



JAVASCRIPT → PARTE 2 CC BY

SOTTO-PROGRAMMI

 Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

 In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {  
  // istruzione/i  
  [return (<valore>);]  
}
```

```
function stampa5Naturali() {  
  for (var i=0; i<5; i++) {  
    document.write(i);  
  }  
}  
  
stampa5Naturali();
```

01234

VERSIONE 2.8 - DIAPOSITIVA 2 ALESSANDRO URSOMANDO

SOTTO-PROGRAMMI

Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {
  // istruzione/i
  [return (<valore>);]
}
```

```
function stampa5Naturali() {
  for (var i=0; i<5; i++) {
    document.write(i);
  }
}
```

```
document.write("(");
stampa5Naturali();
document.write(")");
```

SOTTO-PROGRAMMI

Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {
  // istruzione/i
  [return (<valore>);]
}
```

```
function stampa5Naturali() {
  for (var i=0; i<5; i++) {
    document.write(i);
  }
}
```

```
document.write("(");
stampa5Naturali();
document.write(" - ");
stampa5Naturali();
document.write(")");
```

SOTTO-PROGRAMMI

Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {
  // istruzione/i
  [return (<valore>);]
}
```

```
function stampa5Naturali() {
  for (var i=0; i<5; i++) {
    document.write(i);
  }
}
```

```
for (var i=1; i<=3; i++) {
  document.write(" ");
  stampa5Naturali();
  document.write(" ");
  document.write("<br/>");
}
```

```
(01234)
(01234)
(01234)
```

SOTTO-PROGRAMMI

Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {
  // istruzione/i
  [return (<valore>);]
}
```

```
function stampaRigaTrattini(n) {
  for (var i=1; i<=n; i++)
    document.write("-");
  document.write("<br/>");
}
```

```
stampaRigaTrattini(10);
```

```
-----
```

SOTTO-PROGRAMMI

Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

```
function stampaRigaTrattini(n) {
  for (var i=1; i<=n; i++)
    document.write("-");
  document.write("<br/>");
}
```

```
var x = Math.floor(Math.random()*20)+1;
stampaRigaTrattini(x);
```

.....

In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {
  // istruzione/i
  [return (<valore>);]
}
```

SOTTO-PROGRAMMI

Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

```
function stampaRigaTrattini(n) {
  for (var i=1; i<=n; i++)
    document.write("-");
  document.write("<br/>");
}
```

```
for (var i=20; i>0; i-=5)
  stampaRigaTrattini(i);
```

.....

In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {
  // istruzione/i
  [return (<valore>);]
}
```

SOTTO-PROGRAMMI

Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

```
function miaRandom(a,b) {
  return Math.floor(Math.random()*(b-a+1))+a;
}
```

```
document.write("<p>" + miaRandom(3,5) + "</p>");
```

In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {
  // istruzione/i
  [return (<valore>);]
}
```

SOTTO-PROGRAMMI

Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

```
function miaRandom(a,b) {
  return Math.floor(Math.random()*(b-a+1))+a;
}
```

```
var x = 3;
var y = 5;
document.write("<p>" + miaRandom(x,y) + "</p>");
```

In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {
  // istruzione/i
  [return (<valore>);]
}
```

In questi esempi il passaggio di parametri avviene sempre **per valore**.

SOTTO-PROGRAMMI

Un sotto-programma svolge in autonomia un compito ben preciso più o meno circoscritto; è caratterizzato da un nome; può avere parametri in input; può restituire un valore in output.

In Javascript i sotto-programmi si chiamano **funzioni** e seguono la seguente sintassi.

```
function nomeFunzione([parametri]) {
  // istruzione/i
  [return (<valore>);]
}
```

```
function miaRandom(a,b) {
  return Math.floor(Math.random()*(b-a+1))+a;
}
```

```
var x = prompt("Inserisci un numero");
x = Number(x);
var y = prompt("Inserisci un numero");
y = Number(y);
var z = miaRandom( x, (y+x) );
document.write("<p>" + z + "</p>");
```

ESEMPI

```

/*****
 *                               Sezione delle funzioni                               *
 *****/
function isIntero(num) {
  if (
    (num==null) // verifico che non abbia annullato
    || (num=="") // verifico che abbia scritto qualcosa
    || isNaN(num) // verifico che sia un numero
    || ( Number(num) != Math.floor( Number(num) ) ) // verifico che sia intero
  )
    return false;
  else
    return true;
}

/*****
 *                               Sezione codice da eseguire                               *
 *****/
do
  var x = prompt("Introduci un numero intero:");
while ( !isIntero(x) );

```

ESEMPI

```

/*****
 *
 *           Sezione delle funzioni
 *
 *****/
function isIntero(num) {
  if (
    (num==null)                // verifico che non abbia annullato
    || (num=="")              // verifico che abbia scritto qualcosa
    || isNaN(num)             // verifico che sia un numero
    || ((Number(num)%1)!=0)   // verifico che sia intero
  )
    return false;
  else
    return true;
}

/*****
 *
 *           Sezione codice da eseguire
 *
 *****/
do
  var x = prompt("Introduci un numero intero:");
while ( !isIntero(x) );

```

ESERCIZIO 1.1-00-04-01

ESEMPI

```

/*****
 *
 *           Sezione delle funzioni
 *
 *****/
function isIntero(num) {
  /* omissis */
}

function isInteroInRange(num, inizioRange, fineRange) {
  return (
    (isIntero(num)) &&
    (Number(num)>=inizioRange) &&
    (Number(num)<=fineRange)
  );
}

function isInteroTreCifre(num) {
  return isInteroInRange(num,100,999);
}

/*****
 *
 *           Sezione codice da eseguire
 *
 *****/
do
  var x = prompt("Introduci un numero intero a 3 cifre");
while ( !isInteroTreCifre(x) );

```

ESERCIZIO 1.1-00-04-02

ESERCIZI

PASSAGGIO DI PARAMETRI

Negli esempi e negli esercizi sopra riportati
il passaggio di parametri è sempre avvenuto **per valore**



Il codice che invoca la funzione le
passa dei valori.
La funzione archivia questi valori in
una variabile propria non visibile
all'esterno.

PASSAGGIO PER VALORE

Il codice che invoca la funzione le
fornisce l'indirizzo di una sua
variabile. La funzione – tramite
tale indirizzo – lavora direttamente
su quella variabile.

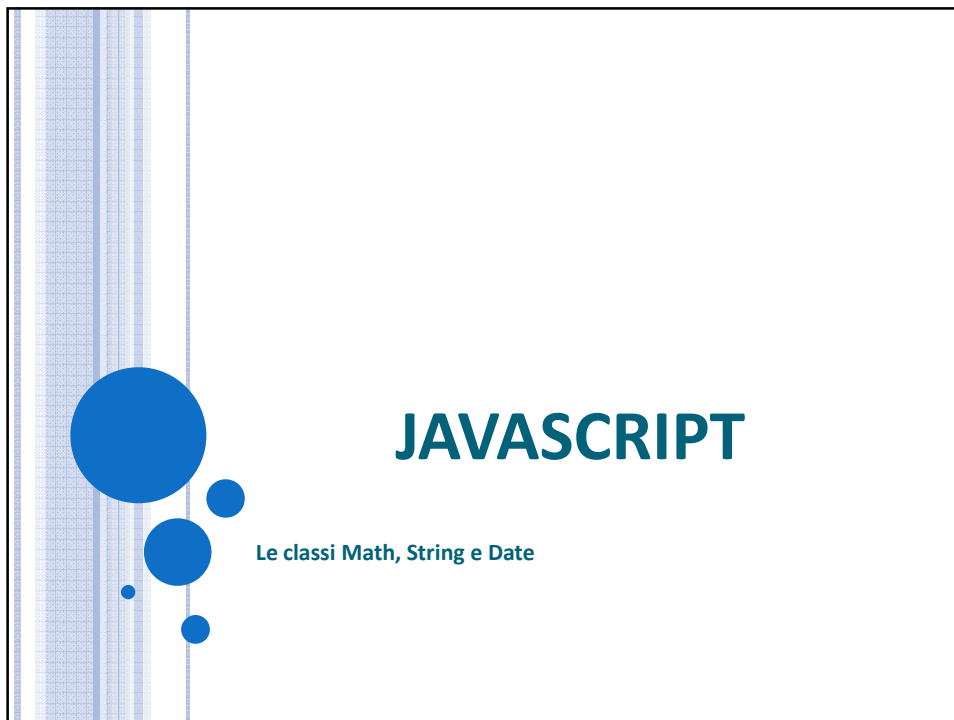
PASSAGGIO PER INDIRIZZO

In Javascript il passaggio per indirizzo coinvolge il concetto di **oggetto**.



Abbiamo tutti il concetto di **oggetto**?





JAVASCRIPT → PARTE 2 → OOP CC BY

INTRODUZIONE

Javascript è un linguaggio OOP e pertanto:

- dispone di classi predefinite
- permette di sviluppare classi personalizzate

Noi ci occuperemo solo di **alcune** di quelle predefinite.

- Math
- String
- Date

Studieremo inoltre la classe Array.

VERSIONE 2.8 - DIAPOSITIVA 18 ALESSANDRO URSOMANDO

LA CLASSE MATH

La classe **Math** l'abbiamo già incontrata. Ecco di nuovo le **proprietà** ed i **metodi** che abbiamo visto.

Si tratta di proprietà e metodi **statici**.

Ricordiamo che una proprietà o un metodo statico (o di classe) viene adoperato **senza istanziare** un oggetto della classe.

E	numero di Eulero (~2.718)
PI	il valore di π greco (~3.14)

abs(x)	valore assoluto
ceil(x)	arrotonda all'intero superiore
exp(x)	e elevato a x
floor(x)	arrotonda all'intero inferiore
log(x)	logaritmo in base e
max(x,y,z,...,n)	massimo
min(x,y,z,...,n)	minimo
pow(x,y)	x elevato a y
random()	casuale
round(x)	arrotonda
sqrt(x)	radice quadrata

LA CLASSE STRING

Il tipo base stringa è sufficiente in molti casi ma per fornire dei metodi per ispezionare e manipolare la stringa è necessaria una classe.

length	la lunghezza della stringa
---------------	----------------------------

charAt(i)	restituisce l'i-esimo carattere
indexOf(s)	r. dove comincia la sottostringa s
lastIndexOf(s)	r. l'ultima occorrenza di s
replace(s1, s2)	r. una stringa con s1 al posto di s2
substr(i)	estrae una sottostringa a partire da i
toLowerCase	restituisce la stringa minuscola
toUpperCase	restituisce la stringa maiuscola

LA CLASSE STRING

Poiché **String** non è una classe statica (come era **Math**), devo prima dichiarare un **oggetto** della classe **String** (mediante l'operatore **new**) e poi posso applicare ad esso i **metodi** della classe.

In realtà Javascript ci permette di usare di metodi della classe **String** **anche** per le stringhe dichiarate come tipo primitivo.

length	la lunghezza della stringa
charAt(i)	restituisce l'i-esimo carattere
indexOf(s)	r. dove comincia la sottostringa s
lastIndexOf(s)	r. l'ultima occorrenza di s
replace(s1, s2)	r. una stringa con s1 al posto di s2
substr(i)	estrae una sottostringa a partire da i
toLowerCase	restituisce la stringa minuscola
toUpperCase	restituisce la stringa maiuscola

```
var s = new String("Ale");
alert(s.length); // stampa 3
```

```
var t = "Alessandro";
alert(t.length); // stampa 10
```

ESEMPI

```

/*****
 *                               Sezione delle funzioni                               *
 *****/
function isUnaParola(s) {
  // verifico che il parametro esista
  if ( (s == null) || (s == "") )
    return false;

  // verifico che non contenga spazi o numeri
  for (var i=0; i<s.length; i++) {
    if (s.charAt(i) == ' ')
      return false;
    for (var j=0; j<=9; j++)
      if (s.charAt(i) == j)
        return false;
  }

  // se sto qui è una sola parola
  return true;
}

/*****
 *                               Sezione codice da eseguire                               *
 *****/
do
  var x = prompt("Introduci una parola con le maiuscole a caso.");
while ( !isUnaParola(x) );

```

ESERCIZIO 11-00-04-03

ESEMPI

```
/*
 * Sezione delle funzioni
 */
function isUnaParola(s) {
    /* omissis */
}

function primaLetteraMaiuscola(s) {
    s = s.toLowerCase();
    var primaLettera = s.charAt(0);
    primaLettera = primaLettera.toUpperCase();
    var tuttoIlResto = s.substr(1);
    return (primaLettera + tuttoIlResto);
}

/*
 * Sezione codice da eseguire
 */
do
    var x = prompt("Introduci una parola con le maiuscole a caso.");
while ( !isUnaParola(x) );
x = primaLetteraMaiuscola(x);
alert(x);
```

Osserviamo che anche in questi esempi il passaggio di parametri avviene **per valore**.



ESERCIZI

LA CLASSE DATE

La classe **Date** ci permette di instanziare oggetti che contengono una data (con un orario).

Utilizzando i costruttori e metodi della classe sarà **impossibile** creare una data non valida.

setDate(x)	imposta il giorno (1-31)
getDate()	lo restituisce
setMonth(x)	imposta il mese (0-11)
getMonth()	lo restituisce
setFullYear(x)	imposta l'anno (4 cifre)
getFullYear()	lo restituisce

getDay()	restituisce il giorno della settimana (0-6)
-----------------	---

setHours(x)	imposta l'ora (0-23)
getHours()	la restituisce
setMinutes(x)	imposta i minuti (0-59)
getMinutes()	li restituisce
setSeconds(x)	imposta i secondi (0-59)
getSeconds(x)	li restituisce

LA CLASSE DATE

Oltre a quanto indicato qui accanto sono implementati:

- o molti **costruttori**
- o alcuni **operatori**

```
var d1 = new Date(a,m,g);
var d2 = new Date();
var d3 = new Date(d1);
```

```
if (d1 > d2) {
  // fai qualcosa
}
```

setDate(x)	imposta il giorno (1-31)
getDate()	lo restituisce
setMonth(x)	imposta il mese (0-11)
getMonth()	lo restituisce
setFullYear(x)	imposta l'anno (4 cifre)
getFullYear()	lo restituisce

getDay()	restituisce il giorno della settimana (0-6)
-----------------	---

setHours(x)	imposta l'ora (0-23)
getHours()	la restituisce
setMinutes(x)	imposta i minuti (0-59)
getMinutes()	li restituisce
setSeconds(x)	imposta i secondi (0-59)
getSeconds(x)	li restituisce

ESEMPIO

```

/*****
 *           Sezione delle funzioni           *
 *****/
function getNomeGiorno(x) {
  switch (x) {
    case 0: return "Domenica";
    case 1: return "Lunedì";
    case 2: return "Martedì";
    case 3: return "Mercoledì";
    case 4: return "Giovedì";
    case 5: return "Venerdì";
    case 6: return "Sabato";
  }
}

/*****
 *           Sezione codice da eseguire       *
 *****/
var oggi = new Date();
var msg = "<h1>";
msg += getNomeGiorno(oggi.getDay()) + " ";
msg += oggi.getDate() + "/";
msg += oggi.getMonth() + "/";
msg += oggi.getYear();
msg += "</h1>";
document.write(msg);

```

ESEMPIO

```

/*****
 *           Sezione delle funzioni           *
 *****/
function convertiOrarioInStringa(orario) {
  var s = orario.getHours() + ":";
  if (orario.getMinutes() < 10)
    s += "0";
  s += orario.getMinutes();
  return (s);
}

/*****
 *           Sezione codice da eseguire       *
 *****/
var oggi = new Date();
document.write("<h1>" + oggi + "</h1>");
document.write("<h1>" + convertiOrarioInStringa(oggi) + "</h1>");

```

ESERCIZI



JAVASCRIPT

Funzioni:
il passaggio di parametri per riferimento.

PASSAGGIO DI PARAMETRI PER RIFERIMENTO

In passato abbiamo detto che in Javascript il passaggio per indirizzo coinvolge il concetto di **oggetto**.

Adesso che abbiamo usato la classe statica **Math**, la classe speciale **String** e la classe (normale) **Date** vogliamo approfondire questo concetto.

Il codice che invoca la funzione le passa dei valori. La funzione archivia questi valori in una variabile propria non visibile all'esterno.

Il codice che invoca la funzione le fornisce l'indirizzo di una sua variabile. La funzione – tramite tale indirizzo – lavora direttamente su quella variabile.

In Javascript il passaggio per indirizzo coinvolge il concetto di **oggetto**.

Abbiamo tutti il concetto di **oggetto**?

In generale, vale la regola seguente:

- un parametro di **tipo oggetto** è passato **per indirizzo**
- un parametro di **altro tipo** è passato **per valore**

PASSAGGIO DI PARAMETRI PER RIFERIMENTO

```
function CambiaIlValore(x) {
  x = 0;
}
/*****/
var num = 13;
CambiaIlValore(num);
alert(num); // stampa 13
```

```
function CambiaIlValore(x) {
  x = "Ciao";
}
/*****/
var str = "Ale";
CambiaIlValore(str);
alert(str); // stampa Ale
```

```
function CambiaIlValore(x) {
  x.setFullYear(2013);
}
/*****/
var nascita = new Date(1976,3,19);
CambiaIlValore(nascita);
alert(nascita); // stampa 19 Apr 13
```

```
function CambiaIlValore(x) {
  x = new Date();
}
/*****/
var nascita = new Date(1976,3,19);
CambiaIlValore(nascita);
alert(nascita); // stampa 19 Apr 76
```

In generale, vale la regola seguente:

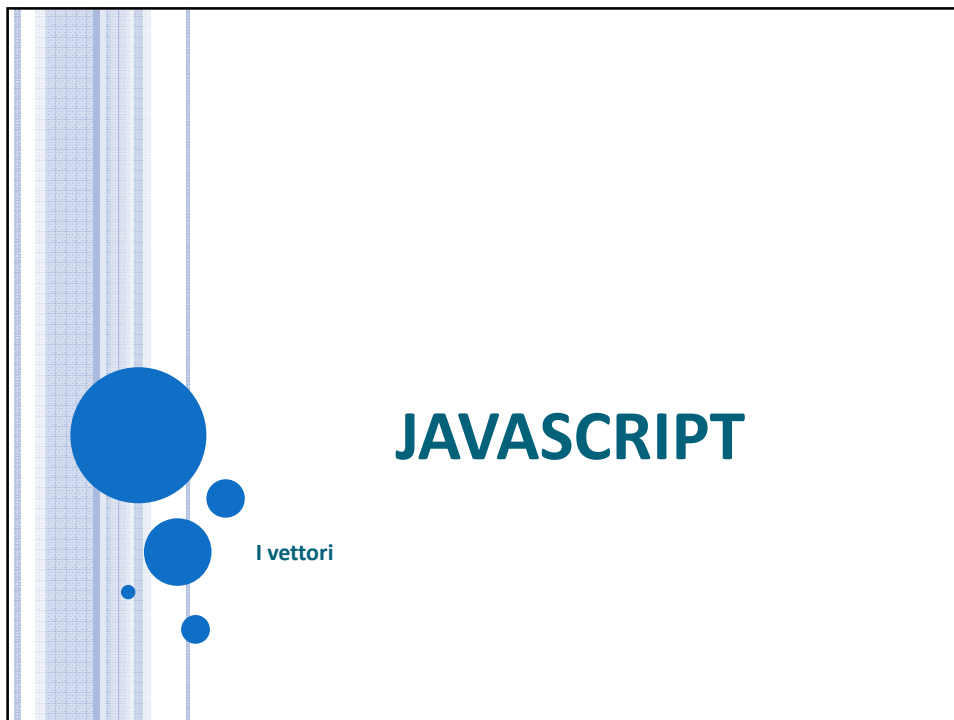
- un parametro di **tipo oggetto** è passato **per indirizzo**
- un parametro di **altro tipo** è passato **per valore**

Si faccia attenzione ad usare i metodi della classe per modificare l'oggetto.

ESEMPIO

```
/* *****  
 *           Sezione delle funzioni           *  
 * ***** */  
function setOrarioCasuale(orario) {  
    orario.setHours(Math.floor(Math.random()*24));  
    orario.setMinutes(Math.floor(Math.random()*60));  
    orario.setSeconds(0);  
}  
function convertiOrarioInStringa(orario) {  
    /* omissis */  
}  
  
/* *****  
 *           Sezione codice da eseguire       *  
 * ***** */  
var oggi = new Date();  
document.write("<h1>" + convertiOrarioInStringa(oggi) + "</h1>");  
setOrarioCasuale(oggi);  
document.write("<h1>" + convertiOrarioInStringa(oggi) + "</h1>");
```

ESERCIZI



JAVASCRIPT → PARTE 2 → VETTORI CC BY

LA CLASSE ARRAY

! La classe Array ci permette di instanziare oggetti di tipo vettore.

! Esistono diversi modi per dichiarare e inizializzare un vettore.

```
var v = new Array(n);
var v = new Array();
var v = new Array(1,2,3);
var v = [n];
var v = [];
var v = [1,2,3];
```

length	restituisce la dimensione
sort()	ordina il vettore
reverse()	inverte l'ordine degli elementi
indexOf(x)	restituisce l'indice dell'elemento x
lastIndexOf(x)	ce ne fossero più d'uno, dà l'ultimo
push(x)	inserisce x nel vettore (in coda)
pop()	toglie e restituisce l'ultimo elemento
shift()	toglie e restituisce il primo elemento

! Se la **sort** un giorno non dovesse comportarsi come ti aspetti chiamala così:

```
sort(function(a,b){return (b-a)});
```

VERSIONE 2.8 - DIAPOSITIVA 36
ALESSANDRO URSOMANDO

ESEMPI

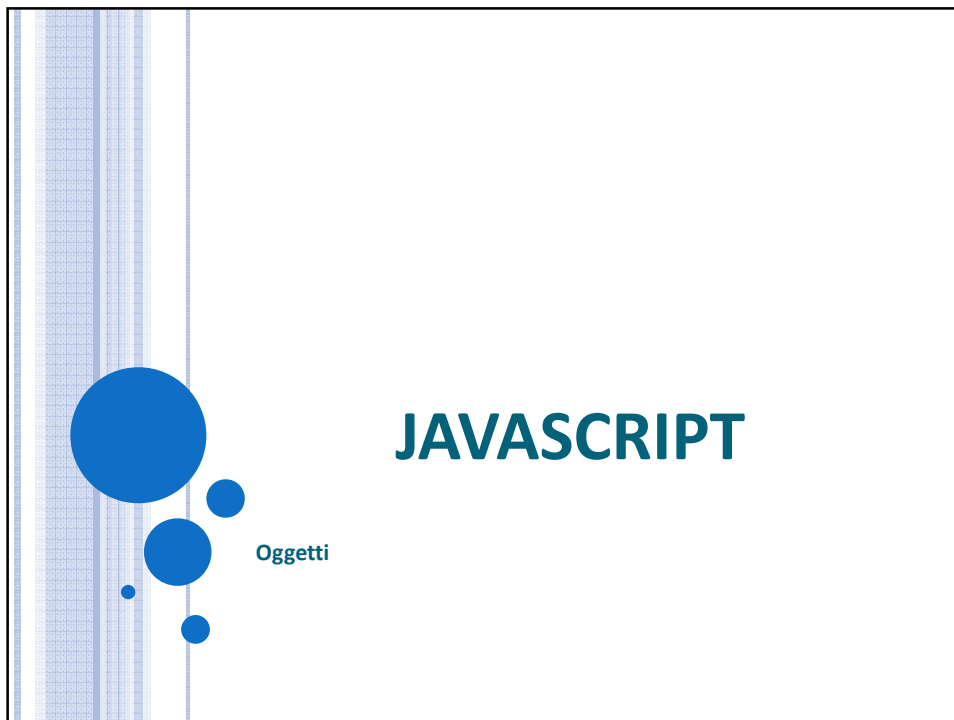
```
function stampaVettore(v) {  
  document.write("<table><tbody><tr>")  
  for (var i=0; i<10; i++) {  
    document.write("<td>"+v[i]+"</td>");  
  }  
  document.write("</tr></tbody></table>")  
}
```

```
// creo un vettore di dieci elementi  
var vett = new Array(10);  
for (var i=0; i<10; i++) {  
  vett[i] = Math.floor(Math.random()*100);  
}  
  
// lo stampo prima e dopo l'ordinamento  
stampaVettore(vett);  
vett.sort();  
stampaVettore(vett);
```

Ordinamento di un vettore


33	40	58	35	51	95	13	69	43	64
13	33	35	40	43	51	58	64	69	95

ESERCIZI




JAVASCRIPT → PARTE 2 → OGGETTI CC BY


JAVASCRIPT E LA OOP

Javascript **non supporta** la programmazione ad oggetti. 

↕

Javascript consente di realizzare qualcosa di **simile** ad un oggetto **in almeno tre modi diversi**. 

↕

Noi cercheremo di restare aderenti alla OOP **da manuale** e vedremo in che modo Javascript ci permette di realizzare quanto necessario. 

VERSIONE 2.8 - DIAPOSITIVA 40 ALESSANDRO URSOMANDO

DEFINIRE UNA CLASSE

La prima cosa un po' **strana** è che in Javascript non esiste un costrutto che identifichi una **classe** ma per definire una classe si usa il costrutto **function**.

```
function Punto () {  
  this.x = 0;  
  this.y = 0;  
}
```



IL COSTRUTTO THIS

La prima cosa un po' **strana** è che in javascript non esiste un costrutto che identifichi una **classe** ma per definire una classe si usa il costrutto **function**.

```
function Punto () {  
  this.x = 0;  
  this.y = 0;  
}
```



In questa funzione un po' speciale il costrutto **this** serve a definire come **propri** attributi e metodi della classe.



DOV'È IL MIO COSTRUTTORE?

La prima cosa un po' **strana** è che in javascript non esiste un costrutto che identifichi una **classe** ma per definire una classe si usa il costrutto **function**.

```
function Punto () {  
  this.x = 0;  
  this.y = 0;  
}
```

Il costruttore **coincide** con la classe.
Infatti nell'esempio assegniamo subito un valore ai due attributi.

In futuro vedremo come ottenere costruttori multipli.

INSTANZIARE UN OGGETTO

La prima cosa un po' **strana** è che in javascript non esiste un costrutto che identifichi una **classe** ma per definire una classe si usa il costrutto **function**.

```
function Punto () {  
  this.x = 0;  
  this.y = 0;  
}
```

Come abbiamo già visto, per istanziare un oggetto di una certa classe si usa l'operatore **new**.

```
var origine = new Punto();  
var p1 = new Punto();
```

ACCESSO A METODI E ATTRIBUTI

La prima cosa un po' **strana** è che in javascript non esiste un costrutto che identifichi una **classe** ma per definire una classe si usa il costrutto **function**.

```
function Punto () {  
  this.x = 0;  
  this.y = 0;  
}
```

Come abbiamo già visto, per accedere ad attributi (e metodi) si usa l'operatore **punto** (.) in questo modo:
<nomeOggetto>.<nomeAttributo>
<nomeOggetto>.<nomeMetodo>

Vediamo un esempio completo

ESEMPIO

```
// Definisco la classe Punto  
function Punto () {  
  this.x = 0;  
  this.y = 0;  
}  
  
// istanzio un oggetto della classe Punto  
var origine = new Punto();  
// ed accedo in lettura ai suoi attributi  
var msg = "<p>";  
msg += "L'oggetto origine vale: ";  
msg += "(" + origine.x + "," + origine.y + ")";  
msg += "</p>";  
document.write(msg);  
  
// istanzio un altro oggetto della classe Punto  
var p = new Punto();  
// ed accedo in scrittura ai suoi attributi  
p.x = 3;  
p.y = 5;
```

INCAPSULAMENTO

```
// Definisco la classe Punto
function Punto () {
  this.x = 0;
  this.y = 0;
}

// istanzio un oggetto della classe Punto
var origine = new Punto();
// ed accedo in lettura ai suoi attributi
var msg = "<p>";
msg += "L'oggetto origine vale: ";
msg += "(" + origine.x + "," + origine.y + ")";
msg += "</p>";
document.write(msg);

// istanzio un oggetto della classe Punto
var p = new Punto();
// ed accedo in scrittura ai suoi attributi
p.x = 3;
p.y = 5;
```

Questa modalità di utilizzo degli attributi non tiene conto della prima caratteristica della OOP: **l'incapsulamento**.

La classe deve essere creata in modo da mascherare la definizione dei suoi attributi alle varie istanze, permettendo loro di accedere ai dati (**privati**) solo mediante metodi (**pubblici**).

In questo caso: setX(x), getX(), setY(y) e getY().

ESERCIZIO 1.1-00-07-01

DEFINIRE UN METODO

```
function Punto () {
  // attributi
  this.x;
  this.y;

  // metodi
  this.setX = function(n) {
    this.x = n;
  }
  this.getX = function() {
    return this.x;
  }
  this.setY = function(n) {
    this.y = n;
  }
  this.getY = function() {
    return this.y;
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Questa modalità di utilizzo degli attributi non tiene conto della prima caratteristica della OOP: **l'incapsulamento**.

La classe deve essere creata in modo da mascherare la definizione dei suoi attributi alle varie istanze, permettendo loro di accedere ai dati (**privati**) solo mediante metodi (**pubblici**).

In questo caso: setX(x), getX(), setY(y) e getY().

ESERCIZIO 1.1-00-07-01

USO DELLE ISTANZE DI UNA CLASSE

```
function Punto () {
  // attributi
  this.x;
  this.y;

  // metodi
  this.setX = function(n) {
    this.x = n;
  }
  this.getX = function() {
    return this.x;
  }
  this.setY = function(n) {
    this.y = n;
  }
  this.getY = function() {
    return this.y;
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Definendo tutti gli attributi e tutti i metodi con il costrutto **this** li rendiamo tutti **pubblici**.

```
// istanzio un oggetto
var p = new Punto();
```

```
// invoco un metodo
p.setX(10);
```

```
// ma posso ancora
// modificare un attributo
// senza passare per un
// metodo della classe
p.y = "Alessandro";
```

No incapsulamento!

ESERCIZIO 1.1-00-07-01

PUBBLICO E PRIVATO

```
function Punto () {
  // attributi
  this.x;
  this.y;

  // metodi
  this.setX = function(n) {
    this.x = n;
  }
  this.getX = function() {
    return this.x;
  }
  this.setY = function(n) {
    this.y = n;
  }
  this.getY = function() {
    return this.y;
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Se il costrutto **this** definisce come **pubblici** attributi e metodi della classe, il costrutto **var** definisce come **privati** attributi e metodi della classe.

ESERCIZIO 1.1-00-07-05

PUBBLICO E PRIVATO

```
function Punto () {
  // attributi
  var this-x;
  var this-y;

  // metodi
  this.setX = function(n) {
    this-x = n;
  }
  this.getX = function() {
    return this-x;
  }
  this.setY = function(n) {
    this.y = n;
  }
  this.getY = function() {
    return this-y;
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Se il costrutto **this** definisce come **pubblici** attributi e metodi della classe, il costrutto **var** definisce come **privati** attributi e metodi della classe.

Riscriviamo tutto compattato..

PUBBLICO E PRIVATO

```
function Punto () {
  // attributi
  var x;
  var y;

  // metodi
  this.setX = function(n) { x = n; }
  this.getX = function() { return x; }
  this.setY = function(n) { y = n; }
  this.getY = function() { return y; }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Se il costrutto **this** definisce come **pubblici** attributi e metodi della classe, il costrutto **var** definisce come **privati** attributi e metodi della classe.

Riscriviamo tutto compattato..

..e inseriamo un **metodo privato**.

PUBBLICO E PRIVATO

```
function Punto () {
  // attributi
  var x;
  var y;

  // metodi
  this.setX = function(n) { x = n; }
  this.getX = function() { return x; }
  this.setY = function(n) { y = n; }
  this.getY = function() { return y; }
  var miaRandom = function() {
    return Math.floor(Math.random()*51)-25;
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Se il costrutto **this** definisce come **pubblici** attributi e metodi della classe, il costrutto **var** definisce come **privati** attributi e metodi della classe.

Riscriviamo tutto compattato..

..e inseriamo un **metodo privato**.

PUBBLICO E PRIVATO

```
function Punto () {
  // attributi
  var x;
  var y;

  // metodi
  this.setX = function(n) { x = n; }
  this.getX = function() { return x; }
  this.setY = function(n) { y = n; }
  this.getY = function() { return y; }
  var miaRandom = function() {
    return Math.floor(Math.random()*51)-25;
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Se il costrutto **this** definisce come **pubblici** attributi e metodi della classe, il costrutto **var** definisce come **privati** attributi e metodi della classe.

Usiamo questo **metodo privato** in un **metodo pubblico**.

PUBBLICO E PRIVATO

```
function Punto () {
  // attributi
  var x;
  var y;

  // metodi
  this.setX = function(n) { x = n; }
  this.getX = function() { return x; }
  this.setY = function(n) { y = n; }
  this.getY = function() { return y; }
  var miaRandom = function() {
    return Math.floor(Math.random()*51)-25;
  }
  this.casuale = function() {
    this.setX(miaRandom());
    this.setY(miaRandom());
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Se il costrutto **this** definisce come **pubblici** attributi e metodi della classe, il costrutto **var** definisce come **privati** attributi e metodi della classe.

Usiamo questo **metodo privato** in un **metodo pubblico**.

ESERCIZIO 1.1-00-07-06

PUBBLICO E PRIVATO

```
function Punto () {
  // attributi
  var x;
  var y;

  // metodi
  this.setX = function(n) { x = n; }
  this.getX = function() { return x; }
  this.setY = function(n) { y = n; }
  this.getY = function() { return y; }
  var miaRandom = function() {
    return Math.floor(Math.random()*51)-25;
  }
  this.casuale = function() {
    this.setX(miaRandom());
    this.setY(miaRandom());
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Se il costrutto **this** definisce come **pubblici** attributi e metodi della classe, il costrutto **var** definisce come **privati** attributi e metodi della classe.

```
// istanzio un oggetto
var p = new Punto();
```

```
// invoco un metodo pubblico
p.setX(10);
```

```
// invoco un metodo privato
alert(p.miaRandom());
// errore
```

ESERCIZIO 1.1-00-07-06

PUBBLICO E PRIVATO

```
function Punto () {
  // attributi
  var x;
  var y;

  // metodi
  this.setX = function(n) { x = n; }
  this.getX = function() { return x; }
  this.setY = function(n) { y = n; }
  this.getY = function() { return y; }
  var miaRandom = function() {
    return Math.floor(Math.random()*51)-25;
  }
  this.casuale = function() {
    this.setX(miaRandom());
    this.setY(miaRandom());
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Se il costrutto **this** definisce come **pubblici** attributi e metodi della classe, il costrutto **var** definisce come **privati** attributi e metodi della classe.

```
// istanzio un oggetto
var p = new Punto();
```

```
// scrivo i valori
p.x = 10;
p.y = 13;
// questo ha
// aggiunto per
// l'istanza x
// e y.
// p.x
// p.y
```

ESERCIZIO 1.1-00-07-06

IL METODO SPECIALE TOSTRING

```
function Punto () {
  // attributi
  var x;
  var y;

  // metodi
  this.setX = function(n) { x = n; }
  this.getX = function() { return x; }
  this.setY = function(n) { y = n; }
  this.getY = function() { return y; }

  this.toString = function() {
    return '('+x+';'+y+'';
  }

  // costruttore
  this.setX(0);
  this.setY(0);
}
```

Il metodo speciale **toString** restituisce una stringa che rappresenta lo stato dell'oggetto e viene invocato automaticamente quando si tenta di stampare l'oggetto.

```
// istanzio un oggetto
var p = new Punto();
```

```
// imposto dei valori
p.setX(10);
p.setY(13);

// stampo l'oggetto
document.write(p); // (10;13)
```

ESERCIZIO 1.1-00-07-07

ESERCIZI

OVERLOADING

```
function Punto () {
  /* omissis */
  this.setPunto = function(x,y) {
    if ((typeof x == 'number') && (typeof y == 'number')) {
      this.setX(x);
      this.setY(y);
    } }
  this.setPunto = function(p) {
    if ( (typeof p == "object") && (p instanceof Punto) ) {
      this.setX(p.getX());
      this.setY(p.getY());
    } }
  /* omissis */
}
```

```
var p = new Punto();
p.setPunto(3,4);
document.write(p); // (0;0)
```

```
var p1 = new Punto();
p1.setX(3);
p1.setY(4);
var p2 = new Punto();
p2.setPunto(p1);
document.write(p2); // (3;4)
```


Con il termine **overloading** si fa riferimento alla possibilità di scrivere un metodo tante volte, ciascuna con **un elenco di parametri diverso**.

Purtroppo in Javascript se sono presenti due funzioni con lo **stesso nome** (anche se con parametri diversi) l'ultima **sovrascrive** le precedenti.

OVERLOADING

```
function Punto () {
  /* omissis */
  this.setPunto = function(x,y) {
    if ((typeof x == 'number') && (typeof y == 'number')) {
      this.setX(x);
      this.setY(y);
    } }
  this.setPunto = function(p) {
    if ( (typeof p == "object") && (p instanceof Punto) ) {
      this.setX(p.getX());
      this.setY(p.getY());
    } }
  /* omissis */
}
```

```
this.setPunto = function(a,b) {
  if ( (typeof a == "object") && (a instanceof Punto) ) {
    this.setX(a.getX());
    this.setY(a.getY());
  }
  if ((typeof a == 'number') && (typeof b == 'number')) {
    this.setX(a);
    this.setY(b);
  }
}
```

Ma poiché  in Javascript è lecito invocare una funzione anche con una quantità di parametri **inferiore** a quelli previsti, è possibile riscrivere il metodo in questo modo.

```
var p = new Punto();
p.setPunto(3,4);
document.write(p);
// (3;4)
```

```
var p1 = new Punto();
p1.setX(3);
p1.setY(4);
var p2 = new Punto();
p2.setPunto(p1);
document.write(p2);
// (3;4)
```

ESERCIZIO 1.1-00-07-04


OVERLOADING


```
function Punto (a, b) {
  // attributi
  var x;
  var y;

  /* omissis */

  // costruttore
  if ( (typeof a == 'number') && (typeof b == 'number')) {
    x = a;
    y = b;
  } else {
    x = 0;
    y = 0;
  }
}
```

```
var p1 = new Punto();           // contiene 0 e 0
var p2 = new Punto(5,5);       // contiene 5 e 5
var p3 = new Punto(5,'n');     // contiene 0 e 0
```

Ma poiché  in Javascript è lecito invocare una funzione anche con una quantità di parametri **inferiore** a quelli previsti, è possibile riscrivere il metodo in questo modo.

Un discorso  analogo si può fare nel caso del **costruttore**.

ESERCIZIO 1.1-00-07-04

OVERLOADING

```
function miaClasse () {
  // metodo senza argomenti
  this.mioMetodo = function() {
    document.write("no parametri");
  }
}
```

Un metodo alternativo è quello di scrivere il metodo tante volte quante sono le versioni che vogliamo realizzare e fare invocare all'interno di un metodo più complesso un metodo più semplice.

```
var p;
p = new miaClasse();
p.mioMetodo();
```

ESERCIZIO 1.1-00-07-10

OVERLOADING

```
function miaClasse () {
  // metodo senza argomenti
  this.mioMetodo = function() {
    document.write("no parametri");
  }
  // conservo un riferimento al metodo
  var _mioMetodo = this.mioMetodo;
}
```

Un metodo alternativo è quello di scrivere il metodo tante volte quante sono le versioni che vogliamo realizzare e fare invocare all'interno di un metodo più complesso un metodo più semplice.

```
var p;
p = new miaClasse();
p.mioMetodo();
```

ESERCIZIO 1.1-00-07-10

OVERLOADING

```
function miaClasse () {
  // metodo senza argomenti
  this.mioMetodo = function() {
    document.write("no parametri");
  }
  // conservo un riferimento al metodo
  var _mioMetodo = this.mioMetodo;
  // ridefinisco il metodo
  this.mioMetodo = function(a) {
    // se il parametro non c'è
    if (typeof a == 'undefined')
      // chiamo versione precedente
      _mioMetodo.call(this);
    else
      document.write("un parametro");
  }
}
```

Un metodo alternativo è quello di scrivere il metodo tante volte quante sono le versioni che vogliamo realizzare e fare invocare all'interno di un metodo più complesso un metodo più semplice.

```
var p;
p = new miaClasse();
p.mioMetodo(5);
p.mioMetodo();
```

ESERCIZIO 1.1-00-07-10

OVERLOADING

```
function miaClasse () {
  // metodo senza argomenti
  this.mioMetodo = function() {
    document.write("no parametri");
  }
  // conservo un riferimento al metodo
  var _mioMetodo = this.mioMetodo;
  // ridefinisco il metodo
  this.mioMetodo = function(a) {
    // se il parametro non c'è
    if (typeof a == 'undefined')
      // chiamo versione precedente
      _mioMetodo.call(this);
    else
      document.write("un parametro");
  }
  // conservo un riferimento al metodo
  var __mioMetodo = this.mioMetodo;
  // ridefinisco il metodo
  this.mioMetodo = function(a, b) {
    // se il secondo parametro non c'è
    if (typeof b == 'undefined')
      // chiamo versione precedente
      __mioMetodo.call(this, a);
    else
      document.write("dueparametri");
  }
}
```

Un metodo alternativo è quello di scrivere il metodo tante volte quante sono le versioni che vogliamo realizzare e fare invocare all'interno di un metodo più complesso un metodo più semplice.

```
var p;
p = new miaClasse();
p.mioMetodo(5,6);
p.mioMetodo(5);
p.mioMetodo();
```

ESERCIZIO 1.1-00-07-10

ESERCIZI

EREDITARIETÀ

L'**ereditarietà** è quel concetto cardine della OOP che ci permette di **estendere** una classe, acquisendone tutti gli attributi e tutti i metodi.



Punto
ascissa
ordinata
costruttore ()
costruttore (x, y)
costruttore (p)
setAscissa (x)
setOrdinata (y)
getAscissa ()
getOrdinata ()
setPunto (x, y)
setPunto (p)
toString ()



Punto3d
ascissa
ordinata
quota
costruttore ()
costruttore (x, y, z)
costruttore (p)
setAscissa (x)
setOrdinata (y)
setQuota (z)
getAscissa ()
getOrdinata ()
getQuota ()
setPunto (x, y, z)
setPunto (p)
toString ()

QUALCOSA DA EREDITARE

L'erediterietà è quel concetto cardine della OOP che ci permette di **estendere** una classe, acquisendone tutti gli attributi e tutti i metodi.

Punto
ascissa
ordinata
costruttore ()
costruttore (x, y)
costruttore (p)
setAscissa (x)
setOrdinata (y)
getAscissa ()
getOrdinata ()
setPunto (x, y)
setPunto (p)
toString ()



Punto3d
ascissa
ordinata
quota
costruttore ()
costruttore (x, y, z)
costruttore (p)
setAscissa (x)
setOrdinata (y)
setQuota (z)
getAscissa ()
getOrdinata ()
getQuota ()
setPunto (x, y, z)
setPunto (p)
toString ()



QUALCOSA DA IMPLEMENTARE

L'erediterietà è quel concetto cardine della OOP che ci permette di **estendere** una classe, acquisendone tutti gli attributi e tutti i metodi.

Punto
ascissa
ordinata
costruttore ()
costruttore (x, y)
costruttore (p)
setAscissa (x)
setOrdinata (y)
getAscissa ()
getOrdinata ()
setPunto (x, y)
setPunto (p)
toString ()



Punto3d
ascissa
ordinata
quota
costruttore ()
costruttore (x, y, z)
costruttore (p)
setAscissa (x)
setOrdinata (y)
setQuota (z)
getAscissa ()
getOrdinata ()
getQuota ()
setPunto (x, y, z)
setPunto (p)
toString ()



QUALCOSA DA MODIFICARE

L'**ereditarietà** è quel concetto cardine della OOP che ci permette di **estendere** una classe, acquisendone tutti gli attributi e tutti i metodi.

Punto
ascissa
ordinata
costruttore ()
costruttore (x, y)
costruttore (p)
setAscissa (x)
setOrdinata (y)
getAscissa ()
getOrdinata ()
setPunto (x, y)
setPunto (p)
toString ()



Punto3d
ascissa
ordinata
quota
costruttore ()
costruttore (x, y, z)
costruttore (p)
setAscissa (x)
setOrdinata (y)
setQuota (z)
getAscissa ()
getOrdinata ()
getQuota ()
setPunto (x, y, z)
setPunto (p)
toString ()



EREDITARIETÀ IN JAVASCRIPT

L'**ereditarietà** è quel concetto cardine della OOP che ci permette di **estendere** una classe, acquisendone tutti gli attributi e tutti i metodi.

L'ereditarietà in Javascript è implementata in almeno **tre modi diversi**.. ma **mai** come avviene negli altri linguaggi OOP!
 Come già dichiarato noi qui ci limiteremo a mostrare come funzionano le cose (uno dei tanti modi) **senza approfondire** troppo la questione.

Punto
ascissa
ordinata
costruttore ()
costruttore (x, y)
costruttore (p)
setAscissa (x)
setOrdinata (y)
getAscissa ()
getOrdinata ()
setPunto (x, y)
setPunto (p)
toString ()



Punto3d
ascissa
ordinata
quota
costruttore ()
costruttore (x, y, z)
costruttore (p)
setAscissa (x)
setOrdinata (y)
setQuota (z)
getAscissa ()
getOrdinata ()
getQuota ()
setPunto (x, y, z)
setPunto (p)
toString ()

COME ACCEDERE ALL'EREDITÀ

```
function Padre() {
  this.metodoPadre = function() {
    return "dalpadre";
  }
}
```

```
function Figlio() {
  // eredito da Padre
  Padre.call(this);
  // metodi propri
  this.metodoFiglio = function () {
    return "dalfiglio";
  }
}
```

```
var x = new Figlio();
document.write(x.metodoPadre()+"<br/>"); // dalpadre
document.write(x.metodoFiglio()+"<br/>"); // dalfiglio
```

! Data la classe padre, sarà possibile avere tutte le sue funzionalità in un'altra classe figlio mediante l'utilizzo del metodo **call**.

! Le istanze della classe figlio possono usare tutti i metodi (pubblici) sia del padre che del figlio.

ESERCIZIO 1.1-00-09-04

SI EREDITA SOLO IL PUBBLICO

```
function Padre() {
  // attributi
  var valore;
  //metodi
  this.setValore = function(x) { valore = x; }
  this.getValore = function() { return valore; }
  //costruttore
  valore=0;
}
```

```
function Figlio() {
  // eredità
  Padre.call(this);
  this.passaDaCasa = function() {
    valore = Math.floor(Math.random()*10);
  }
}
```

```
var x = new Figlio();
x.setValore(20);
document.write(x.getValore()+"<br/>"); //20
x.passaDaCasa();
document.write(x.getValore()+"<br/>"); //20
```

! Per rispettare il concetto di **incapsulamento**, la classe figlio può accedere solo a ciò che è pubblico.

! Pertanto con questa istruzione si crea (implicitamente) un attributo **valore** nella classe Figlio

ESERCIZIO 1.1-00-09-05

SI EREDITA SOLO IL PUBBLICO

```
function Padre() {
  // attributi
  var valore;
  //metodi
  this.setValore = function(x) { valore = x; };
  this.getValore = function() { return valore; };
  //costruttore
  valore=0;
}
```

```
function Figlio() {
  // eredità
  Padre.call(this);
  this.passaDaCasa = function() {
    valore = Math.floor(Math.random()*10);
    this.setValore(Math.floor(Math.random()*10));
  }
}
```

```
var x = new Figlio();
x.setValore(20);
document.write(x.getValore()+"<br/>"); //20
x.passaDaCasa();
document.write(x.getValore()+"<br/>"); //8
```

Per rispettare il concetto di **incapsulamento**, la classe figlio può accedere solo a ciò che è pubblico.

Pertanto con questa istruzione si crea (implicitamente) un attributo **valore** nella classe Figlio

Per accedere agli attributi della classe padre bisognerà usare i metodi (pubblici) da esso messi a disposizione.

OVERRIDING

Il **polimorfismo** sostanzialmente è la possibilità di invocare lo stesso metodo su istanze di classi diverse.

Il polimorfismo è implementato mediante una tecnica che si chiama **overriding**.

Fare **overriding** di un metodo significa **riscrivere** in una classe figlio un metodo della classe padre.

Il metodo della classe padre **non** è più accessibile.

```
function Padre() {
  this.getNomeClasse = function() {
    return "Padre";
  }
}
```

```
function Figlio() {
  Padre.call(this);
  this.getNomeClasse = function() {
    return "Figlio";
  }
}
```

```
var x = new Padre();
var y = new Figlio();
document.write(x.getNomeClasse()+"<br/>");
//Padre
document.write(y.getNomeClasse()+"<br/>");
//Figlio
```

OVERLOADING ED OVERRIDING

Il **polimorfismo** sostanzialmente è la possibilità di invocare lo stesso metodo su istanze di classi diverse.

Il polimorfismo è implementato mediante una tecnica che si chiama **overriding**.

Fare **overriding** di un metodo significa **riscrivere** in una classe figlio un metodo della classe padre.

Il metodo della classe padre **non** è più accessibile.

Abbiamo già visto che anche nel caso dell'**overloading** (riscrittura di un metodo con un diverso elenco di parametri) si perdeva l'accesso ai metodi precedentemente sviluppati. Ed abbiamo già visto come **ovviare** a questo problema.

Rivediamo la cosa in **ereditarietà**.

OVERLOADING ED OVERRIDING

```
function Padre() {
  this.stampa = function(x) {
    var msg;
    msg = "style='color:red; font-size:1em;";
    msg = "<p " + msg + ">" + x + "</p>";
    document.write(msg);
  }
}
```

```
var x = new Padre();
x.stampa("ciao");
```

Abbiamo già visto che anche nel caso dell'**overloading** (riscrittura di un metodo con un diverso elenco di parametri) si perdeva l'accesso ai metodi precedentemente sviluppati. Ed abbiamo già visto come **ovviare** a questo problema.

Rivediamo la cosa in **ereditarietà**.

ciao

OVERLOADING ED OVERRIDING

```
function Padre() {
  this.stampa = function(x) {
    var msg;
    msg = "style='color:red; font-size:1em;";
    msg = "<p " + msg + ">" + x + "</p>";
    document.write(msg);
  }
}
```

```
function Figlio() {
  Padre.call(this);
  this.stampa = function(x, y) {
    var msg = "<p style='font-size:2em;' >";
    msg += "<span style='color:blue;'>";
    msg += x + "</span>";
    msg += "<span style='color:green;'>";
    msg += y + "</span>";
    msg += "</p>";
    document.write(msg);
  }
}
```

```
var y = new Figlio();
y.stampa("au", "revoir");
y.stampa("byebye");
```

! Abbiamo già visto che anche nel caso dell'**overloading** (riscrittura di un metodo con un diverso elenco di parametri) si perdeva l'accesso ai metodi precedentemente sviluppati. Ed abbiamo già visto come **ovviare** a questo problema.

! Rivediamo la cosa in **ereditarietà**.

aurevoir

byebyeundefined

ESERCIZIO 1.1-00-09-07

OVERLOADING ED OVERRIDING

```
function Padre() {
  /* omissis */ function(x) {
    var msg;
    msg = "style='color:red; font-size:1em;";
    msg = "<p " + msg + ">" + x + "</p>";
    document.write(msg);
  }
}
```

```
function Figlio() {
  Padre.call(this);
  this.stampa = function(x, y) {
    var msg = "<p style='font-size:2em;' >";
    msg += "<span style='color:blue;'>";
    msg += x + "</span>";
    msg += "<span style='color:green;'>";
    msg += y + "</span>";
    msg += "</p>";
    document.write(msg);
  }
}
```

```
var y = new Figlio();
y.stampa("au", "revoir");
y.stampa("byebye");
```

! Abbiamo già visto che anche nel caso dell'**overloading** (riscrittura di un metodo con un diverso elenco di parametri) si perdeva l'accesso ai metodi precedentemente sviluppati. Ed abbiamo già visto come **ovviare** a questo problema.

! Rivediamo la cosa in **ereditarietà**.

aurevoir

byebyeundefined

ESERCIZIO 1.1-00-09-07

OVERLOADING ED OVERRIDING

```
function Padre() {
    /* omissis */
}

function Figlio() {
    Padre.call(this);
    _stampa = this.stampa;
    this.stampa = function(x, y) {
        if (typeof y == 'undefined')
            _stampa.call(this, x);
        else {
            var msg = "<p style='font-size:2em;' >";
            msg += "<span style='color:blue;'>";
            msg += x + "</span>";
            msg += "<span style='color:green;'>";
            msg += y + "</span>";
            msg += "</p>";
            document.write(msg);
        }
    }
}

var y = new Figlio();
y.stampa("au", "revoir");
y.stampa("byebye");
```

! Abbiamo già visto che anche nel caso dell'**overloading** (riscrittura di un metodo con un diverso elenco di parametri) si perdeva l'accesso ai metodi precedentemente sviluppati. Ed abbiamo già visto come **ovviare** a questo problema.

! Rivediamo la cosa in **ereditarietà**.

```
aurevoir
byebye
```

ESERCIZIO 11-00-09-10

INVOCAZIONE IMPLICITA DEL COSTRUTTORE

```
function Padre (x) {
    // attributi
    var valore; // stringa

    // metodi
    this.setValore = function(x) {
        if (typeof x == 'string')
            valore = x;
    }
    this.getValore = function() {
        return valore;
    }

    // costruttore
    if (typeof x != 'string') {
        this.setValore('');
    } else {
        this.setValore(x);
    }
}

var a = new Figlio(); // '' 0
var a = new Figlio('ok', 8); // 'ok' 8
var a = new Figlio(15, 8); // '' 8
var a = new Figlio('ok', 'bye'); // 'ok' 0
```

```
function Figlio (x, y) {
    Padre.call(this);

    // attributi
    var altroValore; // numerico

    // metodi
    this.setAltroValore = function(x) {
        if (typeof x == 'number')
            altroValore = x;
    }
    this.getAltroValore = function() {
        return altroValore;
    }

    // costruttore
    if (typeof x != 'string') {
        this.setValore('');
    } else {
        this.setValore(x);
    }
    if (typeof y != 'number') {
        altroValore = 0;
    } else {
        altroValore = y;
    }
}
```

ESERCIZIO 11-00-09-12

INVOCAZIONE IMPLICITA DEL COSTRUTTORE

```
function Padre (x) {
  // attributi
  var valore; // stringa

  /* omissis */

  // costruttore
  if (typeof x !== 'string') {
    this.setValore('');
  } else {
    this.setValore(x);
  }
}
```

```
function Figlio (x, y) {
  Padre.call(this);

  // attributi
  var altroValore; // numerico

  /* omissis */

  // costruttore
  if (typeof x !== 'string') {
    this.setValore('');
  } else {
    this.setValore(x);
  }
  if (typeof y !== 'number') {
    altroValore = 0;
  } else {
    altroValore = y;
  }
}
```

```
var a = new Figlio(); // '' 0
var a = new Figlio('ok',8); // 'ok' 8
var a = new Figlio(15,8); // '' 8
var a = new Figlio('ok','bye'); // 'ok' 0
```

! Osserviamo che le classi Padre e Figlio **condividono** lo stesso codice!

ESERCIZIO 11-00-09-12

INVOCAZIONE IMPLICITA DEL COSTRUTTORE

```
function Padre (x) {
  // attributi
  var valore; // stringa

  /* omissis */

  // costruttore
  if (typeof x !== 'string') {
    this.setValore('');
  } else {
    this.setValore(x);
  }
}
```

```
function Figlio (x, y) {
  Padre.call(this);
  Padre.call(this, x);

  // attributi
  var altroValore; // numerico

  /* omissis */

  // costruttore
  if (typeof x !== 'string') {
    this.setValore('');
  } else {
    this.setValore(x);
  }
  if (typeof y !== 'number') {
    altroValore = 0;
  } else {
    altroValore = y;
  }
}
```

```
var a = new Figlio(); // '' 0
var a = new Figlio('ok',8); // 'ok' 8
var a = new Figlio(15,8); // '' 8
var a = new Figlio('ok','bye'); // 'ok' 0
```

! Osserviamo che le classi Padre e Figlio **condividono** lo stesso codice!

! Estendendo una classe ne **invoca il costruttore** e quindi gli passo i parametri.

ESERCIZIO 11-00-09-12

UN ESEMPIO COMPLETO

Punto
ascissa
ordinata
costruttore ()
costruttore (x,y)
costruttore (p)
setAscissa (x)
setOrdinata (y)
getAscissa ()
getOrdinata ()
setPunto (x,y)
setPunto (p)
toString ()



Punto3d
ascissa
ordinata
quota
costruttore ()
costruttore (x,y,z)
costruttore (p)
setAscissa (x)
setOrdinata (y)
setQuota (z)
getAscissa ()
getOrdinata ()
getQuota ()
setPunto (x,y,z)
setPunto (p)
toString ()

IL PUNTO DI PARTENZA

```
function Punto3d () {
    // estendo la classe Punto
    Punto.call(this);
}
```

```
// istanzio un oggetto
var p = new Punto3d();

// uso i metodi del padre
p.setAscissa(10);
p.setOrdinata(13);
document.write(p); // (10,13)
```

INVOCAZIONE IMPLICITA DEL COSTRUTTORE DELLA SUPERCLASSE

```
function Punto3d (x, y, z) { // per il momento non usiamo il parametro z
  // estendo la classe Punto
  Punto.call(this, x, y);
}
```

```
// istanzio un oggetto
var p = new Punto3d();

// uso i metodi del padre
p.setAscissa(10);
p.setOrdinata(13);
document.write(p); // (10,13)
```

ATTRIBUTI E METODI NUOVI

```
function Punto3d (x, y, z) {
  // estendo la classe Punto
  Punto.call(this, x, y);

  // attributi nuovi
  var quota;

  // metodi nuovi
  this.setQuota = function(z) { quota = z; };
  this.getQuota = function() { return quota; };

  // codice del costruttore
  quota = (typeof(z)=='number') ? z : 0;
}
```

```
// istanzio un oggetto
var p = new Punto3d(1, 2, 3);

// uso i metodi nuovi
p.setQuota(51);
document.write(p.getQuota()); // 51
```

METODI IN OVERRIDE

```
function Punto3d (x, y, z) {
  /* omissis */
  // metodi in override
  this.toString = function() {
    return (" "+this.getAscissa()+" "+this.getOrdinata()+" "+quota+"");
  }
  // codice del costruttore
  quota = (typeof(z)=='number') ? z : 0;
}
```

```
// istanzio un oggetto
var p = new Punto3d();

// uso i metodi in override
p.setAscissa(10);
p.setOrdinata(13);
p.setQuota(51);
document.write(p); // (10,13,51)
```

METODI IN OVERLOAD

```
function Punto3d (x, y, z) {
  /* omissis */
  // metodi in overload
  this.setPunto = function(x, y, z) {
    if ( (typeof(x) == 'object') && (x instanceof Punto3d) ){
      this.setAscissa ( x.getAscissa() );
      this.setOrdinata( x.getOrdinata() );
      quota = x.getQuota();
    }else if((typeof(x)=='number')&&(typeof(y)=='number')&&(typeof(z)=='number')){
      this.setAscissa ( x );
      this.setOrdinata( y );
      quota = z;
    } else {
      this.setAscissa ( 0 );
      this.setOrdinata( 0 );
      quota = 0;
    }
  }
  // codice del costruttore
  this.setPunto(x, y, z);
}
```

```
// istanzio un oggetto
var p = new Punto3d();
// uso i metodi in overload
p.setPunto(1,2,3);
document.write(p); // (1,2,3)
```

INVOCAZIONE DI METODI DEL PADRE DOPO UN OVERLOAD O UN OVERRIDE

```
function Punto3d (x, y, z) {  
  /* omissis */  
  // metodi in overload  
  var _setPunto = this.setPunto;  
  this.setPunto = function(x, y, z) {  
    if ( (typeof(x) == 'object') && (x instanceof Punto3d) ){  
      x = x.getAscissa();  
      y = x.getOrdinata();  
      quota = x.getQuota();  
    }else {  
      _setPunto.call(this,x,y);  
      if(typeof(z)=='number'){  
        quota = z;  
      } else {  
        quota = 0;  
      }  
    }  
  }  
  // codice del costruttore  
  this.setPunto(x, y, z);  
}
```

```
// istanzio un oggetto con il costruttore che invocherà un metodo in overload  
var p = new Punto3d(1,2,3);  
document.write(p); // (1,2,3)
```

ESERCIZI

POLIMORFISMO (AVANZATO)

```
function Padre() {
}

function Figlio() {
  // eredito
  Padre.call(this);
}
```

```
var x = new Figlio();
document.write( x instanceof Padre); // false
```

Uno dei principi del **polimorfismo** è che una istanza della classe figlio è sì un'istanza della classe figlio ma è **contemporaneamente** anche un'istanza della classe padre.

Il modo che abbiamo indicato per implementare la OOP in Javascript **non** offre questa caratteristica.

ESERCIZIO 1.1-00-09-09

POLIMORFISMO (AVANZATO)

```
function Padre() {
}

function Figlio() {
  // eredito
  Padre.call(this);
}
```

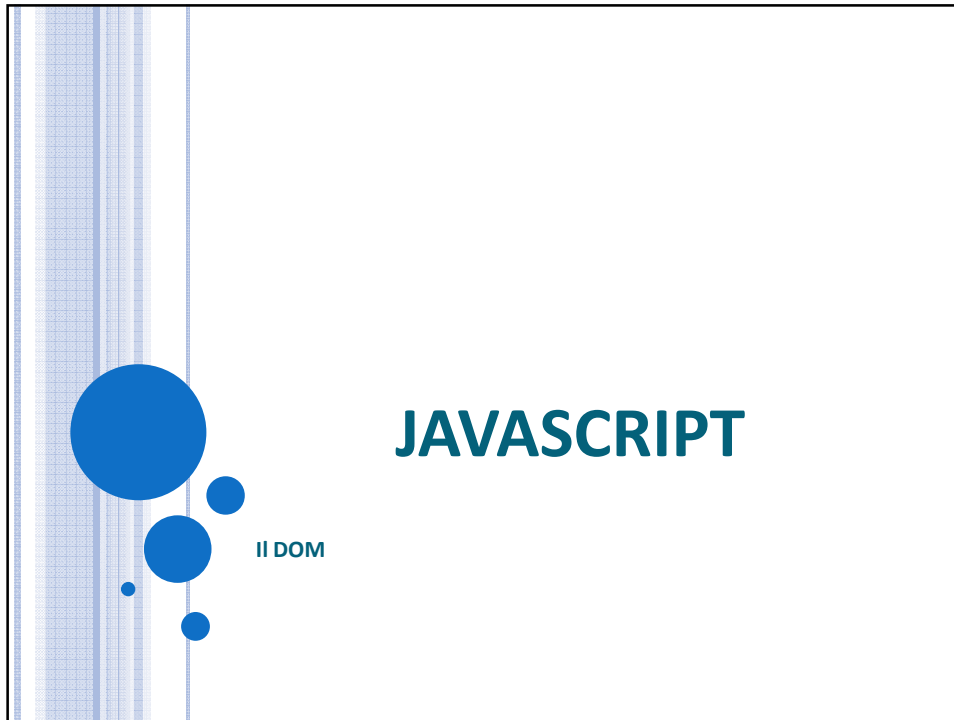
```
var x = new Figlio();
document.write( x instanceof Padre); // false
```

```
Figlio.prototype = new Padre();

var x = new Figlio();
document.write( x instanceof Padre); // true
```

Per ottenere questa caratteristica dobbiamo rifarci ad un'altra strategia OOP, quella dei **prototipi**.

ESERCIZIO 1.1-00-09-09





JAVASCRIPT → PARTE 2 → DOM CC BY


A COSA SERVE IL DOM?

DOM è l'acronimo di **Document Object Model**, ovvero "modello a oggetti del documento".

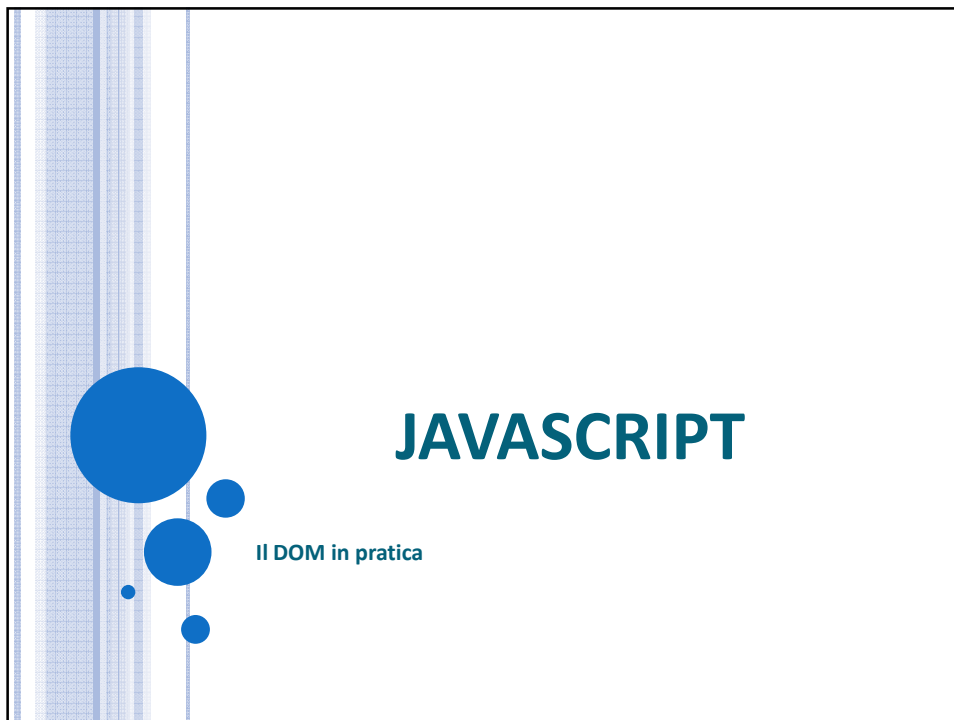
DOM

Il concetto iniziale non è proprio banale e con il passare del tempo si complica ulteriormente. 

D'altra parte però si presta ad un utilizzo **ridotto** e per certi versi addirittura **inconsapevole**. 

Cominciamo quindi con un approccio del tutto **pratico** per poi proporre una visione **teorica**. 

VERSIONE 2.8 - DIAPOSITIVA 96 ALESSANDRO URSOMANDO



JAVASCRIPT → PARTE 2 → DOM CC BY

A COSA SERVE IL DOM?

Con il DOM possiamo **interagire** con l'utente!

Per esempio possiamo mostrare popup ed aprire altre pagine WEB..

The screenshot shows a standard Windows-style dialog box titled "Avviso JavaScript". The text inside reads: "Abbiamo cambiato indirizzo! Ora siamo su www.nuovoindirizzo.bho. Aggiorna i tuoi 'Preferiti'. Per stavolta ti ci portiamo noi." There is an "OK" button at the bottom right.

VERSIONE 2.8 - DIAPOSITIVA 98 ALESSANDRO URSOMANDO

A COSA SERVE IL DOM?

Con il DOM possiamo **interagire** con l'utente!



Per esempio possiamo mostrare popup ed aprire altre pagine WEB..



A COSA SERVE IL DOM?

Con il DOM possiamo **interagire** con l'utente!



Oppure possiamo scoprire se la pagina visualizzata è online o offline e mostrarlo a video..



A COSA SERVE IL DOM?

Con il DOM possiamo **interagire** con l'utente!



Oppure possiamo modificare dinamicamente il titolo della pagina..



PUNTA LE IMMAGINI PER CAMBIARE IL TITOLO DELLA PAGINA



A COSA SERVE IL DOM?

Con il DOM possiamo **interagire** con l'utente!



Oppure possiamo modificare dinamicamente l'immagine da mostrare..



CLICCA SU UNO DEI RADIOBUTTON PER FARE COMPARIRE L'IMMAGINE AD ESSO RELATIVA.

Emergency Medici senza frontiere



A COSA SERVE IL DOM?

Con il DOM possiamo **interagire** con l'utente!



Oppure possiamo modificare dinamicamente il colore di un elemento.



CLICCA SULLO SPAZIO COLORATO INTORNO ALL'IMMAGINE PER STABILIRE IL COLORE.



IL DOM USA GLI OGGETTI

Il **documento** viene visto come un **oggetto** con i suoi **attributi** e i suoi **metodi**.



E la **finestra** che lo ospita pure. Cominciamo da quest'ultima.



CLICCA SULLO SPAZIO COLORATO INTORNO ALL'IMMAGINE PER STABILIRE IL COLORE.



LA CLASSE WINDOW

Ovviamente esiste una sola istanza di questa classe: l'oggetto **window**.



Questo è l'oggetto *padre* di tutti gli altri.



Per questo motivo è possibile utilizzare i suoi metodi e le sue proprietà anche omettendo il suo nome.



Come al solito non vogliamo essere esaustivi ma solo dare **un'idea** di ciò che c'è. La classe Window (e le successive classi che vedremo) potrebbero avere **molti più** attributi e metodi di quelli mostrati.



Window	
Document	document
void	alert(String)
Boolean	confirm(String)
String	prompt(String, String)
void	open(String, String)

LA CLASSE WINDOW

```
<body>
<script src="eser.js"></script>
</body>
```

```
var msg = "Abbiamo cambiato indirizzo!\n";
msg += "Ora siamo su www.nuovoindirizzo.bho\n";
msg += "Aggiorna i tuoi 'Preferiti'.\n\n";
msg += "Per stavolta ti ci portiamo noi.\n";
alert (msg);
open("inizio.html", "_self");
```

```
<body>
<h1>Benvenuto!</h1>

</body>
```

Vediamo un esempio in cui, al momento di mostrare la pagina richiesta, viene proposto un popup e successivamente si viene ricondotti altrove.



LA CLASSE WINDOW

Analizziamo ora l'attributo **document** (di tipo **Document**) che rappresenta il documento.

Anche stavolta (ovviamente) esiste una sola istanza di questa classe.

Vediamola.



Window	
Document	document
void	alert(String)
Boolean	confirm(String)
String	prompt(String, String)
void	open(String, String)

LA CLASSE DOCUMENT

Analizziamo ora l'attributo **document** (di tipo **Document**) che rappresenta il documento.

Anche stavolta (ovviamente) esiste una sola istanza di questa classe.

Vediamola.



Document	
String	URL
String	title
Element	body
Collection[]	cookie
Collection[]	forms
Collection[]	images
Collection[]	links
void	write(String)
Element	getElementById(String)
Element[]	getElementsByTagName(String)

Vediamo adesso un esercizio in cui in base al valore dell'attributo URL stabiliamo se la pagina è **locale** o **remota** ed aggiungiamo un elemento di tipo **footer** per mostrare l'informazione.



LA CLASSE DOCUMENT

```
<body>
<script src="eser.js"></script>
<h1>Benvenuto!</h1>

</body>
```

```
var indirizzo = document.URL;
var protocollo = indirizzo.substr(0,4);
if (protocollo=="file")
document.write("<footer> Pagina locale </footer>");
else // protocollo vale "http"
document.write("<footer> Pagina remota </footer>");
```

BENVENUTO!



PAGINA LOCALE

ESERCIZIO 1.1-00-06-02

GLI EVENTI

Una delle possibilità più interessanti risiede nel fatto che possiamo eseguire del codice Javascript **al verificarsi di un evento**.

- onClick
- onDbClick
- onMouseOver
- onMouseOut
- onChange
- onFocus
- onBlur
- onSubmit
- onReset

EVENTI

Per esempio possiamo fare in modo che cliccando su elementi diversi si modifichi il titolo della pagina.

GLI EVENTI

```


```

PUNTA LE IMMAGINI PER CAMBIARE IL TITOLO DELLA PAGINA



GLI EVENTI

```


```

```


```

```
function SetTitolo1() {
  document.title = "Emergency";
}

function SetTitolo2() {
  document.title = "Medici Senza Frontiere";
}
```


ESERCIZI

LA CLASSE DOCUMENT

Uno dei metodi più interessanti di **Document** è **getElementById**.

Grazie ad esso, conoscendo l'**id** di un certo elemento, possiamo conoscerne e modificarne le **caratteristiche** mediante un oggetto di tipo **Element** che lo rappresenta.



Document	
String	URL
String	title
Element	body
Collection[]	cookie
Collection[]	forms
Collection[]	images
Collection[]	links
void	write(String)
Element	getElementById(String)
Element[]	getElementsByTagName(String)

LA CLASSE ELEMENT

Uno dei metodi più interessanti di **Document** è **getElementById**.
 Grazie ad esso, conoscendo l'**id** di un certo elemento, possiamo conoscerne e modificarne le **caratteristiche** mediante un oggetto di tipo **Element** che lo rappresenta.

Nel prossimo esempio vedremo come ottenere l'oggetto che rappresenta un elemento di tipo **img** e come modificare l'attributo **src**.

Element	
String	tagName
String	className
String	id
Style	style
Element[]	getElementsByTagName(String)
String	getAttribute(String)
void	setAttribute(String, String)
Boolean	hasAttribute(String)
void	removeAttribute(String)

LA CLASSE ELEMENT

```
<form>
  <label>
    <input type="radio" name="scelta" onClick="SetImg1();" checked="checked"/>
      Emergency
    </label>
  <label>
    <input type="radio" name="scelta" onClick="SetImg2();" />
      Medici senza frontiere
    </label>
</form>

```

CLICCA SU UNO DEI RADIOBUTTON PER FARE COMPARIRE L'IMMAGINE AD ESSO RELATIVA.



ESERCIZIO 11-00-06-05

LA CLASSE ELEMENT

```
<form>
  <label>
    <input type="radio" name="scelta" onClick="SetImg1();" checked="checked" />
    Emergency
  </label>
  <label>
    <input type="radio" name="scelta" onClick="SetImg2();" />
    Medici senza frontiere
  </label>
</form>

```

```
function SetImg1() {
  var x = document.getElementById("immagineDinamica");
  x.setAttribute("src", "e.jpg");
}

function SetImg2() {
  var x = document.getElementById("immagineDinamica");
  x.setAttribute("src", "mf.jpg");
}
```

LA CLASSE ELEMENT

Uno dei metodi più interessanti di **Document** è **getElementById**.

Grazie ad esso, conoscendo l'**id** di un certo elemento, possiamo conoscerne e modificarne le **caratteristiche** mediante un oggetto di tipo **Element** che lo rappresenta.

Nel prossimo esempio vedremo come ottenere l'oggetto che rappresenta un elemento di tipo **img** e come modificare l'attributo **src**.

In realtà, oltre alla classe generica **Element**, esistono **tante altre classi** che estendono **Element** con gli attributi propri di vari elementi HTML.

Element	
String	tagName
String	className
String	id
Style	style
Element[]	getElementsByTagName(String)
String	getAttribute(String)
void	setAttribute(String, String)
Boolean	hasAttribute(String)
void	removeAttribute(String)

LA CLASSE ELEMENT

```
function SetImg1() {
  var x = document.getElementById("immagineDinamica");
  x.setAttribute("src", "e.jpg");
}

function SetImg2() {
  var x = document.getElementById("immagineDinamica");
  x.setAttribute("src", "msf.jpg");
}
```

Per esempio se l'elemento è relativo al tag **img** disponiamo – tra l'altro – dell'attributo **src**.

```
function SetImg1() {
  var x = document.getElementById("immagineDinamica");
  x.src = "e.jpg";
}

function SetImg2() {
  var x = document.getElementById("immagineDinamica");
  x.src = "msf.jpg";
}
```

LA CLASSE ELEMENT

Uno degli attributi più interessanti della classe **Element** è senz'altro **style**.

Element	
String	tagName
String	className
String	id
Style	style
Element[]	getElementByTagName(String)
String	getAttribute(String)
void	setAttribute(String, String)
Boolean	hasAttribute(String)
void	removeAttribute(String)

LA CLASSE STYLE

Uno degli attributi più interessanti della classe **Element** è senz'altro **style**.



La classe **Style** non ha metodi, ma dispone di una **grossa quantità** di **attributi** che ci permettono di andare a leggere e modificare le **dichiarazioni CSS** in linea.



Esiste tutta una parte del DOM che permette di andare ad interagire con i **fogli di stile esterni**, ma che non sarà oggetto di questo corso.



Style	
...	
String	backgroundAttachment
String	backgroundColor
String	backgroundImage
String	backgroundRepeat
...	

LA CLASSE STYLE

CLICCA SUL PULSANTE PER MODIFICARE IL COLORE INTORNO ALL'IMMAGINE.



```
<form>
  <input id="colore" type="color" onChange="CambiaColore()" />
</form>

```

```
function CambiaColore() {
  var x = document.getElementById("colore");
  var y = x.value;
  x = document.getElementById("bimba");
  x.style.backgroundColor = y;
}
```

ESERCIZIO 1.1-00-06-07

METODI CHE SIMULANO GLI EVENTI

Tutte le classi che estendono **Element** per gestire **elementi di input** aggiungono – tra l'altro – i metodi per simulare la conquista e la perdita del **focus** e il **click** da parte dell'utente.

La classe che estende **Element** per gestire l'elemento **form** aggiunge – tra l'altro – i metodi per simulare l'**invio** del modulo e il suo **reset**.

Element	
String	tagName
String	className
String	id
Style	style
Element[]	getElementByTagName(String)
String	getAttribute(String)
void	setAttribute(String, String)
Boolean	hasAttribute(String)
void	removeAttribute(String)
void	focus()
void	blur()
void	click()
void	submit()
void	reset()

METODI CHE SIMULANO GLI EVENTI

CLICCA SULLO SPAZIO COLORATO INTORNO ALL'IMMAGINE PER STABILIRE IL COLORE.



```
<form>
  <input id="colore" type="color" onChange="CambiaColore()" />
</form>

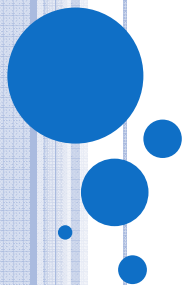
```

```
function ScegliColore() {
  var x = document.getElementById("colore");
  x.click();
}
function CambiaColore() {
  var x = document.getElementById("colore");
  var y = x.value;
  x = document.getElementById("bimba");
  x.style.backgroundColor = y;
}
```

ESERCIZI

JAVASCRIPT

Il DOM in teoria



IL DOM IN TEORIA

Il **DOM** è un **API**
per la **gestione dinamica**
dei **contenuti** di una pagina WEB.

API (Application Program Interface) è un'interfaccia per la programmazione di applicazioni: un **insieme di strumenti** che i programmi possono usare per usufruire di **funzionalità** offerte da un **sistema sottostante**.

API

IL DOM IN TEORIA

Il **DOM** è un **API**
per la **gestione dinamica**
dei **contenuti** di una pagina WEB.

Col tempo i contenuti
di una pagina WEB si sono **moltiplicati**
e sono divenuti via via più **complessi**.

Con il termine DOM
non si intende più
la sola **gestione del documento**
ma anche del **browser**, dei **fogli di stile**, ecc.

<http://docs.oracle.com/javase/1.5.0/docs/guide/plugin/dom>

IL DOM IN TEORIA

Il **DOM** è un **API** per la **gestione dinamica** dei **contenuti** di una pagina WEB.

Col tempo i contenuti di una pagina WEB si sono **moltiplicati** e sono divenuti via via più **complessi**.

Con il termine DOM non si intende più la sola **gestione del documento** ma anche del **browser**, dei **fogli di stile**, ecc.

Il DOM è un API offerta dal **browser**.

Il **W3C** diffonde delle specifiche che sono **indipendenti** dalla piattaforma e dal browser.

Come al solito non è detto che gli **sviluppatori** di browser le **rispettino**.

Il DOM è a disposizione di tutti i **linguaggi lato client**.

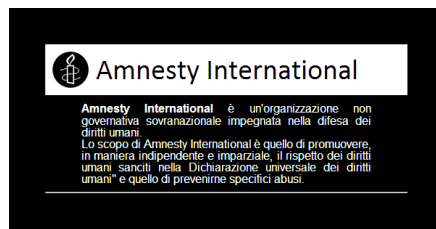
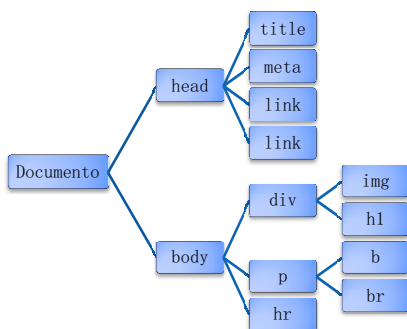
JAVASCRIPT

Il DOM
[corso avanzato]

INTRODUZIONE AL DOM

Il Document Object Model è una forma di rappresentazione dei documenti HTML in cui ogni elemento della pagina è un oggetto.

DOM



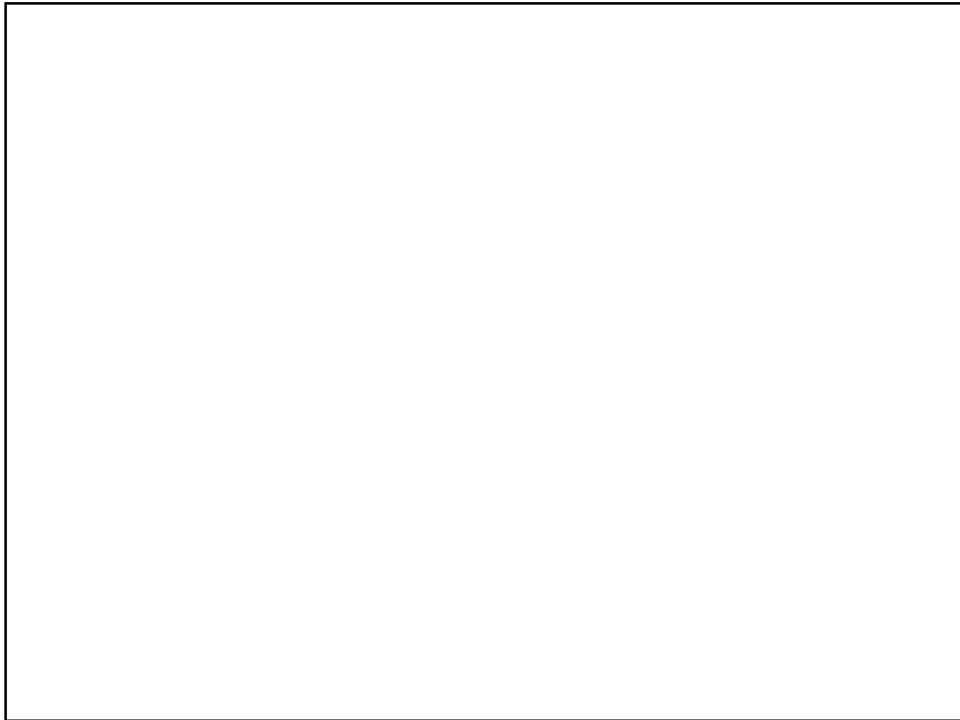
```

<!doctype html>
<html>
  <head> <!-- omissis --> </head>
  <body>
    <div>
      
      <h1>Amnesty International</h1>
    </div>
    <p> <!-- omissis --> </p>
    <hr />
  </body>
</html>
  
```

ESERCIZIO 1.1-00-10-01-01

Il DOM produce una rappresentazione ad albero del documento HTML.
 Un elemento del documento HTML è un nodo dell'albero ed il suo contenuto è il suo sottoalbero. I testi e gli elementi vuoti rappresentano i nodi foglia. Anche gli attributi sono rappresentati come dei nodi.
 Esiste quindi la classe nodo che ha attributi e metodi per ispezionare il padre, i figli ed i fratelli del nodo e per creare nuovi nodi e eliminarne di già presenti.

ESERCIZIO

A decorative graphic on the left side of the slide, featuring a vertical blue line with a fine grid pattern, and several blue circles of varying sizes arranged in a cluster.

JAVASCRIPT

Gestione dei form

ACCESSO AGLI ELEMENTI DI UN FORM

L'accesso agli elementi di un **form** avviene esattamente come per tutti gli altri elementi.

Stavolta oltre a disporre dei metodi **getElementById** e **getElementByName** (che però restituisce un vettore di elementi) possiamo usare anche l'attributo **forms**.

Document	
String	URL
String	title
Element	body
Collection[]	cookie
Collection[]	forms
Collection[]	images
Collection[]	links
void	write(String)
Element	getElementById(String)
Element[]	getElementsByTagName(String)

ESEMPIO

Vediamo un ampio esempio in cui dapprima operiamo il submit del form solo se tutti i campi sono stati riempiti e poi andiamo a leggere i dati inseriti dall'utente nella pagina successiva.

```

<body>
  <form id="unico" action="ingresso.html" method="GET">
    <div>
      <label for="cognome">Cognome</label>
      <input id="cognome" type="text" name="cognome" />
    </div>
    <div>
      <label for="nome">Nome</label>
      <input id="nome" type="text" name="nome" />
    </div>
    <input id="btn"
      type="submit"
      value="OK"
      onClick="controlla()" />
  </form>
  <script src="esercizio.js"></script>
</body>

```

ESEMPIO

Vediamo un ampio esempio in cui dapprima operiamo il submit del form solo se tutti i campi sono stati riempiti e poi andiamo a leggere i dati inseriti dall'utente nella pagina successiva.



```
function controlla() {
  var elem1 = document.getElementById("cognome");
  var elem2 = document.getElementById("nome");
  if ((elem1.value.length > 0) && (elem2.value.length > 0)) {
    var unicoForm = document.getElementById("unico");
    unicoForm.submit();
  } else {
    alert("Riempire tutti i campi!");
  }
}
```

```
<!-- omissis -->
<form id="unico" action="ingresso.html" method="GET">
<!-- omissis -->
```

ESEMPIO

Vediamo un ampio esempio in cui dapprima operiamo il submit del form solo se tutti i campi sono stati riempiti e poi andiamo a leggere i dati inseriti dall'utente nella pagina successiva.



```
<body>
  <script src="ingresso.js"></script>
</body>
```

```
var nome = getParametro("nome");
var cognome = getParametro("cognome");

document.write("<p>" + nome + "</p>");
document.write("<p>" + cognome + "</p>");
```

ESEMPIO

```
function getParametro(s) {
  // recupero la stringa dell'indirizzo
  var url = document.location.toString();

  // mi sposto all'inizio dei parametri
  var pos = url.indexOf("?");
  if (pos == -1) return "";

  // cerco il parametro come primo o come successivo al primo
  pos = url.indexOf("?"+s+"=");
  if (pos == -1) {
    pos = url.indexOf("&"+s+"=");
  }
  if (pos == -1) return "";

  // mi sposto all'inizio del valore
  pos += s.length+2;

  // mi recupero il valore
  var i = pos;
  var valore = "";
  while ( (i<url.length) && (url.charAt(i)!="&") ) {
    valore += url.charAt(i);
    i++;
  }
  return valore;
}
```

ESERCIZIO



Sviluppare un progetto Javascript che presenti uno slideshow di immagini con i file forniti. Cercando di andare oltre la prima o oltre l'ultima immagine si riceve un messaggio.

11-08-01