/  
14 class Docente extends Persona {  
15     constructor(nome, cognome, materie) {  
16         super(nome, cognome);  
17         this.materie = materie || [];  
18     }  
19 }  
20  
21 let docenteJS = new Docente('Ryan', 'Burgos', ['Javascript OOP']);  
22  
23 console.log(docenteJS instanceof Docente) // >> true  
24 console.log(docenteJS instanceof Persona) // >> true
```

## Polimorfismo

- Polimorfismo significa letteralmente, che un oggetto ha la possibilità di assumere più forme
- Riferendoci ad un sistema software ad oggetti, il polimorfismo indicherà l'attitudine di un oggetto a mostrare più implementazioni per una singola funzionalità.
- Supponiamo di scrivere una classe FiguraGeometrica che ha come metodo calcolaArea(), inoltre supponiamo di scrivere due classi figlie Cerchio e Rettangolo, che ereditano il metodo calcolaArea(), ma chiaramente hanno un'implementazione diversa tra loro
- Uno dei maggiori benefici del polimorfismo, come in effetti di un po' tutti gli altri principi della programmazione ad oggetti, è la facilità di manutenzione del codice



## Polimorfismo



```
1 class FormaGeometrica {  
2     constructor(altezza, larghezza) {  
3         this.altezza = altezza;  
4         this.larghezza = larghezza;  
5     }  
6     calcolaArea() {}  
7 }
```



```
1 class Cerchio extends FormaGeometrica {  
2     constructor(altezza, larghezza, raggio) {  
3         super(altezza, larghezza);  
4         this.raggio = raggio;  
5     }  
6     calcolaArea() {  
7         return this.raggio * Math.PI;  
8     }  
9 }  
10  
11 class Rettangolo extends FormaGeometrica {  
12     constructor(altezza, larghezza) {  
13         super(altezza, larghezza);  
14     }  
15     calcolaArea() {  
16         return this.altezza * this.larghezza;  
17     }  
18 }  
19  
20 var rectangle = new Rettangolo(100, 200);  
21 console.log(rectangle.calcolaArea());  
22  
23 var circle = new Cerchio(100, 100, 50);  
24 console.log(circle.calcolaArea());
```



## Sezione 3

- Installazione di Node.js
- Introduzione a TypeScript
- I tipi di dato
  - string, number, boolean, null, undefined, Array
  - Tuple
  - Enum
  - Any
  - Void
- Le funzioni
- Interfacce
- OOP in TypeScript
  - Ambiti di visibilità: public, private, protected, static
  - Metodi accessori get e set
- I moduli in typescript
- Le parole chiave import, export, default in typescript

# Installazione di Node

- Requisito fondamentale del corso, avere installato sul vostro PC o MAC **Node.js** scaricabile dal sito ufficiale <https://nodejs.org>, seguiamo i seguenti passi
- Una volta scaricato il pacchetto di installazione, avviare il tutto con un click
- (Attenzione: solo per utenti MAC, se cliccando sul pacchetto installato vi darà questo messaggio di errore “**Impossibile aprire “node-v12.16.1.pkg” perché Apple non può verificare la presenza di malware.**” andate su Preferenze di Sistema -> Sicurezza e Privacy -> Tab Generali -> nella sezione in basso trovate il bottone Apri comunque
- Una volta apparso il wizard di installazione, cliccare sempre su Continua/Avanti fino alla fine all’ultimo step di installazione
- Una volta finito il processo di installazione tramite wizard avviare il prompt dei comandi(utenti windows)/console(utenti MAC) e digitare il comando **node -v** che dovrebbe restituirci il numero della versione installata di node
- Di seguito controlliamo anche che il package manager che ci servirà successivamente per installare TypeScript, sia stato installato correttamente digitando **npm -v** che come per il comando precedente restituirà il numero della versione

## Installazione di TypeScript

- Una volta installato node correttamente avviamo il prompt dei comandi(windows)/console(mac)
- Per utenti WINDOWS digitare **npm install -g typescript** e parirà l'installazione
- Per utenti MAC digitare **sudo npm install -g typescript** vi chiederà la password, inserite la password di sistema e premete invio per avviare l'installazione
- Una volta completata l'installazione digitare sempre su prompt o console **tsc -v** che vi stamperà la versione di typescript

## Introduzione a Typescript

- Typescript nasce con la missione di fornire a Javascript il supporto se pur opzionale alla tipizzazione stretta, come già presente nei linguaggi fortemente tipizzati come Java o C++
- Typescript è quindi un superset di Javascript, nato in Microsoft nel 2012 per aggiungere il controllo statico dei tipi e altre funzionalità
- Nasce con l'esigenza di scrivere del codice lato Frontend più robusto, infatti ad oggi i più moderni framework e librerie come Angular e React utilizzano typescript come linguaggio
- Un'applicazione Typescript viene tradotta dal compilatore(transpiler) in un'applicazione Javascript eseguibile in un qualsiasi engine (nel nostro caso il browser o node)
- L'enorme vantaggio di Typescript è rappresentato dal fatto che già in fase di transpilazione verso Javascript, il transpiler ci segnala già eventuali errori che potrebbero sfuggirci nel normale sviluppo in Javascript.
- <http://www.typescriptlang.org/docs/handbook/basic-types.html>

## Dichiarazione di variabili

- Le variabili in Typescript si possono dichiarare esattamente come in javascript con le parole chiave **var**, **let** e **const**
- Il vero vantaggio che ci offre Typescript se pur opzionale è di dichiarare il tipo della variabile usando i due punti seguiti dal tipo



```
1 let laMiaVariabile: number = 12345;
```



## String, Number e Boolean

- Come vediamo nelle immagini a seguito del nome della variabile andiamo ad inserire i due punti (:), seguiti poi dai tipi che abbiamo già visto in javascript
- Quindi andremo a scrivere chiaramente:
  - string per le stringhe
  - number per i numeri
  - boolean per i booleani



```
1 let myString: string = 'ciao';
```



```
1 let myNumber: number = 1234;
```



```
1 let myBoolean: boolean = true;
```

## Array

- Nel caso degli array, abbiamo due modi per dichiararli
  - La prima è dichiarare il tipo di dato seguito da parentesi quadre



```
1 let myArray: number[] = [1,2,3,4];
```

- Il secondo è utilizzare la parola chiave Array<tipo di dato> combinato con il tipo di dato



```
1 let myArray: Array<number> = [1,2,3,4];
```

## Tuple

- Typescript introduce un nuovo tipo di dato chiamato **Tuple**
- Le tuple consentono:
  - Di definire array con diversi tipi di dati tra loro
  - fissare il tipo di un numero di elementi (per es. i primi 2). Questo aspetto è importante, perché consente di definire uno schema preciso.



```
1 var persona: [string, number] = ["Ryan", 24];
```

- La tupla pur avendo un schema preciso di dati da rispettare, non ha un grandezza fissa, ciò vuol dire che può avere più elementi



## Enum

- L'enum è un tipo di dato che ci permette di definire un set di costanti
- Il valore delle costanti è sequenziale e di default parte da 0, quindi nell'esempio qui accanto
  - Up = 0
  - Down = 1
  - Left = 2
  - Right =3



```
1 enum Direction {  
2     Up,  
3     Down,  
4     Left,  
5     Right,  
6 }
```



## Enum

- In questo caso l'enum partirà da 1
  - Up = 1
  - Down = 2
  - Left = 3
  - Right = 4



```
1 enum Direction {  
2     Up = 1,  
3     Down,  
4     Left,  
5     Right,  
6 }
```



## Enum

- L'enum inoltre possono avere anche valori predefiniti di tipo stringa oltre che numerici



```
1 enum Direction {  
2     Up = "SOPRA",  
3     Down = "SOTTO",  
4     Left = "SINISTRA",  
5     Right = "DESTRA",  
6 }
```



```
1 enum Direction {  
2     Up = 29,  
3     Down = 12,  
4     Left = 93,  
5     Right = 44  
6 }
```

# Any

- Quando dichiariamo una variabile in JavaScript stiamo implicitamente affermando che **la variabile potrà contenere qualsiasi tipo di dato**. Questa assunzione implicita in JavaScript può essere resa esplicitamente in TypeScript specificando il tipo **any**.
- Il tipo any può essere utile quando chiamiamo una risorsa dal backend di cui non ne conosciamo il tipo



```
1 let list: any[] = [1, true, "free"];
```

## Void

- Il tipo void indica l'assenza di valore e normalmente indica che una funzione non torna nessun valore



```
1 function voidFunction():void {  
2     console.log("test");  
3 }
```

## Le funzioni

- Le funzioni come abbiamo visto in Javascript sono fondamentali per la costruzione di applicazioni
- Fino ad ora siamo stati abituati a scrivere le funzioni in questo modo



```
1 function addizione(a, b) {  
2     return a+b;  
3 }
```

## Le funzioni

- In typescript per sfruttare al 100% le sue potenzialità dovremo abituarci a specificare non solo il tipo di ritorno, ma anche il tipo degli argomenti



```
1 function addizione(a: number, b: number): number {  
2     return a+b;  
3 }
```

## Interfacce

- L'interfaccia in Typescript consente di definire una struttura per semplificare la tipizzazione dei parametri passati a una funzione

```
● ● ●

1 interface Persona {
2   nome: string;
3   cognome: string;
4   eta: number;
5 }
6
7
8 function stampaDati(p:Persona): string {
9   return `Il mio nome è ${p.nome} ${p.cognome} e ho ${p.eta} anni`;
10 }
```

## Interfacce

- I campi di un'interfaccia possono essere anche opzionali mettendo ? Dopo il nome della proprietà

```
1 interface Persona {  
2   nome: string;  
3   cognome?: string;  
4   eta?: number;  
5 }
```



- Le classi in Typescript si presentano in una forma simile a quella della versione di Javascript ES6
- Con la differenza che questa volta i campi sono tipizzati

## Le Classi

```
● ● ●  
1 class Persona {  
2   nome: string;  
3   cognome: string;  
4   eta: number;  
5  
6   constructor(nome: string, cognome: string, eta: number) {  
7     this.nome = nome;  
8     this.cognome = cognome;  
9     this.eta = eta;  
10    }  
11 }
```



## Gli ambiti di visibilità

- In Typescript come in altri linguaggi di programmazione orientati agli oggetti abbiamo a disposizione delle parole chiave per nascondere o rendere possibile la visibilità delle proprietà e dei metodi:
  - public
  - private
  - protected
  - static



## public

- In alcuni linguaggi è obbligatorio specificare che la proprietà sia public per poterla vedere, mentre in typescript le proprietà e i metodi sono **public** di default

```
 1 class Persona {  
 2     public nome: string;  
 3     public cognome: string;  
 4     public eta: number;  
 5  
 6     public constructor(nome: string, cognome: string, eta: number) {  
 7         this.nome = nome;  
 8         this.cognome = cognome;  
 9         this.eta = eta;  
10    }  
11 }
```



- TypeScript a differenza di javascript permette il property hiding, una delle caratteristiche dell'incapsulamento trattato nella lezione precedente
- Ci permette di impedire l'accesso alle proprietà della classe
- Neanche le classi figlie possono accedere alla proprietà privata

## private



```
1 class Persona {  
2     private nome: string;  
3  
4  
5     public constructor(nome: string) {  
6         this.nome = nome;  
7     }  
8 }
```



## protected

- protected si comporta in modo molto simile al modificatore privato, ad eccezione del fatto che è possibile accedere ai membri dichiarati protetti all'interno delle classi derivate.

```
1 class Persona {  
2   protected nome: string;  
3  
4   constructor(nome: string) {  
5     this.nome = nome;  
6   }  
7 }
```

## static

- Le proprietà e i metodi static sono disponibili anche senza istanziare la classe



```
1 class Persona {  
2   static unSaluto: string = "ciao a tutti";  
3  
4   static saluto() {  
5     console.log('buongiorno')  
6   }  
7 }  
8  
9 console.log(Persona.unSaluto) // >> ciao a tutti  
10  
11 Persona.saluto() // >> buongiorno
```

# Le Promise

- Le Promise
  - Struttura di una promise
  - Gestione delle funzione di callback
  - Promise.All
  - Promise.race

## Moduli

- A partire da ECMAScript 2015 o ES6, JavaScript ha un concetto di moduli.
- TypeScript condivide questo concetto. I moduli sono eseguiti nel loro scope, non nello scope globale; ciò significa che variabili, funzioni, classi, ecc. dichiarate in un modulo non sono visibili all'esterno del modulo a meno che non siano esplicitamente esportate usando la parola chiave **export**.
- Al contrario, per consumare una variabile, funzione, classe, interfaccia, ecc. Esportata da un modulo diverso, deve essere importata utilizzando la parola chiave **import**.



## export

- Qualsiasi dichiarazione (come una variabile, funzione, classe, tipo alias o interfaccia) può essere esportata aggiungendo la parola chiave **export**

```
1 // file Persona.ts
2
3 export class Persona {
4     nome: string;
5     constructor(nome: string) {
6         this.nome = nome;
7     }
8 }
```

# import

- L'importazione è semplice tanto quanto l'esportazione da un modulo. L'importazione di una dichiarazione esportata viene effettuata utilizzando la parola chiave **import**

```
1 // file app.ts
2 import { Persona } from './Persona.ts'
3
4 let p = new Persona('Ryan');
```

## Promise

- Una promessa è un oggetto che rappresenta l'eventuale completamento o errore di un'operazione asincrona.
- Poiché molte persone sono consumatori di Promise già create
- In sostanza, una Promise è un oggetto restituito a cui “attaccare” callback, invece di passargliela.



## La struttura di una Promise



```
1 const myPromise = new Promise((resolve, reject) => {  
2     if(true) {  
3         resolve('resolved');  
4     }  
5     reject('rejected');  
6 })
```

## “Attacarsi” a Promise

- l'oggetto Promise una volta instanziato ci mette a disposizione due metodi
- then che accetta come argomento una funzione e viene eseguita solo nel caso in cui la promise è risolta
- catch che accetta come argomento una funzione e viene eseguita solo nel caso in cui la promise è rigetta



## “Attacarsi” a Promise



```
1 myPromise.then((message) => {  
2     console.log(message);  
3 }).catch((errMessage) => {  
4     console.log(errMessage);  
5 })
```

## Gestire più Promise con Promise.all

- Promise.all è un metodo statico che ci mette a disposizione la classe Promise
- Essa ci permette di gestire più promise insieme
- Accetta come argomento un array di promise
- Come una promise normale espone i metodi then e catch
- E torna un array di risposte



## Gestire più Promise con Promise.all



```
1 const promise1 = new Promise((resolve, reject) => { resolve('promise1') });
2
3 const promise2 = new Promise((resolve, reject) => { resolve('promise2') })
4
5
6 Promise.all([ promise1, promise2 ]).then(promises => promises);
```

## Gestire più Promise con Promise.race

- Promise.race come Promise.all è un metodo statico che ci mette a disposizione la classe Promise
- Essa ci permette di gestire più promise insieme
- Accetta come argomento un array di promise
- Come una promise normale espone i metodi then e catch
- E torna la promise che risponde per prima



## Gestire più Promise con Promise.race

● ● ●

```
1 const promise1 = new Promise((resolve, reject) => {
2   setTimeout(() => {
3     resolve('promise1');
4   }, 4000);
5 });
6
7 const promise2 = new Promise((resolve, reject) => {
8   setTimeout(() => {
9     resolve('promise2');
10  }, 2000); })
11
12
13 Promise.race([ promise1, promise2 ]).then(promises => console.log(promises)); //promise2
```