

**Formazione  
specialistica  
per sviluppatore  
front-end**





# Hello!

## Alessandro Pasqualini

Full-stack developer, appassionato di  
programmazione e sviluppo software in generale, amo  
le serie tv e i gatti

1

**Bootstrap: the most  
popular HTML, CSS, and  
JS library in the world.**



# Cosa permette di fare Bootstrap?

- Siti responsive
- Layout multi colonna
- Formattare semplicemente i form
- Barre di navigazione pre-formattate
- Tab per semplificare la gestione di molto contenuto
- Caroselli (ad esempio slider di immagini)
- Messaggi di alert

e molto altro ...



# Perchè usare Bootstrap?

- Compatibilità con i principali browser
- Semplicità di sviluppo di componenti complessi (modal, dropdown, accordions, tab, etc)
- Velocità di sviluppo grazie a moltissimi componenti "pronti all'uso"
- **Responsive:** ci fornisce già la struttura per realizzare un sito responsive (e tutti i suoi componenti lo sono di default)



# Bootstrap è una libreria

Bootstrap è una libreria, quindi non sostituisce completamente la scrittura di HTML/CSS/JS.

Ogni sito è diverso e ha requisiti diversi: necessita di scrivere comunque HTML e CSS custom.

Bootstrap ci aiuta fornendoci tanti componenti già fatti, ma sta a noi decidere se e come usarli (ed eventualmente adattarli alle nostre esigenze)



# Workflow con Bootstrap

Quando scriviamo un sito dobbiamo:

1. Creare la struttura base HTML
2. Includere il file CSS di bootstrap nell'HTML
3. Includere il file JS di bootstrap e jQuery nell'HTML
4. Aggiungere il link al nostro file CSS in cui scriveremo il nostro codice CSS
5. Usare i componenti di bootstrap e adattarli a quello che ci serve usando HTML/CSS
6. Eventualmente scrivere il nostro HTML e stilarlo normalmente con CSS se non c'è un componente adatto

2

# Bootstrap grid system



# Cos'è un layout a griglia?

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

- Funziona come una tabella: ci sono righe e colonne
- Ogni riga viene divisa in **12 "spazi"** di dimensione uguale
- Il programmatore crea le colonne scegliendo un numero intero di "spazi"
- Una colonna può avere al massimo **12 "spazi"**



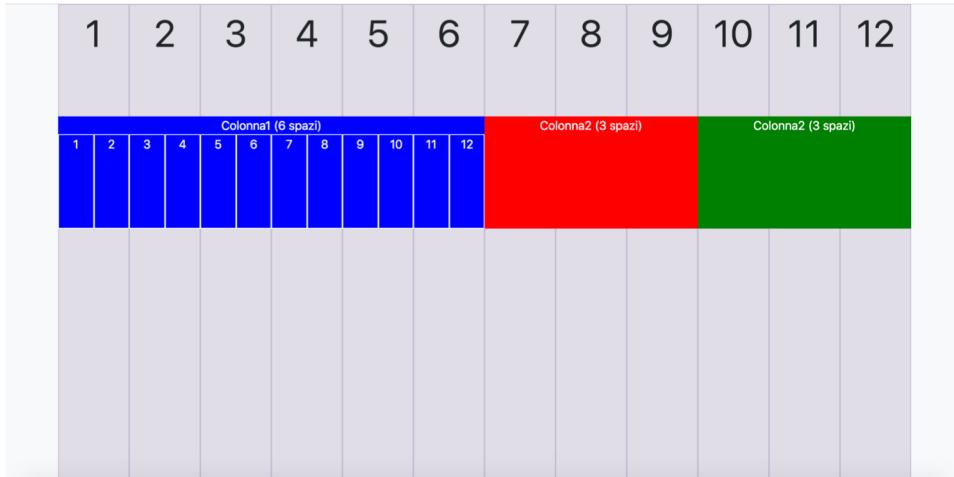
# Cos'è un layout a griglia?



- Funziona come una tabella: ci sono righe e colonne
- Ogni riga viene divisa in **12 "spazi"** di dimensione uguale
- Il programmatore crea le colonne scegliendo un numero intero di "spazi"
- Una colonna può avere al massimo **12 "spazi"**



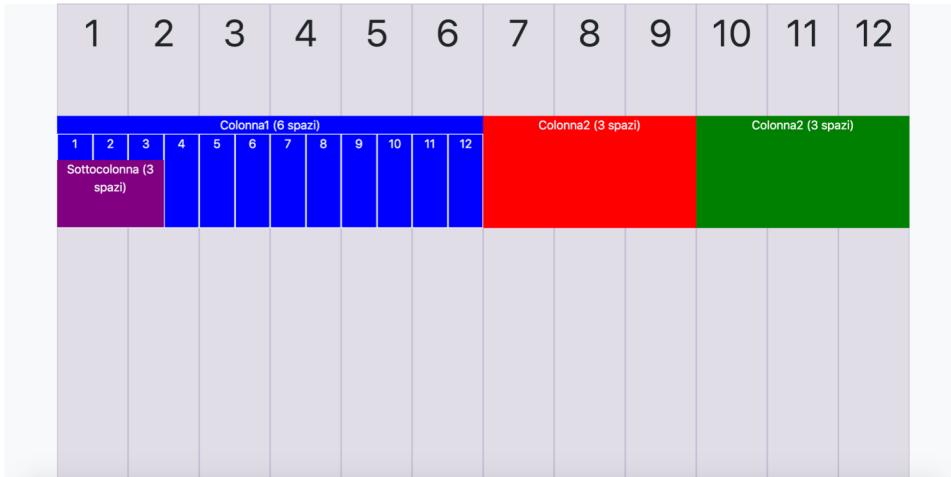
# Cos'è un layout a griglia?



- Ogni colonna che il programmatore ha creato può a sua volta essere divisa in altri 12 spazi.
- Il programmatore allora può creare altre "sottocolonne".



# Cos'è un layout a griglia?



- Ogni colonna che il programmatore ha creato può a sua volta essere divisa in altri 12 spazi.
- Il programmatore allora può creare altre "sottocolonne".



# Cos'è un layout a griglia?

The screenshot shows the homepage of the Indiana Museum of Art (IMA) website. The layout is organized into a grid system. At the top, there's a navigation bar with categories: VISIT, ABOUT, ART, SUPPORT, PROGRAMS, SHOP, and INTERACT. Below the navigation, the IMA logo is prominently displayed. To the left, there's a sidebar for "Visit Us" with links to Museum Hours, Get Directions, and Past Events. The main content area features several grid cells. One large cell on the right contains a banner for "100 Acres" at "THE VIRGINIA B. FAIRBANKS ART & NATURE PARK". Another cell below it has a red background with white text for "The Toby" and "Keep Us Free". To the left of these, there are smaller grid cells for various events like "IMA Stream", "IMA Calendar", and "Recent Videos". The bottom of the page has a footer with links for "IMA Membership", "Nourish Cafe", and "Recent Videos".

L'idea generale è quella di suddividere la pagina in 12 spazi.

Gli spazi sono usati per fare le colonne e ogni colonna può a sua volta essere divisa in altri 12 spazi e così via.

Gli elementi posti nella pagina devono essere multipli degli spazi (es. 2 spazi, 3 spazi etc)



# Vantaggi di un layout a griglia

Semplicità nella trasformazione dalla grafica "cartacea" a quella digitale.

**Responsive** (Bootstrap fornisce già un buon livello di responsive sui principali dispositivi)

**Layout uniforme** (naturale per l'occhio umano)

**Flessibilità** nelle piccole modifiche senza dover stravolgere l'intero layout della pagina



# Responsive

The screenshot shows the Bootstrap documentation homepage. The top navigation bar includes links for Home, Documentation, Examples, Themes, Expo, and Blog. A download button for version 4.3 is visible. The main content area features a search bar and a sidebar with navigation links for Getting started, Layout, Content, and Utilities. A central section discusses containers and includes an Adobe Stock offer for free images.

Desktop

The screenshot shows the Bootstrap documentation website as it appears on a mobile device. The layout is optimized for smaller screens, with the main content area displaying the 'Containers' section and its associated offer.

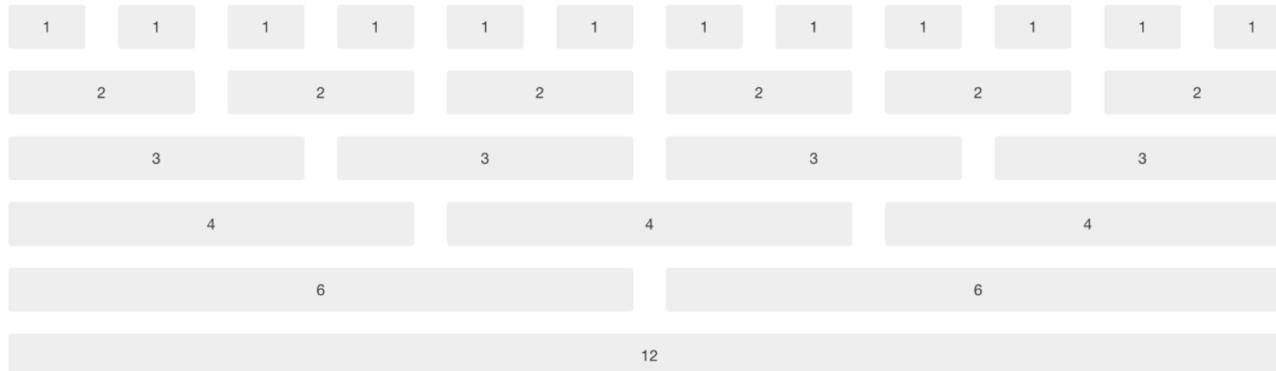
Mobile

The screenshot shows the Bootstrap documentation website as it appears on a tablet. The layout is larger than a mobile phone but smaller than a desktop, showing the 'Containers' section and its offer. It also includes a sidebar with navigation links.

Tablet



# Bootstrap grid system



- 12 spazi per riga
- Ogni colonna è distanziata dall'altra di una quantità chiamata gutter (30 px)
- Il gutter è diviso in due parti, una a sinistra e una a destra. (15 px a sinistra della colonna e 15 px a destra)



# Uso del grid system

La pagina è suddivisa in righe e ogni riga ha 12 spazi.

```
<div class="container">  
  <div class="row">  
    <div class="col-5">...</div>  
    <div class="col-7">...</div>  
  </div>  
</div>
```

5 spazi

7 spazi



# Cose da sapere quando si usa?

Le colonne (create usando tutti o in parte i 12 spazi della riga) sono spaziate da un **gutter**.

Il gutter è semplicemente uno **spazio tra una colonna e l'altra**.  
Bootstrap spazia le colonne di 15 px a sinistra e 15 px a destra  
(totale 30 px di gutter).

In bootstrap una riga è divisa in 12 spazi e quindi è possibile creare **colonne da minimo 1 spazio e massimo 12 spazi**. Non è possibile fare una colonna grande 1,5 spazi o 13 spazi per esempio.



# Cose da sapere quando si usa?

Se creiamo più colonne in una riga (sempre da minimo 1 spazio e massimo 12), ma la somma totale degli spazi di tutte le colonne supera 12, Bootstrap le redistribuisce su più righe in automatico.

```
<div class="row">
  <div class="col-4">4 spazi</div>
  <div class="col-9">9 spazi</div>
  <div class="col-3">3 spazi</div>
  <div class="col-12">12 spazi</div>
</div>
```





# Container della pagina

I container sono il componente base del sistema a griglia di bootstrap.

`<div class="container">...</div>` container a dimensione fissa, cambia solamente al raggiungimento del break point.

`<div class="container-fluid">...</div>` utilizza tutta la dimensione della pagina.



# Container "fluido"



```
<div class="container-fluid">
...
</div>
```

Copy

- Il container "fluido" ha dimensione massima pari al 100% della pagina (tutta la sua larghezza)
- La sua dimensione cambia restringendo la dimensione della pagina
- Non risponde ai breakpoint (sempre 100%)



# Container "normale"



Copy

```
<div class="container">  
  <!-- Content here -->  
</div>
```

- Il container "normale" ha una dimensione massima fissata
- La sua dimensione cambia al superamento del breakpoint
- Viene centrato automaticamente nella pagina



# Breakpoints

```
// Extra small devices (portrait phones, less than 576px)
// No media query for `xs` since this is the default in Bootstrap

// Small devices (landscape phones, 576px and up)
@media (min-width: 576px) { ... }

// Medium devices (tablets, 768px and up)
@media (min-width: 768px) { ... }

// Large devices (desktops, 992px and up)
@media (min-width: 992px) { ... }

// Extra large devices (large desktops, 1200px and up)
@media (min-width: 1200px) { ... }
```

2

Javascript

“

*Java is to Javascript what Car is  
to Carpet*

- Chris Heilmann



# Javascript vs Java

Java è un linguaggio completamente diverso da Javascript.

L'unica cosa che hanno in comune è una parte del nome.

Javascript ha avuto negli anni una brutta reputazione: era il responsabile di redirect, popup e il fulcro di tantissimi problemi di sicurezza.

I browser moderni controllano e impediscono a JS tutta una serie di "attività" per la sicurezza dell'utente.



# Inserire JS nella pagine

```
<script>  
    // Il mio JS  
</script>  
  
<script src="..."></script>
```

Javascript può essere incluso in due modi:

- Embedded nella pagina (similarmente al tag style)
- Inserendo il link del file JS (similarmente al tag link)



# Dove includere il JS nella pagina?

```
<!DOCTYPE html>
<html>
  <head>
    <title>Il mio primo JS</title>
    <script>
      // JS
    <script>
  </head>
  <body>
    <script>
      // JS
    <script>
  </body>
</html>
```

Javascript può essere incluso sia nel tag **head**, oppure nel tag **body**.

Non fa (quasi) differenza.



# Non fa davvero differenza?

In realtà una pagina web **fatta bene** deve includere il JS come ultima cosa del tag **body**

Il browser interpreta la pagina dall'inizio alla fine. Se il JS richiede operazioni pesanti la pagina verrà non verrà renderizzata finché non è stato interpretato (e scaricato) tutto il JS.

E' più importante che la pagina si veda bene piuttosto che sia "dinamica". I tempi di interpretazione "massimi" del JS dovrebbero comunque attestarsi intorno 1s. Massimi non medi!



# Non fa davvero differenza?

Se il JS è scritto in file esterno (e magari pesa parecchio) se è incluso nel tag head il browser impegnerà risorse per scaricare il JS, privandole ad altre risorse più importanti nell'immediato (es. immagini/CSS)

Se il JS è embedded nella pagina e inserito nel tag head il browser inizierà la sua interpretazione subito appena incontra il tag script, privando risorse al rendering della pagina.



# Il JS va sempre inserito nel body

Se il js è esterno oppure embedded ed è posizionato come ultima risorsa del body, quando il browser lo incontra avrà terminato il rendering della pagina oppure sarà in procinto di farlo e quindi non vengono "sprecate" risorse.

Questa regola generale non si applica ad alcuni script particolari (ad esempio il tracciamento dell'utente o il blocco dei cookies) che richiedono di essere inizializzati prima del contenuto (e del resto del codice JS)



# Commenti

In JS esistono due tipi di commento:

- Commenti di una sola riga
- Commenti multiriga

```
<script>
    // Singola riga
</script>
```

```
<script>
    /* Questo commento
       e' composto da
       più righe */
</script>
```



# Punto e virgola

Javascript non obbliga la separazione delle istruzioni attraverso il punto e virgola ":".

E' sempre buona regola metterlo comunque per separare le istruzioni e suddividere meglio il codice.

```
<script>  
    console.log('Senza ;')  
</script>
```

```
<script>  
    console.log('Con ;');  
</script>
```



# Istruzione console.log

```
<script>  
    console.log('test');  
</script>
```

x Expression  
not available  
test  
>

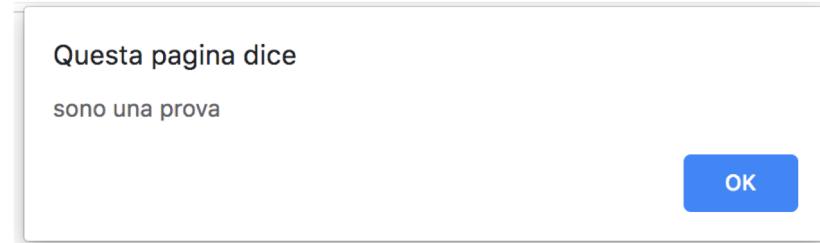
L'istruzione console.log ci permette di mostrare in console delle informazioni. E' perfetta nello sviluppo per stampare a console delle variabili con contenuto "ignoto", frutto di qualche operazione.

Quando il sito va in produzione la console deve essere "pulita", ovvero non devono rimanere nel codice istruzioni di console.log.



# Istruzione alert

```
<script>  
    alert('sono una prova');  
</script>
```



L'istruzione **alert** ci permette di far apparire un box con in cui mostrare un messaggio di attenzione all'utente. Il box contiene un bottone OK per chiudere la finestra. Sono sconsigliate in un sito ed è meglio usare qualcosa di più carino tipo i modal di bootstrap.



# Cos'è una variabile?

Una variabile è un "contenitore" usato per contenere qualche informazione:

In Javascript è possibile memorizzare molte informazioni differenti:

- Numeri interi (number)
- Numeri a virgola mobile (number)
- Un carattere (string)
- Una stringa, ovvero un insieme di caratteri (string)
- Un valore booleano, ovvero **vero** o **falso** (boolean)
- Un insieme di variabili (array)
- Un oggetto (object)
- Una funzione (function)

```
var variabile;
```



# Dare un nome alle variabili

Un nome valido di una variabile:

- Non **deve** coincidere con una parola chiave del linguaggio
- Non può iniziare con un numero
- Non può contenere caratteri speciali (spazi, lettere accentate, il trattino, etc)
- Può contenere un underscore
- Può contenere o iniziare con il simbolo del dollaro (\$)



# I tipi di dato

Ogni variabile possiede un tipo di dato, ovvero:

- L'insieme dei valori permessi
- Le operazioni che si possono effettuare

Javascript viene detto loosely typed, ovvero non è necessario specificare a priori il tipo della variabile ma viene "assegnato" automaticamente da Javascript.

Questo è fonte di molti errori ed è sempre bene prestare particolare attenzione a cosa stiamo cercando di fare!



# I tipi di dato: stringe

```
<script>  
    var string1 = 'abcd';  
    var string2 = "abcd";  
    var empty = '';  
</script>
```

Le stringe sono sequenze di caratteri.

Le stringe sono delimitate da '' oppure da ''. Sono validi entrambi e non fanno differenza.

Scegliete pure quello che vi piace ma state **coerenti**: fare le cose sempre nello stesso modo!



# I tipi di dato: stringhe

Le stringhe possono essere concatenate: ovvero "sommare" due stringhe per formarne una composta dalla giustapposizione della prima e della seconda. L'operatore di concatenazione è il +

```
var string1 = 'abcd';
var string2 = "efgh";
var string3 = string1 + string2;
console.log(string3);
```

```
x Expression
not available
abcdefg
>
```



# I tipi di dato: numeri

```
var numero1 = 6;  
var numero2 = 3.14;
```

Javascript non fa differenza tra numero intero (es 6) e i numeri decimali (es 3.14).

Si usa la notazione americana, quindi il punto al posto della virgola.

Naturalmente sono supportate tutte le principali operazioni sui numeri.



# I tipi di dato: boolean

```
var vero = true;  
var falso = false;
```

Javascript possiede valori booleani, ovvero valori che assumono solamente uno di due valori: TRUE o FALSE.

Sono utilissimo soprattutto nella logica condizionale: if, while, etc dove si decidere di eseguire del codice solamente se si verifica una determina condizione.



# I tipi di dato: array

```
var giorniDellaSettimana = [  
    "lunedì",  
    "martedì",  
    "mercoledì",  
    "giovedì",  
    "venerdì",  
    "sabato",  
    "domenica"  
];
```

Gli array sono un insieme numerato di variabili tutte dello stesso tipo.

E' possibile accedere ad uno specifico elemento attraverso il suo indice (gli indici partono da 0)



# I tipi di dato: array

```
var giorniDellaSettimana = [  
    "lunedì",  
    "martedì",  
    "mercoledì",  
    "giovedì",  
    "venerdì",  
    "sabato",  
    "domenica"  
];  
console.log(giorniDellaSettimana[0]);  
console.log(giorniDellaSettimana[6]);
```

x Expression  
not available

lunedì

domenica

>



# I tipi di dato a cosa servono?

Il tipo di dato serve a:

- Indicare l'insieme dei valori permessi
- Le operazioni permesse (es. operazioni matematiche, etc)

I tipi di dato sono usati dal linguaggio per comprendere "cosa fare con quel dato" e "come relazionarsi" con altri dati (dello stesso tipo o di altri tipi).



# I tipi di dato a cosa servono?

```
var n1 = 1;  
var n2 = '2';  
var n3 = n1 + n2;  
console.log(n3);
```

```
x Expression  
not available  
12  
>
```

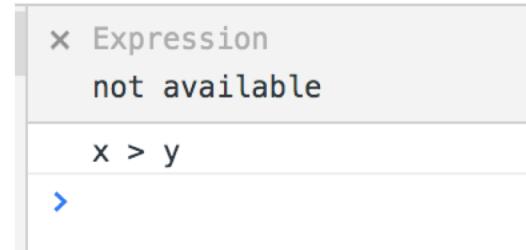
```
var n1 = 1;  
var n2 = 2;  
var n3 = n1 + n2;  
console.log(n3);
```

```
x Expression  
not available  
3  
>
```



# Controllo di flusso: if

```
var x = 10;  
var y = 9;  
  
if (x > y) {  
    console.log('x > y');  
} else {  
    console.log('x < y');  
}
```





# L'operatore ternario

L'operatore ternario (chiamato anche istruzione condizionale) è una "scorciatoia" dell'if in alcuni casi:

```
var variabile;  
if (condizione) {  
    variabile = 'Condizione vera';  
} else {  
    variabile = 'Condizione falsa';  
}
```

```
var variabile = condizione ? 'Condizione vera' : 'condizione falsa';
```



# Iterazioni: for

```
<script>
    for (var i = 0; i < 10; i++) {
        console.log(i);
    }
</script>
```

x Expression  
not available

0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
> |



# Iterazioni: while

```
var x = 10;  
var y = 7;  
while (x > y) {  
    console.log(x);  
    x--;  
}
```

x Expression  
not available

10

9

8

> |



# Istruzione switch

```
switch (variabile) {  
    case '1':  
        console.log('1');  
        break;  
    case '2':  
        console.log('2');  
        break;  
  
    default:  
        console.log('default');  
}
```

```
if (variabile == '1') {  
    console.log('1');  
} else if (variabile == '2') {  
    console.log('2');  
} else {  
    console.log('default');  
}
```



# Istruzione switch

Switch è un costrutto che permette di sostituire una cascata di if e di migliorare la leggibilità del codice.

E' meno potente del costrutto if e ogni **case** deve obbligatoriamente essere "interrotto" da un istruzione **break**;



# Istruzione switch (con break!)

```
var variabile = '1';

switch (variabile) {
    case '1':
        console.log('1');
        break;

    case '2':
        console.log('2');
        break;

    default:
        console.log('default');
}
```

x Expression  
not available

1

> |



# Istruzione switch (senza break!)

```
var variabile = '1';

switch (variabile) {
    case '1':
        console.log('1');

    case '2':
        console.log('2');

    default:
        console.log('default');
}
```

x Expression  
not available

1
2
default

> |



# Le funzioni

Le funzioni sono un modo per raggruppare insieme del codice in modo da poterlo rieseguire senza doverlo riscrivere.

Le funzioni derivano dal concetto matematico di funzione dove dati determinati parametri di ingresso (**input**) ci vengono ritornati uno o più parametri "calcolati" dalla funzione (**output**).

Sono utilizzate anche per raggruppare logicamente il codice.



# Le funzioni

Le funzioni in Javascript posseggono (quasi sempre) un nome e dei parametri di ingresso e restituiscono un valore.

Se la funzione non ha un nome si chiama **funzione anonima**.

Le funzioni anonymous sono molto usate per definire callback.

I nomi delle funzioni, così come i nomi delle variabili, devono essere univoci all'interno di un blocco di codice.



# Parametri

I parametri di ingresso non sono altro che delle variabili che vengono passate alla funzione.

Sono usate per passare dei valori necessari all'esecuzione del codice della funzione.

Possono essere in qualsiasi numero e di qualsiasi tipo e numero.



# Parametri

```
function test(p1, p2, p3) {  
    console.log(typeof p1);  
    console.log(typeof p2);  
    console.log(typeof p3);  
}
```

```
test(1,2,3);  
console.log('-----');  
test(1);
```

```
x Expression  
not available  
number  
number  
number  
-----  
number  
undefined  
undefined
```

>



# Parametri

Se un parametro non viene passato durante l'invocazione della funzione, Javascript gli assegna il tipo speciale **undefined**

Undefined significa che la variabile/parametro non è definito.

Prestare attenzione a questi "casi particolari" e passare sempre tutti i parametri alla funzione. E' bad practise passare meno parametri a meno che non siano definiti dei valori di default.



# Parametri con valore di default

```
function test(parametro = false) {  
    console.log(parametro);  
}  
  
test(true);  
test();
```

x Expression  
not available

---

true

---

false

---

>



# Parametro vs argomento

Un **parametro** è un valore che una funzione prevede che gli venga passato. Quindi è incluso nella sua dichiarazione:

```
function laMiaFunzione(parametro1, parametro2) {  
}
```

Un **argomento** è l'effettivo valore passato alla funzione, ovvero il valore che inseriamo nella sua chiamata:

```
laMiaFunzione(argomento1, argomento2);
```



# L'istruzione return

L'istruzione `return` è la funzione che permette alla funzione di ritornare un valore al termine della funzione.

Può ritornare un valore unico (gli array sono comunque un valore unico).

Può essere usata per interrompere l'esecuzione della funzione se usata da sola.



# Variabili locali e variabili globali

Le variabili definite all'esterno delle funzioni sono chiamate variabili globali.

Le variabili definite all'interno di un blocco di codice (es. funzioni) sono chiamate variabili locali.

La differenza sostanziale è che le variabili globali esistono globalmente, ovvero sono accessibili anche all'interno della funzione, mentre quelle locali hanno vita solamente durante l'esecuzione della funzione. Quando la funzione termina vengono distrutte.

3

jQuery



# Selezionare un elemento

```
<div id="mio-id">...</div>
<div class="mia-classe">...</div>

<script>
  $('#mio-id');
  $('.mia-classe');
</script>
```

\$ ci permette di selezionare uno o più elementi dal DOM e ottenere una referenza a loro semplicemente usando i classi selettori CSS.



# Aggiungere / rimuovere classi

```
<div id="mio-id">...</div>
```

```
<script>  
    $('#mio-id').addClass('classe1');  
    $('#mio-id').removeClass('classe1');  
</script>
```

Per aggiungere una classe ad un elemento:

```
$('....').addClass('classe');
```

Mentre per rimuovere una classe da un elemento:

```
$('....').removeClass('classe');
```



# Aggiungere/rimuovere CSS inline

```
<div id="mio-id">...</div>
```

```
<script>
    $('#mio-id').css('background-color', 'red');
</script>
```

E' possibile aggiungere regole CSS inline semplicemente con:  
\$('...').css('regola', 'valore');



# Mostrare/nascondere un elemento

```
<div id="mio-id">...</div>
```

```
<script>
    $('#mio-id').hide();
    $('#mio-id').show();
</script>
```

Per nascondere un elemento, ovvero applicare `display: none;`  
`$('#...').hide();`

Per mostrare un elemento, ovvero applicare `display: block;`  
`$('#...').show();`



# Testare la presenza di elementi

```
<div id="mio-id">...</div>

<script>
    if ($('#mio-id').length) {
        ...
    }
</script>
```

Per testare se uno o più elementi esistono all'interno della pagina:

`$('....').length`

Sarà maggiore di zero, ovvero uguale al numero di elementi selezionati



# Attributi VS proprietà

Un attributo aggiunge delle informazioni ad un elemento (es. href).

Una proprietà descrive le caratteristiche di un elemento (es. checked)

```
<a href="#">link</a> <!-- href è un attributo -->
```

```
<input type="checkbox" checked> <!-- checked è una proprietà-->
```



# Attributi

```
<a id="mio-a" href="https://google.com">...</a>
```

```
<script>
    console.log($('#mio-a').attr('href'));
</script>
```

Per accedere agli attributi:

```
$(...).attr('nome_attributo');
```

Per impostare un attributo

```
$(...).attr('nome_attributo', 'valore_attributo');
```

x Expression  
not available

⚠ A cookie associated with this page was used to deliver cookies with cross-site requests and see more details at <https://google.com>

⚠ A cookie associated with this page was used to deliver cookies with cross-site requests and see more details at <https://google.com>

> |



# Proprietà

```
<input type="checkbox" id="check">  
<script>  
    $('#check').prop('checked', true);  
</script>
```

Per accedere ottenere il valore di una proprietà:

```
$(...).prop('nome_ proprietà');
```

Per impostare un proprietà

```
$(...).prop('nome_proprietà', 'valore');
```



# Attributi 'data-'

E' possibile usare degli attributi aggiuntivi non standard agli elementi (es. per contenere delle informazioni che ci fornisce il server) usando gli attributi data-.

```
<div data-mio-attributo="mio-valore"></div>
```



# Attributi 'data-'

```
<div data-mio-attributo="mio-valore" id="mio-id"></div>
<script>
    console.log($('#mio-id').data('mio-attributo'));
</script>
```

Per accedere ottenere il valore di un attributo data:

```
$(...).data('nome-senza-data');
```

Per impostare un attributo data:

```
$(...).data('nome-senza-data', 'valore');
```

The screenshot shows a browser developer tools console window. The first line of text is "Expression not available". Below it, there are two warning messages: "A cookie associated with this page was delivered via cookie delivery mechanism and see more details" and another identical message. At the bottom of the list, the value "mio-valore" is displayed.

```
x Expression
not available
⚠ A cookie associated with this page was delivered via cookie delivery mechanism and see more details
⚠ A cookie associated with this page was delivered via cookie delivery mechanism and see more details
mio-valore
```



# Modificare il contenuto

```
<div id="mio-id"><div>
<script>
  $('#mio-id').html($('<p>test</p>'));
</script>
```

test

Per accedere al contenuto:

```
$(...).html();
```

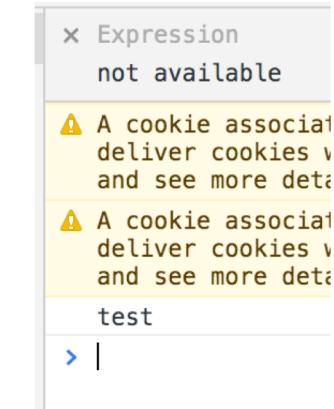
Per impostare il contenuto:

```
$(...).html('valore');
```



# Accedere al contenuto di un input

```
<input id="mio-i" value="test">  
<script>  
    console.log($('#mio-i').val());  
</script>
```



Per accedere al contenuto:

```
$('....').val();
```

Per impostare il contenuto:

```
$('....').val('valore');
```



# Eventi ed event handler

jQuery permette di associare un event handler ad un evento, ed il click di un bottone. La sintassi base è:

```
$(..).on('evento', function () {  
    // Codice  
});
```

Allo stesso modo è possibile anche eliminare un event handler

```
$(..).off('evento');
```



# Transizioni di jQuery

jQuery possiede alcune transizioni di base utili per mostrare o nascondere del contenuto:

- fadeIn /fadeOut
- slideDown/slideUp



# **\$this**

Quando usate jQuery e siete all'interno di una funzione (ad esempio perchè state scrivendo un event handler) a volte capita di dover accedere all'elemento che ha generato l'evento.

jQuery mette a disposizione questa sintassi:  
**\$this**



# preventDefault();

Il browser assegna delle azioni di default ad alcuni elementi (link, form, etc).

A volte capita di dover indicare al browser di NON eseguire l'azione di default ed eseguire il nostro codice. A questo scopo possiamo usare **preventDefault()**:

```
$('a').on('click', function (e) {  
    e.preventDefault();  
    // Codice  
});
```



# **\$(document).ready(...)**

\$(document).ready(...) ci permette di registrare un callback (una funzione) da eseguire quando il browser emettere l'evento "pagina pronta".

Questo ci permette di ritardare l'esecuzione del codice fino a che la pagina non è pronta.

```
$(document).ready(function () {  
    console.log('La pagina è pronta');  
});  
console.log('Scrivi subito');
```

Scrivi subito
La pagina è pronta



# **\$(window).on('load', ...)**

`$(document).load(...)` ci permette di registrare un callback (una funzione) da eseguire quando il browser emettere l'evento "pagina completamente caricata" comprensiva di immagini, iframe etc.

```
$(window).on('load', function () {  
    console.log('La pagina è caricata');  
});  
$(document).ready(function () {  
    console.log('La pagina è pronta');  
});  
console.log('Scrivi subito');
```

Scrivi subito

La pagina è pronta

La pagina è caricata

.



# **\$(window).on('load', ...)**

Attenzione che **load** funziona solo su **window** e non **document**.

```
$(document).on('load', function () {  
    console.log('La pagina è caricata');  
});
```



```
$(window).on('load', function () {  
    console.log('La pagina è caricata');  
});
```

La pagina è caricata





# Creare elementi con jQuery

Capita spesso di dover generare elementi HTML dinamicamente usando Javascript in risposta a certi eventi (callback, richieste/risposte con il server mediante Ajax, etc)

jQuery ci permette di generare elementi utilizzando un sintassi molto semplice:

```
var elemento = $('

</div>');


```



# Creare elementi con jQuery

```
var elemento = $('

</div>');


```

jQuery ci ritorna un oggetto che rappresenta il nostro nuovo elemento.

A questo punto possiamo modificarlo ad esempio aggiungendo classi, css, event handler (es. click), etc

Successivamente dobbiamo ricordarci di includerlo all'interno della pagina.



# Creare elementi con jQuery

```
var elemento = $('

</div>');  
$('body').append(elemento);


```

Quando jQuery crea un elemento **non lo aggiunge** automaticamente al DOM (e quindi al documento) e di conseguenza l'elemento non viene reso (rendered) dal browser.

Per questo jQuery ci mette a disposizione un paio di metodi (funzioni) per aggiungere l'elemento al DOM.



# Creare elementi con jQuery

```
var elemento = $('

</div>');  
$('body').append(elemento);


```

Innanzitutto dobbiamo scegliere un elemento (ad esempio il body) come "punto di ingresso" per inserire il nuovo elemento. E poi chiamare una delle seguenti funzioni:

- **append** aggiunge l'elemento come **ultimo figlio** del "punto di ingresso".
- **prepend** aggiunge l'elemento come **primo figlio** del "punto di ingresso".



# Creare elementi con jQuery

```
var elemento = $('

</div>');  
$('body').insertBefore(elemento);


```

Oppure possiamo scegliere se inserire il nuovo elemento prima o dopo il nostro elemento "punto di ingresso", ovvero decidiamo di inserirlo come *sibling* (fratello).

- **insertBefore** aggiunge l'elemento sopra il "punto di ingresso".
- **insertAfter** aggiunge l'elemento sotto il "punto di ingresso".



# Templating Javascript

Le stringhe templating sono racchiuse dal carattere backtick (` ), che non è l'apice singolo) e possono comprendere più linee.

```
var tpl = `<div class="myclass">
    <p>Sono un template</p>
</div>`;
```

Naturalmente le stringhe di templating sono solamente stringhe e non un elemento. Dovremo comunque fare ricorso a jQuery per costruire gli elementi ed inserirli nel DOM.



# Templating Javascript

```
var tpl = `<div class="myclass">
    <p>Sono un template</p>
</div>`;

var elemento = $(tpl);
$('body').append(elemento);
```

```
var div = $('<div></div>')
    .addClass('myclass');

var p = $('<p></p>')
    .text('Sono un template');

var div.append(p);
$('body').append(div);
```



# Templating Javascript

La vera potenza delle stringhe di templating di Javascript è la possibilità di usare dei **segnaposto**.

Un segnaposto non è altro che un'espressione Javascript (del codice javascript).

```
var text = 'Sono un template';
var tpl = `<div class="myclass">
            <p>${text}</p>
        </div>`;
```

```
var elemento = $(tpl);
$('body').append(elemento);
```



# Templating Javascript

I segnaposti sono composti in questo modo:

`${espressione}`

Possono essere inclusi direttamente nella stringa template:

```
var tpl = `1 + 1 = ${1+1}`;
```

4

# Oggetti Javascript



# Oggetti

Un oggetto è una collezione di dati e/o funzionalità correlati.

Nella programmazione ad oggetti si "uniscono" dati e funzioni (chiamate metodi).

In javascript sono molto utilizzati anche come "configurazione" di alcune librerie. Es. slick

```
$('.one-time').slick({  
    dots: true,  
    infinite: true,  
    speed: 300,  
    slidesToShow: 1,  
    adaptiveHeight: true  
});
```



# Proprietà degli oggetti

Gli oggetti sono variabili e quindi si istanziano (creano):

```
var persona = {};
```

Javascript assegna il tipo *object* alla variabile che contiene un oggetto.

Gli oggetti possono essere visti come un insieme di associazioni *chiave-valore* chiamate *proprietà*. Le proprietà sono delle variabili e quindi possono contenere tutto quello che una variabile può contenere.



# Proprietà degli oggetti

```
var persona = {  
    nome: 'John Smith',  
    anni: 33  
};  
  
console.log(persona);
```

▼ {*nome: "John Smith", anni: 33*} ⓘ  
  *nome: "John Smith"*  
  *anni: 33*  
► \_\_proto\_\_: Object

Naturalmente è possibile accedere ad ogni proprietà singolarmente usando la notazione `oggetto.proprieta`

```
var persona = {  
    nome: 'John Smith',  
    anni: 33  
};
```

```
console.log(persona.nome);
```

John Smith

5

JSON



# JSON

JSON è un formato per il trasporto dei dati e deriva dalla notazione per gli oggetti di Javascript.

JSON è indipendente dal linguaggio usato. Si può usare anche con altri linguaggi di programmazione, ad esempio PHP, Java, etc

JSON sta sostituendo XML (un formato "simile" ad HTML) per il trasporto dei dati perchè è molto leggero (poca formattazione) ed è molto facile da leggere anche per gli umani.



# XML VS JSON

```
{  
  "id": 123,  
  "title": "Object Thinking",  
  "author": "David West",  
  "published": {  
    "by": "Microsoft Press",  
    "year": 2004  
  }  
}
```

JSON

```
<?xml version="1.0"?>  
<book id="123">  
  <title>Object Thinking</title>  
  <author>David West</author>  
  <published>  
    <by>Microsoft Press</by>  
    <year>2004</year>  
  </published>  
</book>
```

XML



# La sintassi

JSON deriva dalla sintassi degli oggetti di Javascript e quindi è molto simile alla loro dichiarazione.

```
{  
    "prop1": valore1,  
    "prop2": valore2,  
    ...  
}
```

JSON supporta stringhe, numeri, array, oggetti, boolean, null (è un tipo speciale)

5

AJAX



# Ajax

Ajax è la tecnologia più usata per realizzare applicazioni interattive. Il concetto che sta alla base è quello di scambiare dati con il server senza dover ricaricare la pagina (ovvero eseguire la chiamata in background).

jQuery ci permette di realizzare delle chiamate AJAX con pochissimo sforzo.

Le chiamate AJAX sono chiamate HTTP classiche (GET, POST, etc).

AJAX originariamente usava XML come formato di trasporto dei dati ma è stato quasi completamente sostituito da JSON (e da altri formati)



# \$load

\$load è un metodo di jQuery (\$ di jQuery è un oggetto) che ci permette di caricare del contenuto (principalmente HTML) in modo asincrono.

```
<button id="clICCami">ClICCami</button>
<div id="contenuto"></div>
<script>
    $('#clICCami').click(function () {
        $('#contenuto').load('snippet.html', function () {
            console.log('Contenuto caricato');
        });
    }
</script>
```



# \$.get

\$.get è un metodo di jQuery che ci permette di effettuare una richiesta HTTP GET per prelevare del contenuto in maniera asincrona.

```
$(document).ready(function () {  
    $.get('users.json', function (usernames) {  
        console.log(usernames);  
    });  
});
```



# \$.**post**

\$.**post** è un metodo di jQuery che ci permette di effettuare una richiesta HTTP POST per inviare del contenuto al server in modo asincrono.

\$.**post** accetta un oggetto Javascript contenente i dati da inviare, quindi il nostro compito è crearlo inserendo i dati da comunicare al server.

\$.**post** permette anche di impostare un il tipo dei dati che il server ci ritornerà (ad esempio json). Se non è impostato viene considerato testo e quindi l'eventuale json NON è automaticamente trasformato in un oggetto Javascript.



# \$ .post

```
$.post(  
    'url_della_pagina',  
    { ... },  
    function (risposta) {  
        ...  
    },  
    'tipo_dei_dati_in_risposta_dal_server'  
);
```



# \$.ajax

jQuery mette a disposizione dei programmati anche un metodo chiamato `$.ajax` che permette di avere un controllo maggiore sulla richiesta asincrona.

`$.load`, `$.get` e `$.post` sono solamente delle "scorciatoie" per `$.ajax`.

Nella documentazione di jQuery è possibile trovare tutti i parametri accettati da questi metodi per modificarne il comportamento. Ad esempio è possibile inserire nella richiesta HTTP headers aggiuntivi.



# Il problema degli event handler

jQuery ci permette di registrare delle funzioni da eseguire quando si verifica un certo evento. es. click

```
$('#id').click(function () { ... });
```

Questi event handler sono però registrati al caricamento della pagina sul contenuto attuale. Se carichiamo del nuovo contenuto HTML gli event handler NON vengono registrati nel il nuovo codice.



# Il problema degli event handler

```
<button class="clICCami">ClICCami 1</button>
<script>
    $('.clICCami').click(function () {
        alert('ok');
    });
</script>
```

Se ora aggiungiamo un nuovo bottone attraverso del codice JS (magari caricandolo con \$.load)...



# Il problema degli event handler

```
<button class="cliccammi">Cliccammi 1</button>
```

```
<button class="cliccammi">Nuovo bottone</button>
```

```
<script>
  $('.cliccammi').click(function () {
    alert('ok');
  });
</script>
```

L'event handler non funziona per il nuovo bottone



# Il problema degli event handler

La soluzione è registrare l'event handler a livello di document

```
$('.cliccammi').click(function () {  
    alert('ok');  
});
```

```
$(document).click('.cliccammi', function () {  
    alert('ok');  
});
```



# CORS

CORS (Cross Origin Resource Sharing) è un meccanismo usato per indicare al browser che l'applicazione in esecuzione su una specifica origine (il dominio) ha l'autorizzazione di accedere a risorse su un'altra origine (altro dominio).

Per ragioni di sicurezza i browser limitano l'utilizzo di CORS per proteggere l'utente da situazioni pericolose.

Tutte le richieste CORS che provengono da script (Javascript) vengono sottoposte al controllo del browser prima di consentire la richiesta.

6

# Callback



# Callback

```
var laMiaFunzione = function () {  
    console.log('La mia funzione');  
}  
  
laMiaFunzione(); // Stampa 'La mia funzione'
```

Quindi è possibile dichiarare una funzione e salvarla all'interno di una variabile. Successivamente è possibile invocarla (eseguirla) semplicemente utilizzando il nome della variabile al posto del nome della funzione.



# Callback

```
var laMiaFunzione = function () {  
    console.log('La mia funzione');  
}  
  
laMiaFunzione();
```

```
function laMiaFunzione() {  
    console.log('La mia funzione');  
}  
  
laMiaFunzione();
```

Entrambe le funzioni si comportano in maniera completamente identica e possono essere invocate allo stesso modo.

L'unica differenza tra le due dichiarazioni è che la seconda esplicita il nome della funzione, mentre la prima è implicitamente il nome della variabile.



# Callback

Se Javascript permette di salvare le funzioni all'interno di una variabile cosa ci impedisce di passarle come argomenti di una funzione?

Un **callback** è una funzione passata come argomento di un'altra funzione.



# Callback

```
// Dicho una funzione che accetta un callback
function funzioneDiTest(callback) {
    callback();
}
```

```
// Dicho una funzione all'interno di una variabile
var laMiaFunzione = function () {
    console.log('La mia funzione');
}
```

```
// Chiamo la prima funzione
funzioneDiTest(laMiaFunzione);
```



# Callback

```
// Dicho una funzione che accetta un callback
function funzioneDiTest(callback) {
    callback();
}
```

```
// Dicho una funzione all'interno di una variabile
var laMiaFunzione = function () {
    console.log('La mia funzione');
}
```

```
// Chiamo la prima funzione
funzioneDiTest(laMiaFunzione);
```

La mia funzione





# Callback

A volte è necessario dichiarare una funzione "temporanea", ovvero una funzione che serve solo una volta e non ci interessa salvarla da qualche parte: non la useremo mai più!

Allora al posto di dichiarare una variabile in una funzione e successivamente passare questa variabile ad una funzione come callback possiamo saltare questo passaggio: passiamo direttamente la funzione.

I parametri di una funzione (function (parametro1, parametro2, ...) {...}) sono variabili!



# Callback

```
// Dicho una funzione che accetta un callback
function funzioneDiTest(callback) {
    callback();
}
```

```
// Chiamo la prima funzione passando direttamente la
// funzione come argomento
funzioneDiTest(function () {
    console.log('La mia funzione');
});
```



# Callback

```
// Dicho una funzione che accetta un callback
function funzioneDiTest(callback) {
    callback();
}
```

```
// Chiamo la prima funzione passando direttamente la
// funzione come argomento
funzioneDiTest(function () {
    console.log('La mia funzione');
});
```

La mia funzione





# A cosa servono i callback?

Possiamo passare una funzione a come argomento ad un'altra funzione ma perché farlo?

Il codice Javascript che scriviamo viene eseguito principalmente al verificarsi di qualche evento: l'utente clicca su un bottone, la richiesta ajax è finita, la pagina è pronta etc

Quindi non è possibile conoscere a priori il flusso che il programma Javascript seguirà ma saranno gli eventi (o l'utente) che determineranno l'esatta sequenza di esecuzione.



# A cosa servono i callback?

Visto che non possiamo conoscere a priori la sequenza (e quando) il nostro codice verrà eseguito dobbiamo prevedere un meccanismo capace di garantire comunque la corretta esecuzione del codice.

Questo meccanismo è fornito dai callback!

Attraverso i callback possiamo prevedere del codice che verrà eseguito solo se un certo evento si presenta.



# A cosa servono i callback?

```
$('#pulsante').click(function () {  
    console.log('Sono un callback!');  
});
```

Quando definiamo un event handler (il codice da eseguire quanto un evento si verifica) stiamo in realtà definendo un callback.

Quello che stiamo dicendo a Javascript di fare è: quando l'utente clicca sul pulsante allora esegui la funzione che ti passo come argomento, ovvero esegui il callback!

**1**

# **Versionamento del software: GIT**



# Cos'è il versionamento

I sistemi di controllo di versione sono strumenti software che permettono ad un team di gestire modifiche al codice durante un periodo di tempo (tipicamente la vita dell'intero progetto).

Questi sistemi tengono traccia di ogni modifica apportata al codice in uno speciale database.

Se viene commesso un errore, i developer possono "riportare indietro l'orologio" e comparare vecchie versioni del codice senza causare ulteriori problemi.



# Versionamento **NON** è backup

L'idea fondamentale del controllo di versione è mantenere diverse revisioni della stessa unità di informazione (es. codice).

L'idea di backup è mantenere al sicuro l'ultima versione dell'informazione (es. codice). Le vecchie versioni possono essere sovrascritte o eliminate.



# Repository

Il concetto di repository è centrale e fondamentale quando si parla di VCS.

Un repository è simile ad una cartella del pc dove sono contenuti tutti i file del progetto e tutte le loro revisioni.

Una modifica ad un file in terminologia VCS è chiamata revisione.

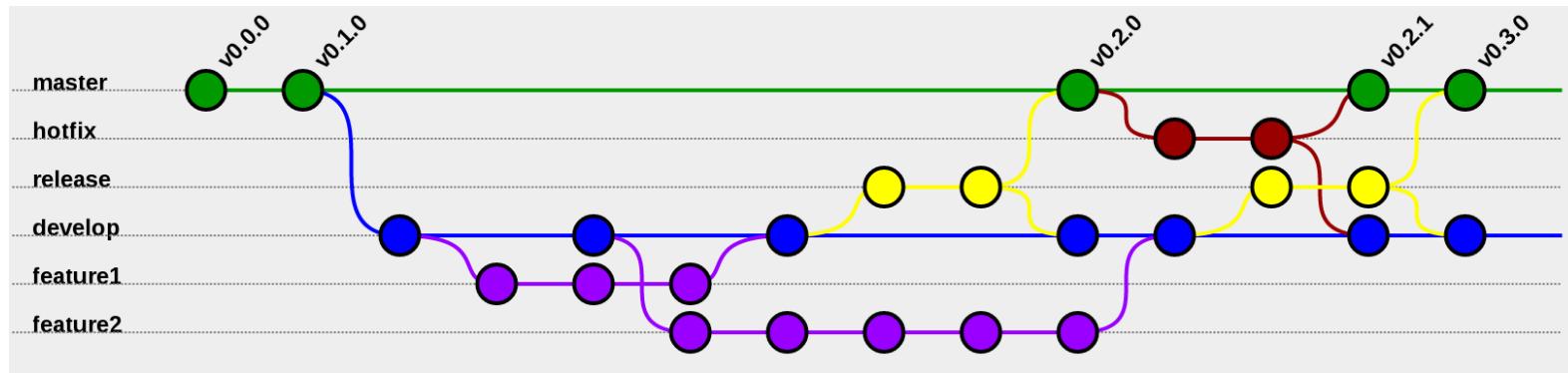


# Commit

Quando si apportano delle modifiche a dei file del progetto è necessario salvarle e darci una descrizione.

Nella terminologia Git, salvare le modifiche equivale a create un **commit** (dall'inglese commettere).

Quindi un commit è un insieme di modifiche apportate ad uno o più file, insieme ad un messaggio testuale che descrive le modifiche fatte. (Il messaggio è obbligatorio).





# "Clonare" un repository

Quando si lavora con un repository è necessario "scaricarene" una copia in locale in modo da poterci lavorare.

Fare un "clone" di un repository equivale a scaricare una copia locale dello stesso, mantenendo un collegamento alla "sorgente" dalla quale è stato scaricato.

Il comando da eseguire è:

```
git clone url_del_repository
```

Git creerà una cartella chiamata con lo stesso nome del repository in cui scaricherà tutto il suo contenuto (e tutte le revisioni dei file).



# Aggiungere un file al repository

Per aggiungere un file al repository:

```
git add nome_del_file
```

Il file è semplicemente stato aggiunto, ma non è ancora parte di un commit, quindi le sue modifiche non sono tracciate.

Aggiungere un file significa dire a Git che quel file verrà inserito all'interno di un commit a breve.



# Vedere lo stato del repository

Per vedere lo stato del repository e quindi conoscere quali file sono stati modificati, aggiunti, cancellati, etc, ma non committati:

```
git status
```

Git ci mostra lo stato corrente e quindi solo quello che è stato modificato dal commit precedente (le modifiche precedenti).



# Creare il nostro primo commit

Una volta aggiunti i file modificati è necessario salvare le modifiche creando un commit ("save point").

```
git commit -m "Messaggio descrittivo"
```

Il messaggio deve essere una descrizione, seppur concisa, delle modifiche contenute nel commit (le modifiche fatte ai file che inseriamo nel commit).

Sono PROIBITI messaggi inutili o "tanto perché è obbligatorio il messaggio"



# Caricare il commit su Github

Una volta che abbiamo creato il nostro commit dobbiamo caricarlo online, ad esempio su Github in modo che le altre persone possano scaricarlo.

```
git push
```

Se non abbiamo commit da caricare Git ci dice che non ha apportato modifiche al repository online.



# Git step by step

Primo passo obbligatorio è clonare il repository, altrimenti non abbiamo il codice su cui lavorare!

Successivamente questi step si ripetono a ciclo:

1. Modificare, creare, eliminare file (quindi fare il lavoro sul codice)
2. Aggiungere a Git i file modificati da includere nel commit
3. Creare il commit insieme al messaggio di descrizione
4. Fare il push del commit
5. Tornare al punto (1) e ricominciare il ciclo

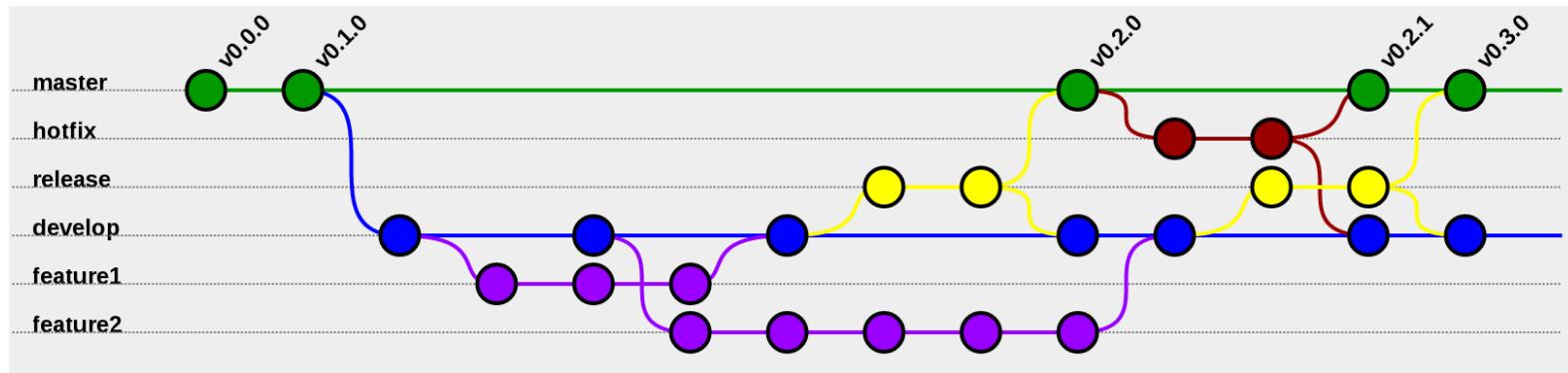


# Branch

Dall'inglese significa ramificazione.

Durante lo sviluppo è possibile che sia necessario intraprendere diverse strade: pensate all'implementazione di una nuova funzionalità del nostro software.

In Git un branch è una ramificazione dello sviluppo, una strada alternativa di sviluppo.





# Branch

Nei repository esiste sempre un branch "speciale" chiamato **master** che rappresenta la linea di sviluppo principale.

Tutte i branch (ramificazioni del codice) possono successivamente essere inclusi nel branch master (quello principale), abbandonati e quindi eliminati dal repository, etc.

Nei casi più semplici si usa solamente il branch master.



# Come creare un branch

Git ci mette a disposizione un semplice comando per creare un nuovo branch:

```
git checkout -b nome_del_nuovo_branch
```

Un branch è una ramificazione del codice e quindi all'interno di questo nuovo "ramo" troviamo tutte le modifiche (commit) effettuati fino al momento in cui è stato creato il branch.

Successivamente i branch seguiranno "strade" differenti e quindi avranno commit differenti (avranno una storia divergente).



# Perchè usare diversi branch?

Usare branch differenti ha diversi vantaggi anche se introduce alcune complessità.

E' comune contenere nel branch **master** il codice attualmente in **produzione**, mentre spostare su un branch differente (a volte chiamato **develop**) lo sviluppo normale. Quando il codice contenuto nel secondo branch viene ritenuto abbastanza maturo da essere messo in produzione, i due branch vengono uniti (**merge**).

Questo permette, in caso di bug in produzione (gli unici che hanno priorità assoluta), di poter essere risolti conoscendo esattamente quale codice è installato nel server e che ha causato il problema.



# Perchè usare diversi branch?

Un'altra "metodologia" molto comune è utilizzare un branch differente per ogni membro del team.

Questo permette che le modifiche (magari non complete) compiute da un membro non "intralcino" il lavoro svolto dal resto del team. Solo quando le modifiche sono complete (e testate!) possono essere unite al codice in master.

Il tutto funziona correttamente (e senza problemi) se i vari membri del team lavorano su funzionalità che "non si intersecano" tra di loro.

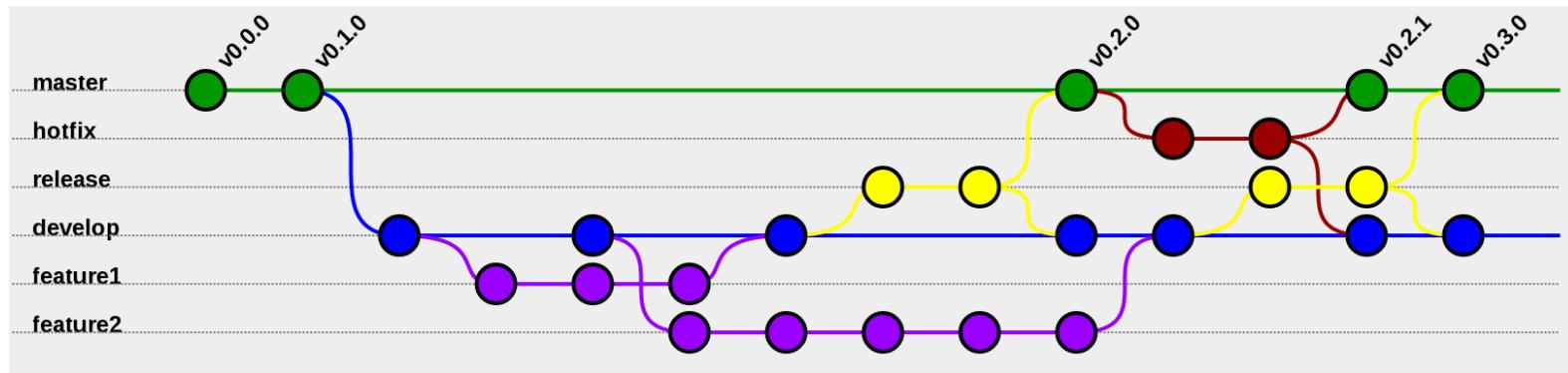


# Perchè usare diversi branch?

I branch sono ramificazioni del codice e le loro storie divergono (commit diversi) e quindi le modifiche effettuate su un branch non incidono sugli altri branch.

Questo permette di poter eseguire delle "prove" di sviluppo di un branch separato.

Se queste modifiche vengono alla fine ritenute "buone" allora possono essere unite al branch principale, altrimenti eliminando il branch vengono eliminate anche le modifiche ripristinando il codice originale.





# Come vedere tutti i branch

Per vedere quali branch sono presenti in un repository è possibile eseguire il comando:

```
git branch
```

C'è un però: questo comando mostra solamente i branch "locali", ovvero i branch presenti sul pc. Per vedere tutti i branch, compresi quelli online:

```
git branch -a
```



# Come muoversi tra branch

Quando siamo su un branch possiamo entrare in un altro branch semplicemente eseguendo il comando

```
git checkout nome_del_branch
```

E' quasi lo stesso comando che eseguiamo per creare un nuovo branch: l'unica differenza è l'assenza dell'attributo -b



# Come eliminare un branch

Per eliminare un branch locale è possibile eseguire il comando:

```
git branch -d nome_del_branch
```

Questo comando viene eseguito da git solamente se il branch è stato unito (merge) in un altro branch. Per forzare git e obbligarlo ad eliminare un branch in ogni caso (USARE CON CAUZIONE):

```
git branch -D nome_del_branch
```



# Come eliminare un branch

Il comando

```
git branch -d nome_del_branch
```

elimina solamente il branch locale ma non elimina il branch online (su Github per esempio). Per eliminare il branch online dobbiamo usare una sintassi leggermente differente:

```
git push --delete origin nome_del_branch
```



# Come eliminare un branch

Eliminare un branch è un'operazione **irreversibile**: una volta eliminato non c'è possibilità di recuperarlo.

Se il branch viene eliminato solamente localmente è possibile scaricarlo dal repository online (Github) ma se viene eliminato anche online non c'è possibilità di recupero.

Prima di eliminare un branch verificate bene quello che state facendo!



# Come caricare un branch online

Quando creiamo un branch in locale esso non viene automaticamente caricato online (Github). Per caricare un branch online dobbiamo eseguire:

```
git push --set-upstream-to origin/nome_del_branch
```



# Come caricare un branch online

Dobbiamo ricordarci questo comando "complicato"? No git ci aiuta e ci suggerisce il comando corretto se eseguiamo solamente:

```
git push
```

Successivamente possiamo caricare online i nostri commit come al solito eseguendo solamente il comando:

```
git push
```



# Come unire due branch

Quando abbiamo completato il lavoro su un branch dobbiamo unire le modifiche nel branch principale (di solito master)

Nella terminologia di git questa operazione si chiama **merge**.

Eseguire un merge è un'operazione che può portare ad un conflitto: stessi file e stesse righe modificate nel branch destinazione (master) e nel branch da unire (il nostro branch).



# Come unire due branch

Prima di unire due branch dobbiamo spostarci sul branch di destinazione del merge (quello in cui vogliamo unire le modifiche dell'altro branch):

```
git checkout nome_del_branch_destinazione
```

Successivamente possiamo eseguire il merge dell'altro branch dentro quello in cui siamo in questo momento (quello destinazione):

```
git merge nome_del_branch_con_le_modifiche
```



# Se si presentano conflitti

Se nell'operazione di merge si presenta un conflitto git ci avverte automaticamente e ci indica quali file sono "andati in conflitto".

```
[MacBook-Pro-di-Alessandro-2:cartella senza titolo alessandro$ git merge test
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
MacBook-Pro-di-Alessandro-2:cartella senza titolo alessandro$ ]
```

```
<<<<< HEAD
modifica sul branch master
=====
modifica effettuata sul branch 'test'
>>>>> test
```

Il contenuto del file in conflitto viene modificato come quello in parte in modo da mostrargli le righe che hanno generato il conflitto.



# Come risolvere un conflitto

```
<<<<< HEAD  
modifica sul branch master  
=====  
modifica effettuata sul branch 'test'  
>>>>> test
```

Il contenuto del file in conflitto viene modificato come quello in parte in modo da mostrarcene le righe che hanno generato il conflitto.

Per risolvere un conflitto è sufficiente decidere quale riga tenere eliminando la riga (e le righe <<<< e >>>> aggiunte da git), aggiungere il file e creare il commit normalmente.

VSCode ci aiuta a risolvere i conflitti in modo "visuale".



# Aggiornare un repository

In uno stesso repository è possibile che lavorino un team di persone.

Ogni persona esegue le sue modifiche e le carica nel repository online.

Ogni tanto è necessario scaricare le modifiche effettuate dagli altri membri e visionare il loro lavoro.

`git pull`

Questo comando scarica i commit presenti nel repository online ma non nel nostro repository locale.



# Thanks!

## Any questions?

Per qualsiasi domanda contattatemi:  
[alessandro.pasqualini.1105@gmail.com](mailto:alessandro.pasqualini.1105@gmail.com)