

# Arrow function

Le nuove funzioni

Con ES6 sono state introdotte delle nuove funzioni dette **Arrow freccia**.

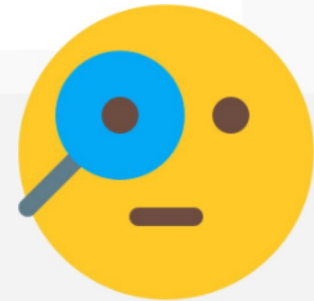
**Come si scrive una arrow function?**

- **Parentesi tonde:**  
Le parentesi tonde sono necessarie, anche se non abbiamo parametri
- **Simbolo freccia =>**
- **Codice da eseguire**  
Dopo la freccia inseriamo il nostro codice

```
() => // blocco di codice su una riga
```

```
() => {  
  // blocco di codice  
  // su più righe  
}
```

Le parentesi graffe sono obbligatorie se scriviamo codice su più righe.



# Arrow function

Le nuove funzioni

---

Come usiamo una Arrow function?

Come le funzioni “normali”,  
le **funzioni a freccia** possono avere  
un nome oppure essere anonime.

```
//funzione con nome
const myFunction = () => 1 + 1;
const resultFunction = myFunction();

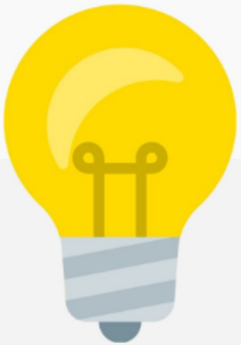
//funzione anonima
document.getElementById('button').addEventListener('click',
    () => console.log(this)
);
```

# Arrow function

Particolarità

La Arrow function non si distingue solo per la sintassi abbreviata:

- 1 Non ha un proprio **this**, ma lo eredita dal suo scope genitore.
- 2 Il **return** è implicito - *quando non usiamo le parentesi graffe.*



Queste caratteristiche come si riflettono sul nostro codice?

# Arrow function

Particolarità

## This ereditato

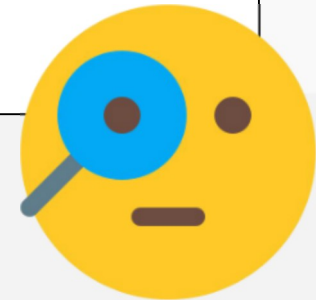
Sappiamo che usando una function anonima, possiamo intercettare l'elemento del DOM che ha scatenato l'evento, utilizzando il **this**.

Con la Arrow Function invece il nostro **this** rappresenta lo scope nella quale è stata invocata.

```
document.getElementById('button').addEventListener('click', function() {  
  console.log(this);  
  // qui otteniamo <button id="button">Click</button>  
});
```

```
document.getElementById('button').addEventListener('click', () =>  
  console.log(this));  
// qui otteniamo Window {}
```

Scopriremo tra un po' che questa caratteristica può tornarci utile.



# Arrow function

Particolarità

## Return implicito senza graffe

Sappiamo che una funzione restituisce sempre **undefined** se non esplicitiamo il return.

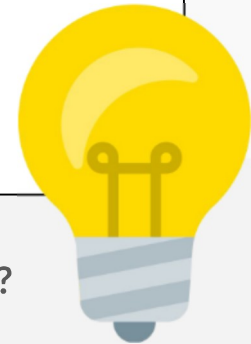
Nella Arrow Function **su una sola linea senza graffe** il return del valore è implicito.

```
const myFunction = () => 1 + 1; ← qui abbiamo il return implicito  
console.log(myFunction()); // otteniamo 2
```

```
const myFunction2 = () => {1 + 1}; ← qui non abbiamo un valore di return  
console.log(myFunction2()); // otteniamo undefined
```

```
const myFunction3 = () => 'pippo';  
console.log(myFunction3());
```

Qui che risultato ci aspettiamo?

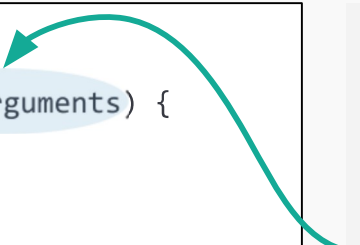


# Rest e Spread

A volte può essere utile conservare il **resto**, la parte rimanente, di una serie di elementi. In questo esempio abbiamo una funzione che potrebbe ricevere un numero variabile di argomenti.

- **...**  
tre punti, operatore **Spread**
- **Nome parametro**  
Inseriamo un nostro nome per l'**array** che conterrà gli elementi

```
function myFunction(...myArguments) {  
  console.log(myArguments);  
}  
myFunction('uno', 3, {name: 'pippo'});  
//avremo un array composto da tre elementi
```



L'unione dell'**operatore Spread** e il **nome del parametro**, prendono quindi il nome di **parametro Rest**.

# L'operatore Spread

Copiamo e uniamo!

```
const arr1 = [3, 5, 1];
const arr2 = [8, 9, 15];

let mergedArray = [...arr1, ...arr2];
console.log(mergedArray); //[3,5,1,8,9,15]

const obj = {name: 'palla', peso: 50};
const copyObj = {...obj, colore: 'blue'};
console.log(copyObj); //{name: "palla", peso: 50, colore: "blue"}
```

L'operatore Spread può essere usato anche per copiare e/o unire parti di array e objects



E in ES5 come si concatenano gli array?

# Arrow function

Le nuove funzioni

Con ES6 sono state introdotte delle nuove funzioni dette **Arrow freccia**.

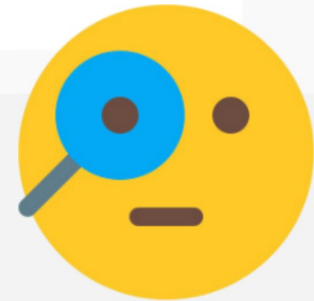
**Come si scrive una arrow function?**

- **Parentesi tonde:**  
Le parentesi tonde sono necessarie, anche se non abbiamo parametri
- **Simbolo freccia =>**
- **Codice da eseguire**  
Dopo la freccia inseriamo il nostro codice

```
() => // blocco di codice su una riga
```

```
() => {  
  // blocco di codice  
  // su più righe  
}
```

Le parentesi graffe sono obbligatorie se scriviamo codice su più righe.





# Arrow function

Le nuove funzioni

---

Come usiamo una Arrow function?

Come le funzioni “normali”,  
le **funzioni a freccia** possono avere  
un nome oppure essere anonime.

```
//funzione con nome
const myFunction = () => 1 + 1;
const resultFunction = myFunction();

//funzione anonima
document.getElementById('button').addEventListener('click',
    () => console.log(this)
);
```

# Arrow function

Particolarità

La Arrow function non si distingue solo per la sintassi abbreviata:

- 1 Non ha un proprio **this**, ma lo eredita dal suo scope genitore.
- 2 Il **return** è implicito - *quando non usiamo le parentesi graffe.*



Queste caratteristiche come si riflettono sul nostro codice?

# Arrow function

Particolarità

## This ereditato

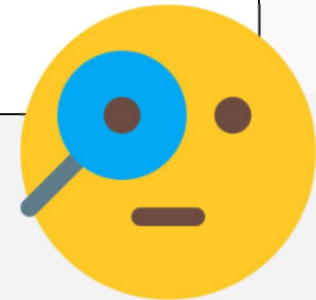
Sappiamo che usando una function anonima, possiamo intercettare l'elemento del DOM che ha scatenato l'evento, utilizzando il **this**.

Con la Arrow Function invece il nostro **this** rappresenta lo scope nella quale è stata invocata.

```
document.getElementById('button').addEventListener('click', function() {  
  console.log(this);  
  // qui otteniamo <button id="button">Click</button>  
});
```

```
document.getElementById('button').addEventListener('click', () =>  
  console.log(this));  
// qui otteniamo Window {}
```

Scopriremo tra un po' che questa caratteristica può tornarci utile.



# Arrow function

Particolarità

## Return implicito senza graffe

Sappiamo che una funzione restituisce sempre **undefined** se non esplicitiamo il return.

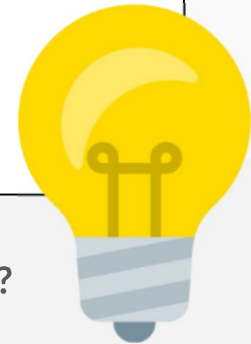
Nella Arrow Function **su una sola linea senza graffe** il return del valore è implicito.

```
const myFunction = () => 1 + 1; ← qui abbiamo il return implicito  
console.log(myFunction()); // otteniamo 2
```

```
const myFunction2 = () => {1 + 1}; ← qui non abbiamo un valore di return  
console.log(myFunction2()); // otteniamo undefined
```

```
const myFunction3 = () => 'pippo';  
console.log(myFunction3());
```

Qui che risultato ci aspettiamo?

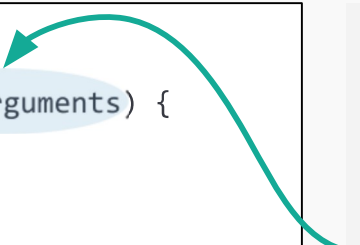


# Rest e Spread

A volte può essere utile conservare il **resto**, la parte rimanente, di una serie di elementi. In questo esempio abbiamo una funzione che potrebbe ricevere un numero variabile di argomenti.

- **...**  
tre punti, operatore **Spread**
- **Nome parametro**  
Inseriamo un nostro nome per l'**array** che conterrà gli elementi

```
function myFunction(...myArguments) {  
  console.log(myArguments);  
}  
myFunction('uno', 3, {name: 'pippo'});  
//avremo un array composto da tre elementi
```



L'unione dell'**operatore Spread** e il **nome del parametro**, prendono quindi il nome di **parametro Rest**.

# L'operatore Spread

Copiamo e uniamo!

```
const arr1 = [3, 5, 1];
const arr2 = [8, 9, 15];

let mergedArray = [...arr1, ...arr2];
console.log(mergedArray); //[3,5,1,8,9,15]

const obj = {name: 'palla', peso: 50};
const copyObj = {...obj, colore: 'blue'};
console.log(copyObj); //{name: "palla", peso: 50, colore: "blue"}
```

L'operatore Spread può essere usato anche per copiare e/o unire parti di array e objects



E in ES5 come si concatenano gli array?

# Ciclo forEach

Un nuovo vecchio ciclo

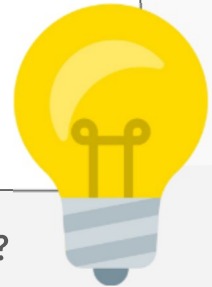
ForEach ci aiuta a ciclare sugli array in maniera più concisa

- **keyword:**  
forEach
- **function:**  
All'interno delle parentesi tonde inseriamo una funzione che riceverà come argomenti:  
**element** - l'elemento dell'array sul quale stiamo girando  
**index** - l'indice di quell'elemento  
**array** - l'array stesso
- **codice:**  
che verrà eseguito ad ogni giro

```
const students = ['Paolo', 'Giulia', 'Marco'];

students.forEach((element, index, array) => {
  console.log(element, index);
});
// Paolo 0
// Giulia 1
// Marco 2
```

Questa che tipo di funzione è?



# Ciclo forEach

Un nuovo vecchio ciclo

ForEach ci aiuta a ciclare sugli array in maniera più concisa

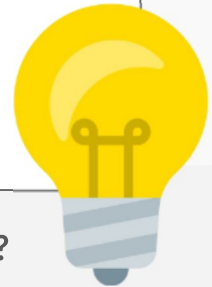
- **keyword:**  
forEach
- **function:**  
All'interno delle parentesi tonde inseriamo una funzione che riceverà come argomenti:  
**element** - l'elemento dell'array sul quale stiamo girando  
**index** - l'indice di quell'elemento  
**array** - l'array stesso
- **codice:**  
che verrà eseguito ad ogni giro

```
const students = ['Paolo', 'Giulia', 'Marco'];

students.forEach((element, index, array) => {
  console.log(element, index);
});

// Paolo 0
// Giulia 1
// Marco 2
```

Questa che tipo di funzione è?



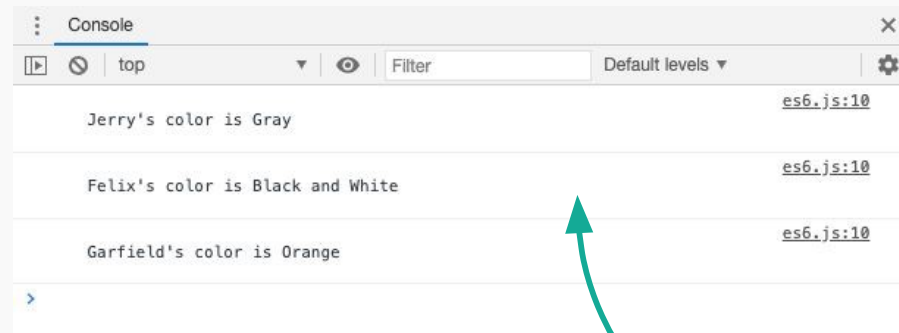


# Ciclo forEach

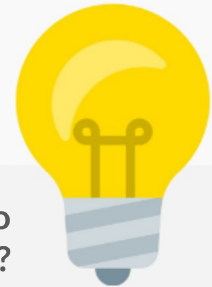
Un nuovo vecchio ciclo

Se voglio ciclare su un array di object?

```
const cats = [  
  {name: 'Jerry', color: 'Gray'},  
  {name: 'Felix', color: 'Black and White'},  
  {name: 'Garfield', color: 'Orange'},  
];  
  
cats.forEach((element) => {  
  console.log(`  
    ${element.name}'s color is ${element.color}  
  `);  
});
```



Come mai abbiamo tutto questo spazio  
tra una riga e l'altra?



# Map e Filter

Alcuni trucchi con gli array

---



Sappiamo già come iterare sugli elementi degli array, filtrare dati, fare operazioni come sommare tutti gli elementi...

Grazie ad alcuni metodi di JS (*implementati già nella ES5 e ora supportati dalla maggior parte dei browser*) possiamo fare alcune di queste operazioni con più semplicità!

# Map

Un ciclo che ci restituisce qualcosa

**Map** assomiglia molto al `forEach...`

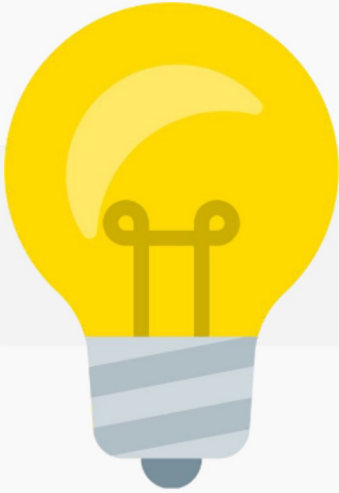
ma se ne differenzia per il fatto che **ci restituisce un nuovo array** che dobbiamo salvare in una variabile.

```
const students = ['Paolo', 'Giulia', 'Marco'];
const newStudents = students.map((element, index, array) => {
  console.log(index);
  if(element == 'Paolo') {
    return 'Paola';
    // qui abbiamo bisogno di usare il return per poter restituire il valore
  }
  return element;
  // ma dobbiamo restituire anche gli altri elementi
});
//0,1,2
console.log(newStudents); // ['Paola', 'Giulia', 'Marco']
```

# Map

Alcuni trucchi con gli array

---



Quindi cosa succede se metto un return in un forEach?

# Filter

E se volessi estrapolare solo alcuni elementi di un array?

---

**Filter** ha una sintassi molto simile a quella di Map ma **restituisce solo i valori che verificano una data condizione**.

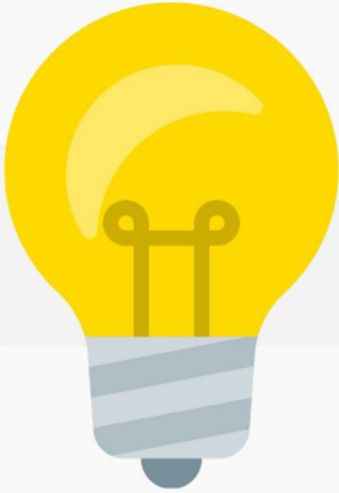
Come per Map dobbiamo restituire un valore quindi abbiamo bisogno di un return.

```
const filteredArray = array.filter((element, index, array) => {  
  return CONDIZIONE  
});
```

# Filter

E se volessi estrapolare solo alcuni elementi di un array?

---



Sappiamo farlo con un ciclo for?

# Filter

E se volessi estrapolare solo alcuni elementi di un array?

Ritornando a Filter possiamo farlo con meno righe, ma il nostro **return** è un po' particolare...

Dobbiamo inserire la **condizione** che mettiamo nel nostro **If** e **Filter** ci restituirà solo gli elementi che verificano la condizione.

```
if (element === 'Paolo') { //Condizione verificata
  return element; //allora restituiamo element
}
```

```
const students = ['Paolo', 'Giulia', 'Marco'];

const filteredStudents = students.filter((element) => {
  return element === 'Paolo'
});

console.log(filteredStudents);
//['Paolo']
```