

# Corso Front End Developer


## TypeScript

Emanuele Galli



[www.linkedin.com/in/egalli/](https://www.linkedin.com/in/egalli/)








# Eccezioni

- Gestione rigorosa degli errori
- Se l'eccezione non viene gestita, lo script termina 

```
function indexToMonthName(index) {  
  if (index < 1 || index > 12) {  
    throw 'invalid month number';  
  }  
  
  let months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',  
    'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];  
  return months[index - 1];  
}
```



```
try {  
  console.log(indexToMonthName(1));  
  console.log(indexToMonthName(12));  
  console.log(indexToMonthName(0));  
} catch (exc) {  
  console.log(exc);  
} finally {  
  console.log('done');  
}
```



# Array



- Array con dimensione: `Array(size)`
- Inizializzazione: `fill()`
- Ordine in-place: `sort()`
- Copia di intervallo: `slice()`
- Copia filtrata: `filter()`
- Copia di array o iterable: `Array.from()`
- ...

```
let array = Array(5); // [undefined, ...]  
array.fill(0); // [0, ...]
```



```
// ...  
array.sort(  
  (left, right) => left == right ? 0 :  
    left < right ? -1 : 1);
```



```
let sliced = array.slice(1, 3);
```



```
let odds = array.filter(value => value % 2);
```



```
let chars = Array.from('hello');
```


# Set e Map

- collezioni iterabili in ordine di inserimento
- Set
  - valori unici (verifica via '===' ma NaN considerato === NaN)
  - add(), clear(), delete(), forEach(), has(), values(), size
- Map
  - Relazione chiave → valore
  - Le chiavi possono essere di qualunque tipo
  - clear(), delete(), entries(), forEach(), get(), has(), keys(), set(), values()

# Altri loop


for ... in  
(oggetti)

```
let props = { a: 1, b: 2, c: 3 };  
for (let prop in props) {  
    console.log(` ${prop} is ${x[prop]} `);  
}
```



for ... of  
(iterabili)

```
let ys = [1, 2, 3, 4, 5, 6];  
for (let y of ys) {  
    console.log(y);  
}
```



Array.forEach()

```
ys.forEach((y) => {  
    console.log(y);  
});
```



# TypeScript



- Linguaggio di programmazione, superset di JavaScript
- Nato nel 2012 (Anders Hejlsberg @ Microsoft) “JavaScript that scales”  
<https://www.typescriptlang.org/>
- Ben supportato da VS Code via Node JS
- Installazione via npm
  - `npm install -g typescript`
- tsc compila codice TypeScript in JavaScript (source to source compiler, transcompiler, o transpiler)



# Hello TypeScript

- Creazione di un file TypeScript
- Transpiler
  - tsc hello.ts
  - genera il file hello.js
- Il file di configurazione **tsconfig.json** automatizza la generazione dei file .js via tsc
- (Quasi) tutte le funzionalità JS sono supportate in TS

function greetings(name: string) {  
 return "Hello, " + name;  
}

type annotation  
opzionale



console.log(greetings("TypeScript"));

function greetings(name) {  
 return "Hello, " + name;  
}  
console.log(greetings("TypeScript"));

```
{  
  "compilerOptions": {  
    "target": "es6",  
    "module": "commonjs",  
    "outDir": "out"  
  }  
}
```



# Tipi

- Type-checking (opzionale) per scrivere e leggere più facilmente il codice
- Tipizzazione **statica**, specificata al momento della dichiarazione
  - `let i: number = 42;`
  - `function hello(name: string): string { /* ... */ }`
- Tipi primitivi JS
  - `boolean`
  - `number`
  - `string`
- Array
  - `type[]` o `Array<type>`
  - `let x: number[] = [42, 12];`
- `any` `/* ogni valore è ammissibile */`
- Tupla
  - `[type1, type2]`
  - `let x: [string, number] = ['hi', 42];`
- Enumeration 
  - `enum Role { Model, View, Controller };`
  - `let role: Role = Role.View;` 
- `void`
  - Funzione che non ha un return type



# class



```
class Person {  
  first: string;  
  last: string;
```

TypeScript

```
  constructor(first: string, last: string) {  
    this.first = first;  
    this.last = last;  
  }
```

No ctor overload in TS

```
  fullInfo(): string {  
    return this.first + ' ' + this.last;  
  }  
}
```

visibilità membri:  
public (default)  
o private



```
class Person {
```

```
  constructor(first, last) {  
    this.first = first;  
    this.last = last;  
  }
```

JavaScript  
ES6

```
  fullInfo() {  
    return this.first + ' ' + this.last;  
  }  
}
```

```
let p: Person = new Person('Tom', 'Jones');
```





# Pseudoproprietà: get e set

```
class Person {  
  // ...  
  
  get fullName() {  
    return this.first + ' ' + this.last;  
  }  
  
  set fullName(name: string) {  
    let buffer = name.split(' ');  
    this.first = buffer[0];  
    this.last = buffer[1];  
  }  
}
```



JavaScript ES6  
e TypeScript

```
let p = new Person('Tom', 'Jones');  
p.fullName = 'Bob Hope';  
console.log(p.fullName);
```





# Static



JavaScript ES6  
e TypeScript

```
class Person {  
  // ...  
  
  static merge(p1: Person, p2: Person) {  
    return new Person(p1.first + p2.first, p1.last + p2.last)  
  }  
}
```

```
let tom = new Person('Tom', 'Jones');  
let bob = new Person('Bob', 'Hope');  
  
console.log(Person.merge(tom, bob).fullName);
```





# Ereditarietà



JavaScript ES6  
e TypeScript

```
class Employee extends Person {  
  salary: number;  
  
  constructor(first: string, last: string, salary: number) {  
    super(first, last);  
    this.salary = salary;  
  }  
  
  fullInfo(): string {  
    return super.fullInfo() + ': ' + this.salary;  
  }  
}
```



```
let jon = new Employee('Jon', 'Voight', 2000);  
console.log(jon.fullInfo());
```



# interface



Possono essere usate per descrivere come devono essere strutturati oggetti

```
interface Person {  
  first: string;  
  last: string;  
}  
  
let tom: Person = {  
  first: 'Tom',  
  last: 'Jones'  
};
```

```
interface Message {  
  sender: string,  
  recipient?: string,  
  subject?: string,  
  message: string  
}
```



optional

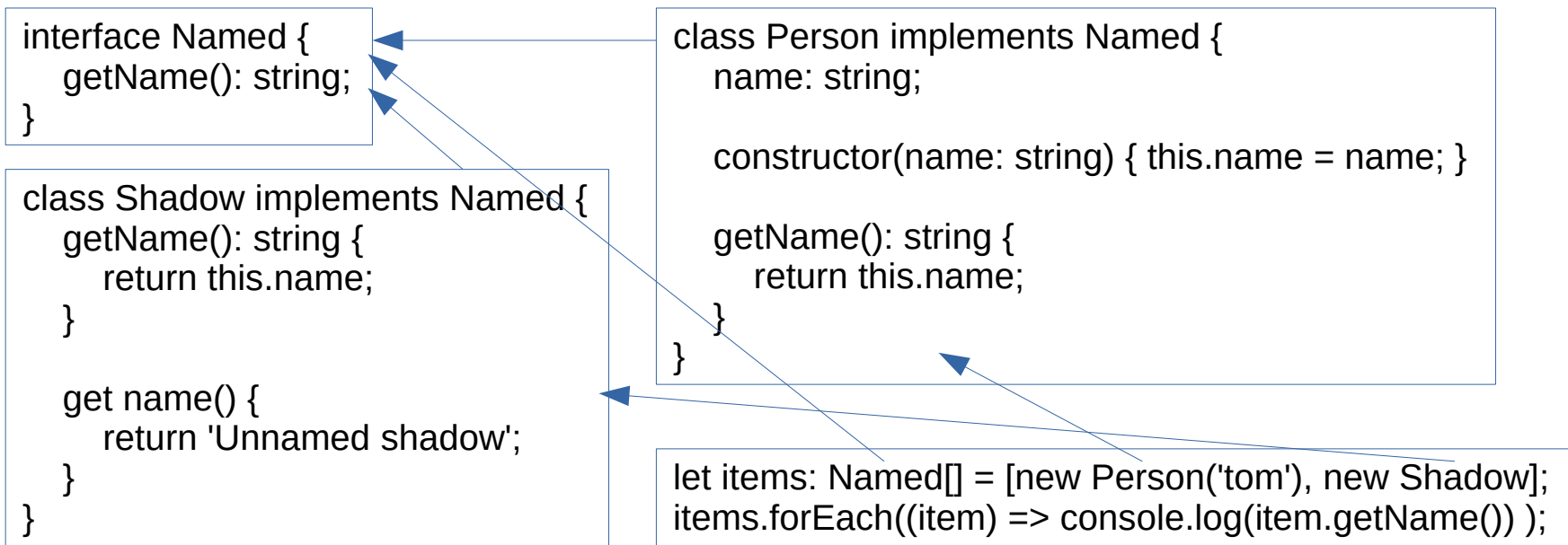
```
let bob: {  
  first: string;  
  last: string;  
} = {  
  first: 'Bob',  
  last: 'Coe'  
};
```

inline type  
annotation

```
function sayHello(person: Person, message: Message) {  
  // ...  
}
```

# Interfacce e classi

In ambito Object-Oriented l'interfaccia ha più propriamente lo scopo di dichiarare le funzionalità richiamabili sulle classi che la implementano



# generic


Cfr: `Array<T>.reverse()`

```
function reverseCopy<T>(data: T[]): T[] {  
    let result = [];  
    for (let i = data.length - 1; i >= 0; i--) {  
        result.push(data[i]);  
    }  
    return result;  
}
```



# Moduli



- Un file .ts è un modulo se ha almeno un import o un export
- By default il contenuto di un modulo è privato 
- export → permette l'accesso da altri file
- import → dichiara l'accesso ad altri file



```
export function hello(): void {  
    console.log('hello export');  
}
```

exporter.ts

```
import { hello } from './exporter';  
  
hello();
```



# Moduli

- Definizione ed esportazione anche separate
- Alias in esportazione o importazione con 'as'
- Default export
- Full import

```
export default function hi(): void {  
    console.log('hi');  
}
```

```
import hi from './exporter2';  
  
hi();
```

```
import * as cheers from './exporter';  
  
cheers.hello();  
cheers.goodbye();
```



```
function hello(): void {  
    console.log('hello export');  
}
```

```
function local(): void {  
    console.log('hello local');  
}
```

```
function bye(): void {  
    console.log('bye');  
}
```

```
export { hello, bye as goodbye }
```

```
import { hello as hi, goodbye } from './exporter';  
  
hi();  
goodbye();
```