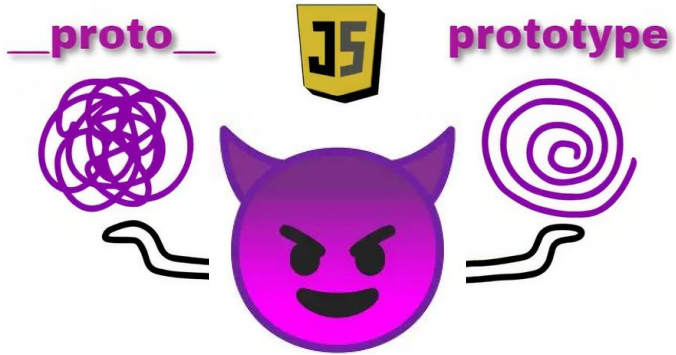
# Laboratorio I
# a.a. 2025/2026

Classi, oggetti, e caratteristiche avanzate

# Contenuti

- Espressioni-classe
- Ereditarietà
  - Superclassi e sottoclassi
  - Overriding
  - `super`
- Dichiarazione di campo
- Membri privati
- Membri statici
- Getter & setter
- Generatori
- Q&A

# Chiarimento sui diabolici __proto__ e prototype




La chiave **prototype** ce l'hanno solo le funzioni, mentre la chiave **__proto__** ce l'hanno tutti gli oggetti (e in particolare anche le funzioni).

Dato un oggetto *o*, il suo prototipo è l'oggetto *o.__proto__* (che può essere *null*, se siamo arrivati al termine della catena di prototipi).

Data una funzione *fun*, essa possiede la chiave **prototype**. Se *fun* costruisce *o* allora risulta vera l'uguaglianza:

*fun*.**prototype** === *o*.**__proto__**

(*fun.__proto__* è la funzione prototipo di tutte le funzioni, che è un oggetto diverso da *o.__proto__*)

# Introduzione

Oggi passeremo in rassegna alcune caratteristiche delle classi in JavaScript che ancora non abbiamo affrontato.

**Non daremo** per ciascuna molti esempio di uso; vedremo però degli esempi nell'esercitazione della prossima lezione, e in generale nelle prossime settimane quando costruiremo insieme dei progetti complessi.

Questa parte del linguaggio è comunque **in rapida evoluzione**; è possibile che fra un anno, le cose saranno (leggermente) diverse.

# Traduzione delle dichiarazioni

## Javascript

```
class Persona {
  constructor(nome,età) {
    this.nome=nome
    this.età=età
  }

  compleanno() {
    this.età++
  }
}


var pippo = new Persona(“Pippo”,35)
```

## Pseudocodice

```
function Persona(this,nome,età) {
  this.nome=nome
  this.età=età
  this.__proto__=Persona.prototype
}

Persona.prototype = {
  constructor: Persona,
  compleanno: function(this) {
    this.età++
  }
}

let t={}; Persona(t,”Pippo”,35)
var pippo = t
```

# Espressioni-classe

Finora abbiamo visto le classi come **dichiarazioni**:

```
class Persona{
  constructor(n,e) {
    this.nome=n
    this.eta=e
  }
  compleanno() {
    this.eta++
  }
}
var pippo=new Persona("Pippo",35)
```

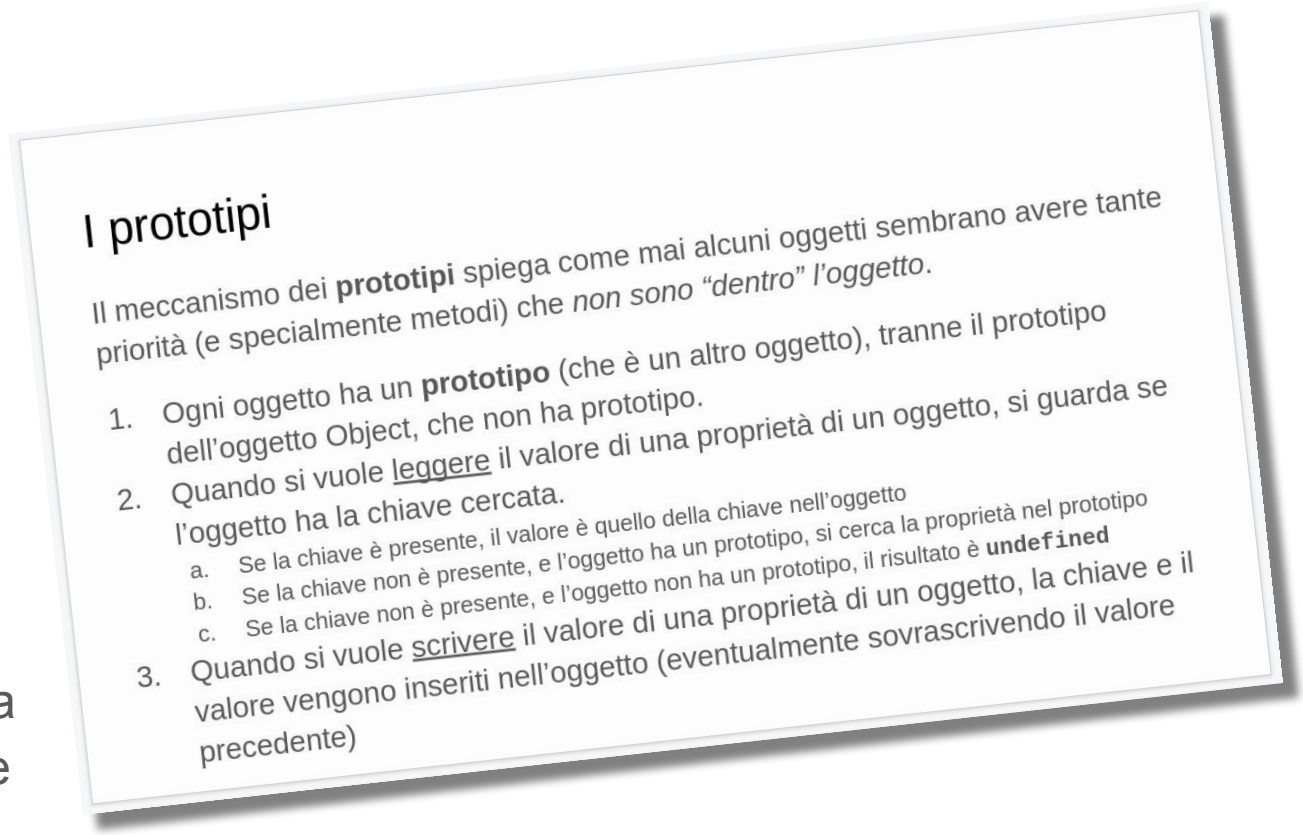
Esiste però anche la variante **espressione** (come abbiamo visto anche per function):

```
var Persona = class{
  constructor(n,e) {
    this.nome=n
    this.eta=e
  }
  compleanno() {
    this.eta++
  }
}
var pippo=new Persona("Pippo",35)
```

# Ereditarietà

Abbiamo visto come per ogni oggetto sia definita la sua **catena dei prototipi**.

Quando si accede in lettura a una proprietà, se la chiave non è definita nell'oggetto, si va a cercare nel prototipo (e via così, ricorsivamente).



## I prototipi

Il meccanismo dei **prototipi** spiega come mai alcuni oggetti sembrano avere tante priorità (e specialmente metodi) che *non sono "dentro" l'oggetto*.

1. Ogni oggetto ha un **prototipo** (che è un altro oggetto), tranne il prototipo dell'oggetto Object, che non ha prototipo.
2. Quando si vuole leggere il valore di una proprietà di un oggetto, si guarda se l'oggetto ha la chiave cercata.
   a. Se la chiave è presente, il valore è quello della chiave nell'oggetto
   b. Se la chiave non è presente, e l'oggetto ha un prototipo, si cerca la proprietà nel prototipo
   c. Se la chiave non è presente, e l'oggetto non ha un prototipo, il risultato è **`undefined`**
3. Quando si vuole scrivere il valore di una proprietà di un oggetto, la chiave e il valore vengono inseriti nell'oggetto (eventualmente sovrascrivendo il valore precedente)

# Ereditarietà

È anche possibile **impostare** il prototipo di una nuova classe ad un'altra classe.

Grazie al meccanismo dei prototipi, gli oggetti della nuova classe avranno anche tutti i metodi definiti dalla classe che viene **estesa**.

In più, la sottoclasse può aggiungere ulteriori metodi.

Solo le **istanze** della **sottoclasse** avranno i metodi aggiunti.


```
class Studente extends Persona {
  laurea() {
    return "Evviva!"
  }
}

var ugo = new Studente("Ugo",19)

ugo → Studente { nome: 'Ugo', 'età': 19 }
ugo.compleanno() →
ugo → Studente { nome: 'Ugo', 'età': 20 }
ugo.laurea() → Evviva!

pippo.laurea() → TypeError: pippo.laurea is
not a function
```

# Ereditarietà

In Inglese si chiama ***overriding***

Applicando il meccanismo di ricerca nella catena dei prototipi, osserviamo che:

- Una sottoclasse può **ridefinire** un metodo già definito in una sua superclasse (diretta o indiretta)
- Una sottoclasse eredita tutti i metodi della superclasse che non ridefinisce
- Usando l'operatore **delete** (sul prototipo) è anche possibile cancellare dalla sottoclasse un metodo definito dalla superclasse senza sovrascriverlo
- Un'istanza può cancellare o sovrascrivere un metodo della propria classe

```
class StMagistrale extends Studente {
  laurea() {
    return "Stra-evviva galattico!"
  }
}

var pia = new StMagistrale("Pia",23)

pia.laurea() → Stra-evviva galattico!
ugo.laurea() → Evviva!
pippo.laurea() → TypeError: pippo.laurea is not
a function
```

Tutto questo vale anche per constructor()

# Ereditarietà

Capita spesso che il codice in una sottoclasse debba fare riferimento ai metodi della sua superclasse (per esempio, per completarli o aggiungere caratteristiche).

La parola chiave **`super`** è un riferimento alla superclasse di un oggetto, come **`this`** è un riferimento all'oggetto su cui il metodo è invocato.

Ricordate, una classe è in realtà una funzione, quindi **`super`** è una funzione

```
class StMagistrale extends Studente {
  constructor(nome, eta, triennale) {
    super(nome,eta)
    this.triennale=triennale
  }
  laurea() {
    return super.laurea()+" Evviva galattico!"
  }
}

var pia=new StMagistrale("Pia",24,"Informatica")

pia → StMagistrale { nome: 'Pia', 'età': 24,
triennale: 'Informatica' }
pia.laurea() → 'Evviva! Evviva galattico!'
```

# Implementazione dell'ereditarietà

```
class Persona { … }

class Studente extends Persona {
  laurea() {
    return "Evviva!"
  }
}
```

Di fatto, l'ereditarietà è implementata impostando la catena dei prototipi in modo che il prototipo di **Studente** abbia come *suo* prototipo quello di **Persona**.

Come sempre, i prototipi possono anche cambiare dinamicamente (durante l'esecuzione!)

```
Studente.prototype = {
  constructor: Studente,
  laurea: function(this) {
    return "Evviva!"
  },
  __proto__: Persona.prototype
}
```

Questo serve per ereditare i **metodi di istanza** (catena dei prototipi delle istanze)

…

```
Studente.__proto__ === Persona → True
```

Questo serve per ereditare i **metodi statici** delle classi (catena dei prototipi delle funzioni costruttrici)

# La parola chiave **super**

```
class Persona{
  constructor(nome,eta) {...}
  compleanno() {this.eta++}
}

class Studente extends Persona{
  compleanno(){
    super.compleanno()
    console.log("auguri studente")
  }
}

var ugo = new Studente("Ugo",19)
```

Parola chiave js che (informalmente) significa: "**parti dal prototipo del padre, non da quello corrente**"

Che succede quando scrivo questo?

```
ugo.compleanno()
```

# La parola chiave **super**

```
class Persona{
  constructor(nome,eta) {...}
  compleanno() {this.eta++}
}

class Studente extends Persona{
  compleanno(){
    super.compleanno()
    console.log("auguri studente")
  }
}

var ugo = new Studente("Ugo",19)
```

Parola chiave js che (informalmente) significa: "**parti dal prototipo del padre, non da quello corrente**"

Che succede quando scrivo questo?

```
ugo.compleanno()
```

1. `super.compleanno()` significa: prendi il metodo compleanno dal prototipo del padre di ugo e chiamalo con this=ugo
2. Incremento l'età di ugo e poi stampo "auguri studente"

# **super()** dentro constructor

super() è obbligatorio se volete definire il constructor di una classe extends.

super() dentro constructor di Studente fa queste cose:

- chiama il costruttore della superclasse (Persona)
- inizializza le proprietà definite nella superclasse sull'oggetto istanza *this*

Tutto questo permette di usare metodi della superclasse dentro il constructor di Studente usando *this*, per esempio *this*.compleanno()

Come viene creata l'istanza pia di StMagistrale?

```
pia = new StMagistrale("Pia",24,"Informatica")
```

1. Il *new* crea {} e associa *this*={}, attaccandolo alla catena dei prototipi grazie a *extends*.
2. Chiamo il constructor di StMagistrale.
3. Dentro il constructor di StMagistrale trovo **super**(), quindi chiamo il constructor di Studente, e così via fino ad arrivare a chiamare il constructor di Persona.
4. Di fatto, eseguo i constructor in questo ordine: Persona->Studente->StMagistrale. Ogni constructor aggiunge proprietà proprie all'oggetto istanza *this*.
5. Il *new* ritorna l'oggetto *this* popolato e allacciato alla catena dei prototipi, che viene associato alla variabile pia.

# Dichiarazioni di campo

JavaScript è un linguaggio ancora in rapida evoluzione.

Una proposta (entrata nello standard ECMAScript 2022 del linguaggio) prevede che si possano dichiarare le proprietà di un oggetto direttamente nella classe, anziché con assegnamenti a this.*proprietà* nel costruttore

```
class Persona {
  nome="anonimo"
  età=-1
  /* eccetera */
}
```

Se non viene fornito un inizializzatore di default, il valore iniziale sarà *undefined*, per cui in pratica serve comunque inizializzarle nel constructor().

Ci sono comunque casi in cui le dichiarazioni di campo sono utili (per esempio, quando il valore di inizializzazione è calcolato chiamando un metodo, anche della superclasse).

# Membri privati

JavaScript è un linguaggio ancora in rapida evoluzione.

Una proposta (entrata nello standard ECMAScript 2022 del linguaggio) prevede che i campi che iniziano per **#** siano visibili **solo all'interno della classe**.

In particolare, non partecipano alla catena dei prototipi, non sono ereditabili

```
class C {
  #privato() { /* codice */ }
  pubblico() { /* qui si può usare this.#privato() */ }
}

var c= new C()
c.pubblico()
c.#privato()
```

Ogni accesso a una proprietà il cui nome inizia con **#** fuori dalla definizione della classe produce un errore

# Membri statici

Una classe può includere delle proprietà **statiche**, introdotte dalla parola chiave **static**.

Queste proprietà non saranno proprietà delle singole istanze, ma della classe stessa.

```
class Prof extends Persona {
  static boccia() { /* codice */ }
}

var andrea = new Prof()
Prof.boccia()   ✅
andrea.boccia()   💀
```

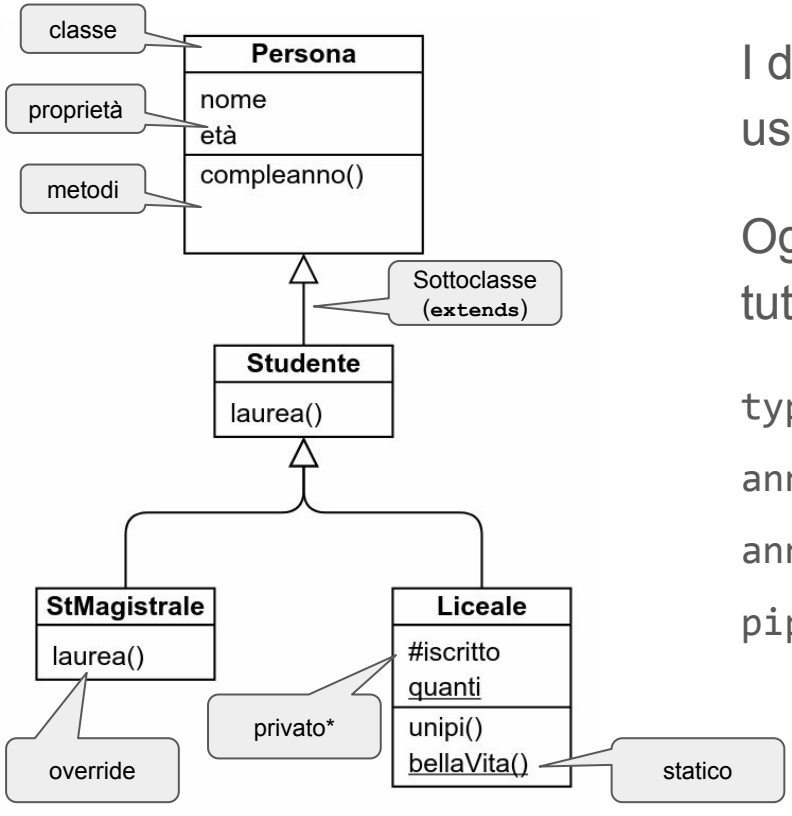
A differenza di altri linguaggi, non si può accedere a un membro statico tramite un'istanza -- occorre proprio indicare il nome della classe.

# Un esempio

```
class Liceale extends Studente {
  static quanti=0
  #iscritto=false
  constructor(...args) {
    super(...args)
    Liceale.quanti++
  }
  unipi() {return this.#iscritto}
  static bellaVita() {return true}
}

var luca=new Liceale("Luca", 15)
var anna=new Liceale("Anna", 16)
```

```
Liceale.quanti → 2

luca.unipi() → false

luca.#iscritto → SyntaxError: Private field
'#iscritto' must be declared in an enclosing
class

luca.quanti → undefined


luca → Liceale { nome: 'Luca', 'età': 15 }

Liceale → [class Liceale extends Studente] {
quanti: 2 }

Liceale.bellaVita() → true

anna.bellaVita() → TypeError: anna.bellaVita
is not a function
```

# Ereditarietà e tipi




I diagrammi delle classi si disegnano spesso usando il **linguaggio di modellazione UML**.

Ogni istanza di una classe è anche istanza di tutte le sue superclassi (ricorsivamente).

```
typeof anna → "object"
anna instanceof Liceale → true
anna instanceof Persona → true
pippo instanceof Studente → false
```

anna è una Liceale, pippo è una Persona

* In UML, + indica public, - private, # protected, ~ package. Qui seguiamo la notazione JavaScript anziché quella UML.

# Altre caratteristiche

JavaScript include alcune caratteristiche che sono di uso generale, ma che vengono usate con particolare efficacia nella definizione di classi.

Le prossime due caratteristiche (*getter* & *setter* e generatori) possono essere usate anche fuori da una classe, per esempio con

```
let oggetto= { get x() {return 1}, set x() {;} }
```

```
function* range(a,b) {var i=a; while (i<b) yield i++}
```

Curiosi? Bene, vediamo di cosa si tratta…

# Getter & setter (metodi di accesso o *accessors*)

I metodi di accesso consentono di “simulare” la presenza di una proprietà in un oggetto; però le letture o le scritture di quella proprietà causano l’esecuzione di un metodo, anziché una lettura o scrittura nel dizionario dell’oggetto.

Le parole chiave **get** e **set**, davanti al nome della proprietà, creano i metodi di accesso. Dall’esterno, la proprietà appare come una normale chiave con valore.

```
let o = { get x() {return 1}, set x(v) {;} }
```

```
o.x → 1        o.x=5 → 5      o.x → 1
```

Ogni lettura di proprietà causa l’invocazione del getter. Ogni scrittura di proprietà causa l’invocazione del setter (passando come argomento il valore assegnato).

# Getter & setter nelle classi

I metodi di accesso possono essere usati, per esempio, per consentire l'accesso allo stesso dato in modi diversi.

```
class Distanza {
  static #MIGLIO=1.60934    // fattore di conversione
  #distanza=0   // in km
  get km() { return this.#distanza}
  set km(v) { this.#distanza=v}
  get miglia() { return this.#distanza/Distanza.#MIGLIO}
  set miglia(v) {this.#distanza=v*Distanza.#MIGLIO}
}

var d=new Distanza()
```

```
d → Distanza {}
d.km → 0
d.km=5 → 5
d.km → 5
d.miglia →
3.106863683249034
d.miglia=3 → 3
d.km → 4.82802
```

# Precisiamo la strategia di ricerca di una proprietà

1. Quando si vuole leggere il valore di una proprietà di un oggetto, si guarda se l'oggetto ha la chiave cercata.
    a. Se la chiave è presente, si controlla se è un getter o una proprietà base
        i. Se è un getter, si invoca la funzione corrispondente, e il valore è quello restituito dalla funzione
        ii. Se è una proprietà base, il valore è quello della chiave nell'oggetto
    b. Se la chiave non è presente, e l'oggetto ha un prototipo, si cerca la proprietà nel prototipo, ricorsivamente
    c. Se la chiave non è presente, e l'oggetto non ha un prototipo, il risultato è `undefined`
2. Quando si vuole scrivere il valore di una proprietà di un oggetto, si guarda se l'oggetto ha la chiave cercata.
    a. Se la chiave è presente, si controlla se è un setter o una proprietà base
        i. Se è un setter, si invoca la funzione corrispondente, passando come unico argomento il valore che si vuole assegnare
        ii. Se è una proprietà base, si assegna il valore alla proprietà (eventualmente sovrascrivendo il valore precedente)
    b. Se la chiave non è presente, si cerca ricorsivamente un setter nel prototipo, ricorsivamente
        i. Se si trova un setter lungo la catena, si invoca la funzione corrispondente, passando come unico argomento il valore che si vuole assegnare
        ii. Se non si trova un setter lungo la catena, la proprietà viene aggiunta all'oggetto, con il valore che si vuole assegnare.

Come dire: getter e setter vengono ereditati

# Generatori

In alcuni casi si vorrebbe poter tornare da una invocazione di funzione restituendo il controllo al chiamante, ma poi riprendere la computazione da dove si era rimasti.

I generatori sono un particolare tipo di funzione (e dunque di metodo) che ha proprio questa caratteristica.

I generatori si dichiarano con

- `function* f() {}` anziché `function f() {}`, oppure
- `*metodo() {}` anziché `metodo() {}` (dentro le classi).

Nel corpo di un generatore (e solo lì) si può usare il comando `yield expr`, che restituisce al chiamante il valore di ***expr***, ma riprende l'esecuzione dal comando successivo (e non dall'inizio del corpo) in caso di "rientro"

# Generatori

```
function* range(a,b) { var i=a; while (i<b) yield i++ }
```

La funzione `range(a,b)` restituisce un **generatore** che, quando lo scorreremo, ci darà ogni volta un valore compreso fra `a` (incluso) e `b` (escluso), ricordandosi di dove eravamo arrivati.

Per scorrere un generatore, si chiama ripetutamente il suo metodo **`next()`**.

Il generatore è un oggetto con due proprietà: **`value`** è il valore restituito, e **`done`** vale true se la generazione è completa.

Nella pratica, spesso lo scorrimento è fatto con un **`for (... of ...)`**, e non si invoca esplicitamente **`next()`**.

# Generatori

```
function* range(a,b) { var i=a; while (i<b) yield i++ }
```

```
var x=range(4,8)

x → Object [Generator] {}
x.next() → { value: 4, done: false }
x.next() → { value: 5, done: false }
x.next() → { value: 6, done: false }
x.next() → { value: 7, done: false }
x.next() → { value: undefined, done: true }
x.next() → { value: undefined, done: true }
x.next() → { value: undefined, done: true }
```

Se viene passato un argomento a **`next()`**, questo sarà trattato come il valore restituito da **`yield`** *dentro il corpo* al momento della ripresa

```
for (var i of range(4,8))
  console.log(i)

4
5
6
7
```

Con il **`for (… of …)`**

```
[...range(3,7)] → [3,4,5,6]
```

Con l'operatore spread

# Generatori, parametri, terminazione ed eccezioni

I generatori hanno alcune caratteristiche aggiuntive che vale la pena di citare.

1. Una funzione generatore può avere dei parametri, come abbiamo visto. Ma anche il metodo **`next()`** di un generatore può avere un parametro – uno soltanto, ma può essere un oggetto e contenere qualunque cosa serva. Il valore passato diventa il valore di ritorno della **`yield`**. Il corpo del generatore è libero di usare l'argomento come meglio crede.
2. I generatori hanno anche due metodi aggiuntivi, opzionali:
    a. **`return(v)`**, la cui invocazione fa sì che la successiva chiamata **`next()`** si comporti come se fosse stato eseguito un return **`v`** nel corpo, ovvero terminando l'iterazione.
    b. **`throw(e)`**, la cui invocazione fa sì che la successiva chiamata **`next()`** si comporti come se fosse stato eseguito un **`throw`** e nel corpo, anche qui terminando l'iterazione.

# Nota su generatori ricorsivi

Esiste una forma particolare di **`yield`**, che è denotata con **`yield* E`**, con il seguente comportamento: la **`E`** deve a sua volta essere iterabile; in questo caso, la **`yield*`** restituisce al chiamante tutti i valori forniti dall'iterazione su **`E`**, uno alla volta, e quando **`E`** è terminato, procede con l'esecuzione del comando successivo come un normale **`yield`**.

Questa forma è comodissima per implementare generatori su strutture ricorsive (vedremo degli esempi nelle esercitazioni), o più in generale per delegare una parte della propria generazione a un altro generatore.

Per esempio, **`*merge(g1,g2) { yield* g1; yield* g2 }`** è un generatore che scorre in sequenza **`g1`** e poi **`g2`**.

# Generatori e classi

```
class Poesia {
  #testo
  constructor(t) {this.#testo=t}
  *parole() {
    var i=0
    while (true) {
      let f=this.#testo.indexOf(" ",i)
      if (f>=0)
        yield this.#testo.slice(i,f)
      else {
        yield this.#testo.slice(i)
        break
      }
      i=f+1
    }
  }
}
```

```
[...p.parole()] →
[
  'La',          'vispa',
  'Teresa',      'avea',
  'fra',         "l'erbetta",
  "d'un",        'tratto',
  'sorpresa',    'gentil',
  'farfalletta'
]
```

```
var p=new Poesia("La vispa Teresa avea fra l'erbetta d'un tratto sorpresa gentil farfalletta")
```

# Q & A

# Esercizio 1 - Coda

Si definisca una classe **`Queue`** che implementi una coda FIFO (*First In First Out*) offrendo i seguenti metodi

- **`put(x)`** → per aggiungere l'elemento **`x`** in fondo alla coda
- **`get()`** → per ottenere (e rimuovere) l'elemento in testa alla coda
- **`size()`** → per ottenere la dimensione della coda
- **`empty()`** → per verificare se la coda sia vuota
- **`toString()`** → per ottenere una rappresentazione della coda come stringa

# Esercizio 2 - Coda Limitata

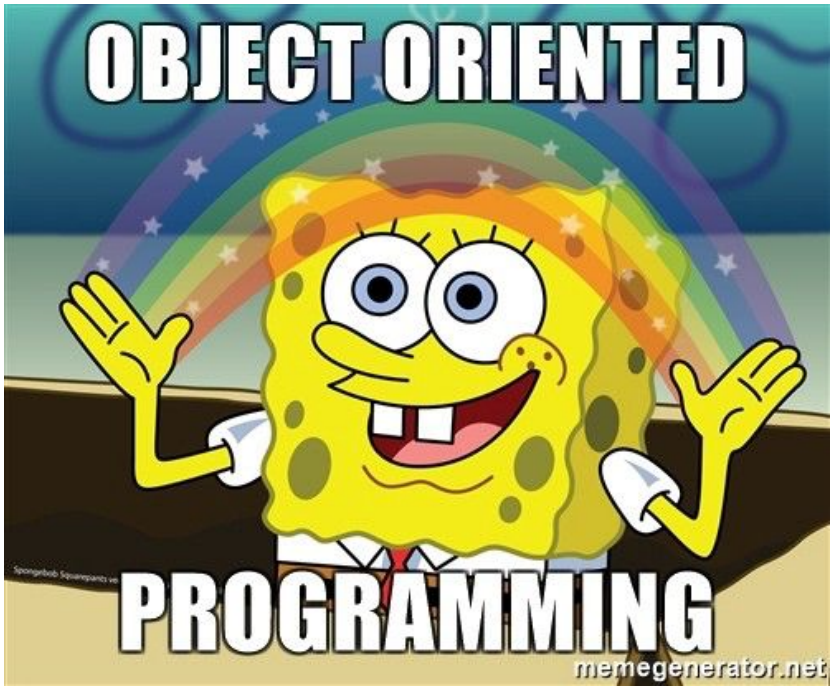
Si definisca una classe **`LimitedQueue`** che implementi una coda FIFO (*First In First Out*) offrendo i seguenti metodi

- **`put(x)`** → per aggiungere l'elemento **`x`** in fondo alla coda (se c'è posto)
- **`get()`** → per ottenere (e rimuovere) l'elemento in testa alla coda
- **`size()`** → per ottenere la dimensione della coda
- **`empty()`** → per verificare se la coda sia vuota
- **`toString()`** → per ottenere una rappresentazione della coda come stringa
- **`full()`** → per verificare se la coda contenga il numero massimo di elementi

Il costruttore di **`LimitedQueue`** prenderà un numero **`max_size`** come parametro, che denoterà la dimensione massima della coda stessa.

**IDEONA: copiamo e modifichiamo `Coda`**

# Ma.. “copiare” è davvero una buona idea? (in questo caso, almeno)



```
class Queue {
    #array = [];
    put(x) {
        this.#array[this.#array.length] = x;
    }
    get() {
        let x = this.#array[0];
        this.#array = this.#array.slice(1);
        return x;
    }
    size() {
        return this.#array.length;
    }
    empty() {
        return this.#array.length == 0;
    }
    toString() {
        return `Queue with ${this.size()} elements`;
    }
}
```

```
class LimitedQueue extends Queue {
    #max_size;
    constructor(max_size) {
        super();
        this.#max_size = max_size;
    }
    put(x) {
        if(this.full()) {
            console.log('Queue is full!!');
        } else {
            super.put(x);
        }
    }
    full() {
        return this.size() == this.#max_size;
    }
    toString() {
        return super.toString() + ` of ${this.#max_size}`;
    }
}
```