Oggetti

CREAZIONE DI UN OGGETTO

- Un oggetto può essere creato tramite l'assegnazione di una **ObjectExpression** ad una variabile.

```
let obj = { value: 42 };
```

> NOTA
> Forse avete incontrato `new Object`. Si tratta di una forma arcaica, da evitare in favore di un "object literal": `{ ... }`.

OGGETTI

- le proprietà di un oggetto vengono accedute grazie alla sintassi `<EXPR>.<IDENT>`:

```
const obj = { value: 42 };
console.log(obj.value); // 42
```

- si può accedere alle proprietà calcolandone dinamicamente il nome usando la sintassi `<EXPR>[<EXPR>]`:

```
const obj = { value: 42 };
const propertyName = "value";
console.log(obj[propertyName]); // 42
```

OGGETTI

è possibile assegnare funzioni alle property di un oggetto

```
let obj = {
  aMethod: function {
    // Corpo del metodo
  },
  anotherMethod: () => {
    // Corpo del metodo
  },
  otherMethod() {
    // Corpo del metodo
  }
};
```

OGGETTI

- è possibile assegnare ad un oggetto delle proprietà utilizzando lo shorthand quando il nome della proprietà coincide con il nome della variabile da cui avreste estratto il valore:

```
const payload = 1234;

getAjax({ payload });
// ...è uguale a...
getAjax({ payload: payload });
```

- è possibile assegnare ad un oggetto dei parametri i cui nomi sono calcolati dinamicamente:

```
const obj = { ["par" + "am"]: 42 };
obj; // { param: 42 }
```

OGGETTI

Una funzione creata con la keyword `function` può essere usata per istanziare un nuovo oggetto:

```
function myFunction() {
  this.value = 42;
}

let obj = new myFunction();

obj; // Object { value: 42 }
```

> NOTA
>
> **Non si possono usare le Arrow!** Si dice che "non sono istanziabili".
>
> NOTA 2
>
> **Non è una sintassi ed un "problema" che si incontra nello sviluppo React**.

Prototype

PROTOTYPE

- In JavaScript l'ereditarietà è gestita tramite prototype.
- Ogni oggetto ha il proprio prototype (`__proto__` che punta ad un altro oggetto) o a `null`.

```
let obj = {
  value: 42
};

obj; // Object { value: 42 }

obj.__proto__; // Object { }

obj.__proto__.__proto__; // null
```

PROTOTYPE

- Instanziando un oggetto con la reseved keyword `new` creiamo un oggetto con una property `__proto__` che punta al prototype della funzione che ha lo ha generato.
- Ogni `function` ha un property `prototype`.
- ATTENZIONE: Le arrow function non hanno la property `prototype`.

PROTOTYPE

Quando chiamo il campo di un oggetto questo viene cercato nella prototype chain.

```
let MyObject = function() {
  this.aMethod = function() {
    return "aMethod called";
  }
}

let a = new MyObject();
a.aMethod(); // "aMethod called"

MyObject.prototype.anotherMethod = function() {
  return "anotherMethod called";
}
a.anotherMethod(); // "anotherMethod called"
```

PROTOTYPE

Quando chiamo il campo di un oggetto questo viene cercato nella prototype chain.

```
a;
/*
{
  aMethod: [function]
  <prototype>: { // non è property di a ma di MyObject
    anotherMethod: [function]
    constructor: [function]
    <prototype>: {...}
  }
}
*/
```

PROTOTYPE

- Quando ci si riferisce ad una property di un oggetto viene utilizzato il primo riferimento valido nella prototype chain dell'oggetto.

```
let obj = { value: "foo" };
obj.__proto__.value = "bar";
obj.value; // "foo"
```

> ATTENZIONE
>
> La ricerca delle property nella prototype chain può causare problemi di prestazioni. Si sconsiglia utilizzare lunghe prototype chain.

PROTOTYPE

è possibile modificare prototype esistenti sovrascrivendone proprietà e metodi. **Da evitare con tutte le vostre forze. Negli anni questa pratica, molto comune in librerie "di utilità" è diventata la causa di molteplici problemi di incompatibilità.**

Sappiate che è possibile farlo.

```
let obj = { value: 42 };
obj.toString(); // "[object Object]"

Object.prototype.toString = function() { return "TEST"; }
obj.toString(); // "TEST"
```

PROTOTYPE

- Domanda

```
let ArrowObject = () => {};

let b = new ArrowObject();
```

PROTOTYPE

- Domanda

```
let ArrowObject = () => {};

let b = new ArrowObject(); // TypeError: ArrowObject is not a constructor
```

PROTOTYPE

- Domanda

```
let myArrowFunction = () => {
  console.log(this);
};

let myFunction = function() {
  this.aValue = 42;
  console.log(this);
};

myFunction();
myArrowFunction();

let obj = new myFunction();
```

Cosa stampa nei tre casi?

PROTOTYPE

- Domanda

```
let myArrowFunction = () => {
  console.log(this);
};

let myFunction = function() {
  this.aValue = 42;
  console.log(this);
};

myFunction(); // window
myArrowFunction(); // window

let obj = new myFunction(); // Object { aValue: 42 }
```

Cosa stampa nei tre casi?

PROTOTYPE

- DOMANDA: Cosa succede nelle due chiamate?

```
myFunction.__proto__.aMethod = function() {
  console.log("test")
};

obj.aMethod();
myFunction.aMethod();
```

PROTOTYPE

- DOMANDA: Cosa succede in questo caso?

```
myFunction.__proto__.aMethod = function() {
  console.log("test")
};

obj.aMethod(); // TypeError: obj.aMethod is not a function
myFunction.aMethod(); // test
```

PROTOTYPE

- DOMANDA: Cosa succede agli oggetti già istanziati?

```
function myFunction() {}
let obj = new myFunction();
myFunction.prototype.test = 42;
obj.test;
```

PROTOTYPE

- DOMANDA: Cosa succede agli oggetti già istanziati?

```
function myFunction() {}
let obj = new myFunction();
myFunction.prototype.test = 42;
obj.test; // 42
```

essendo `__proto__` un riferimento gli oggetti già istanziati risentiranno delle modifiche ai prototype della loro chain.

PROTOTYPE

- DOMANDA

```
let Person = function() {}
let me = new Person();
```

Quale dei seguenti è equivale?

```
let me = {};
me.__proto__ = Person;
```

```
let me = {};
me.__proto__ = Person.__proto__;
```

```
let me = {};
me.__proto__ = Person.prototype;
```

```
let me = {};
me.prototype = Person.__proto__;
```

PROTOTYPE

Per far puntare il prototype di un oggetto ad un altro ovvero "estendere" un oggetto si può usare `Object.create()`

```
let obj = {
  a: 1
}

let otherObj = Object.create(obj);
otherObj.a; // 1
```

CLASS

- `class` offre zucchero sintattico per la creazione di una funzione che può generare un oggetto.

```
class MyClass {}
let obj = new MyClass();
```

CLASS

Usando `class` al momento dell'istanziazione di un oggetto viene chiamato il metodo `constructor`

```
class MyClass {
  constructor() {
    console.log("constructor called");
  }
}

let MyFunction = function() {
  console.log("construcor called")
};
```

- ATTENZIONE

```
MyFunction();
// contsructor called
```

```
MyClass();
// TypeError: class constructors must be invoked with 'new'
```

CLASS

Con il costrutto `class` si possono dichiarare variabili e metodi

```
class MyClass {
  constructor() {

  }

  varaiable = "test";

  myMethod() {

  }
}
```

CLASS

Con il costrutto `class` si possono dichiarare funzioni statiche

```
class MyClass {
  static aFunction() {
    return "foo";
  }
}

MyClass.aFunction(); // "foo"
let obj = new MyClass();
obj.aFunction(); // TypeError: obj.aFunction is not a function
```

CLASS

Congli oggetti è possibile usare la sintassi di **MethodInvocation**:

```
let a = {
  b: function() {
    console.log(this);
  }
}
a.b();
// {b: [function]}
```

```
let a = {
  b: () => {
    console.log(this);
  }
}
a.b();
// Window {...}
```

CLASS

- Con `delete` è possibile eliminare una proprietà di un oggetto.

```
const obj = { prop: 12, other: 45 };
console.log(Object.keys(obj)); // ['prop', 'other']
delete obj.prop;
console.log(Object.keys(obj)); // ['other']
```

```
delete someGlobalVar; // è come usare window.someGlobalVar!
```

> `delete` elimina la proprietà di un oggetto e restituisce `true` (se è riuscito ad eliminare la proprietà) o `false` (se non è riuscito).