Operazione Rif. P.A. 2022-17295/RER - Approvata dalla Regione Emilia Romagna con DGR 1379/2022 del 01/08/2022

Corso I.F.T.S. 2022 - 2023
"TECNICO PER LA PROGETTAZIONE E LO SVILUPPO DI APPLICAZIONI INFORMATICHE"

Periodo di svolgimento: novembre 2022 – giugno 2023

Durata: 800 ore

DISPENSE DIDATTICHE
03: TypeScript

Modulo n° 10: LINGUAGGI DI SCRIPTING
Modulo n° 14: TECNOLOGIE PER WEB APPLICATION

Docente: DANIELE ARDUINI

Fondazione En.A.I.P. S. Zavatta Rimini
Viale Valturio, 4 47923 Rimini
Tel. 0541.367100 – fax. 0541.784001
www.enaiprimini.org; e-mail: info@enaiprimini.org

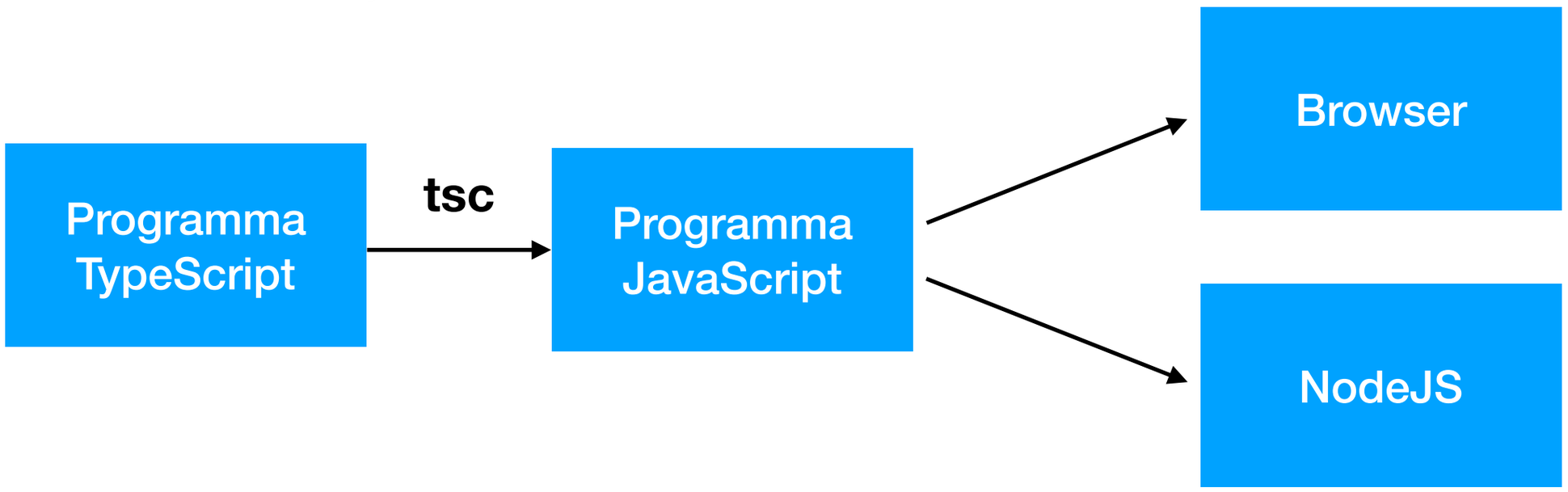
1. Introduzione a TypeScript

Cos’è TypeScript

- TypeScript è un linguaggio di programmazione open source, che estende JavaScript aggiungendo nuove funzionalità.

- In particolare, TypeScript fornisce tipi statici, classi, interfacce e molte altre funzionalità di programmazione orientata agli oggetti.

- Il codice TypeScript viene convertito in codice JavaScript da un particolare programma “traduttore” per essere eseguito nei browser o nei server Node.js.



1. Introduzione a TypeScript

Differenze tra TypeScript e JavaScript

- In JavaScript, le variabili sono dichiarate senza un tipo e il controllo del tipo avviene solo a runtime.

- In TypeScript, invece, si dichiarano esplicitamente i tipi di variabili e funzioni e questo controllo avviene a tempo di compilazione, prima dell'esecuzione.

- Questo significa che gli errori di tipo vengono rilevati in fase di sviluppo e non a runtime, rendendo il codice più robusto e riducendo gli errori di esecuzione.

- Il risultato è quello di avere minori bug durante lo sviluppo ed una migliore gestione in progetti complessi

1. Introduzione a TypeScript

Esempi di codice TypeScript

```
// Dichiarazione di una variabile con tipo esplicito
let age: number = 28;

// Dichiarazione di una interfaccia
interface Person {
  name: string;
  age: number;
}

// Dichiarazione di una funzione con tipo di ritorno esplicito
function greet(name: string): string {
  return `Hello ${name}!`;
}
```

1. Introduzione a TypeScript

Opportunità offerte da TypeScript

- TypeScript offre molte opportunità per la scrittura di codice più robusto e ben strutturato.

- Grazie ai tipi statici, il codice diventa più facile da comprendere e mantenere, poiché il tipo di ogni variabile o funzione è esplicitamente dichiarato.

- Inoltre, la presenza di classi e interfacce facilita la scrittura di codice orientato agli oggetti, migliorando la struttura e la modularità del codice.

Riferimenti:

- Documentazione ufficiale: https://www.typescriptlang.org/docs/

- La prima presentazione a JSConf 2012, da parte di Anders Heijlsberg & Luke Hoban: https://www.youtube.com/watch?v=3UTIcQYQ8Rw

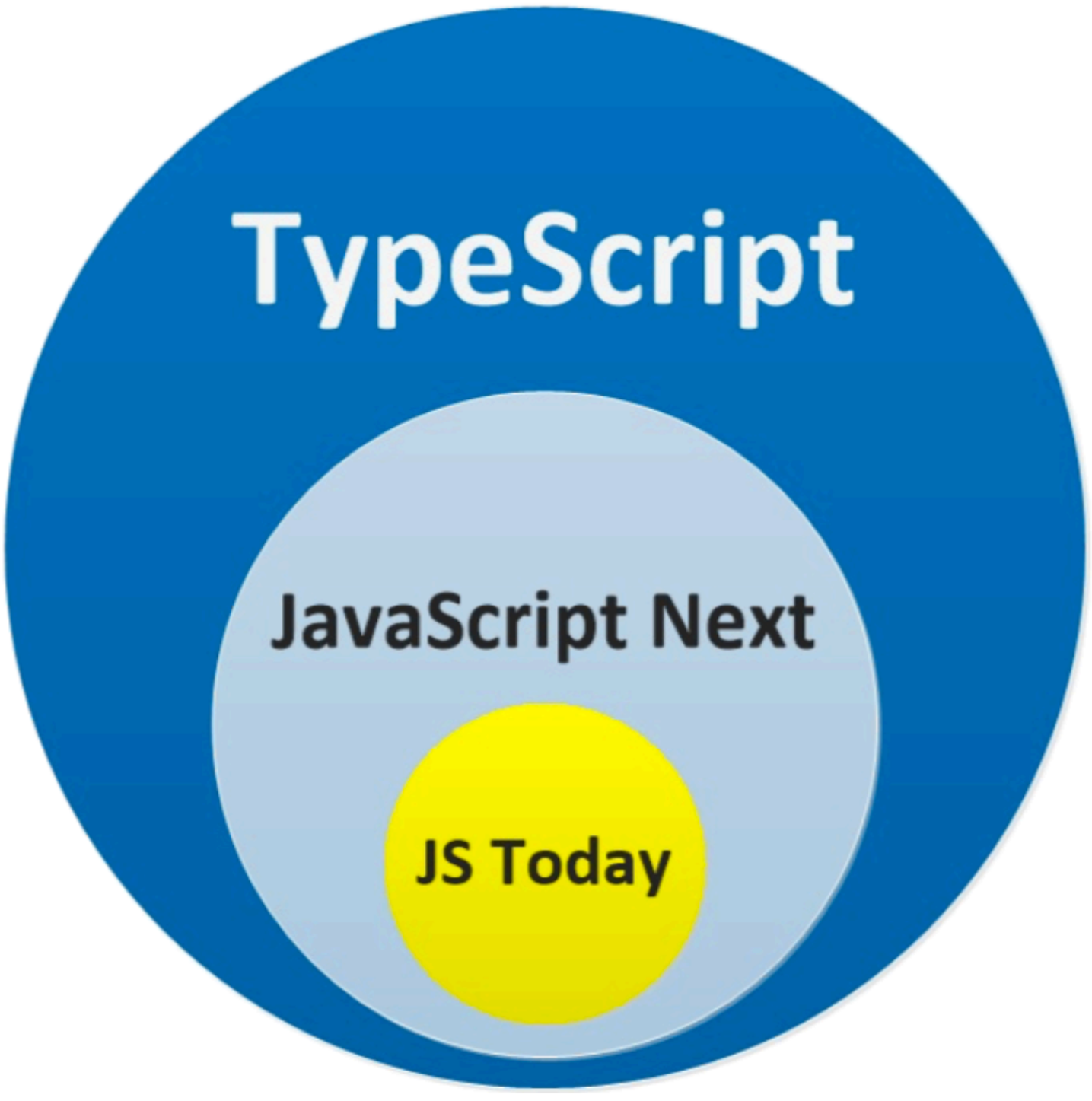
Typescript

- Sviluppato e gestito da Microsoft

- Usato in molti progetti, es. Angular

- 

Il JavaScript è Typescript
(ma non viceversa!)



Typescript: un JavaScript migliore

```
// La galleria degli orrori:

[] + [];                    // JS: "", TS: error
{} + [];                    // JS: 0, TS: error
[] + {};                    // JS: [object Object], TS: error
{} + {};                    // JS: NaN o [object Object][object Object]
                            // (dipende dal browser), TS: error

"hello" - 1;                // JS: NaN, TS: error

function add(a, b) {
    return
        a + b;              // JS: undefined, TS: error 'unreachable code'
}
```

Typescript: un JavaScript migliore

```
// La galleria degli orrori /2:


console.log(5 == "5");          // JS: true, TS: error

console.log(5 === "5");         // JS: false, TS: error
```

```
console.log("" == "0");      // JS: false, TS: error

console.log(0 == "");        // JS: true, TS: error

console.log("" === "0");     // JS: false, TS: error

console.log(0 === "");       // JS: false, TS: error
```

Hint: usate sempre === o !== salvo il controllo di null

https://dorey.github.io/JavaScript-Equality-Table/

Typescript

- Installazione e compilazione base
- Principali caratteristiche di Typescript
- Tipi ed interfacce
- Classi ed ereditarietà
- Moduli

Setup Ambiente

PACKAGE MANAGER FOR WINDOWS

https://chocolatey.org/

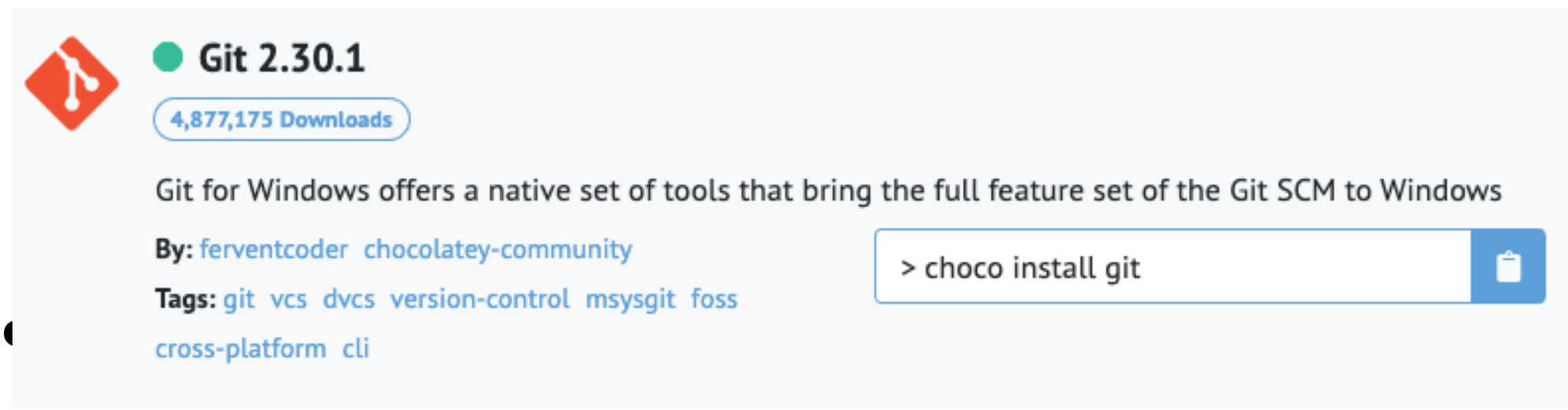
- Aprire il browser a:
  https://chocolatey.org/install
- Seguire istruzioni per l'installazione
- `c:\> choco -?`

Setup: GIT Repo

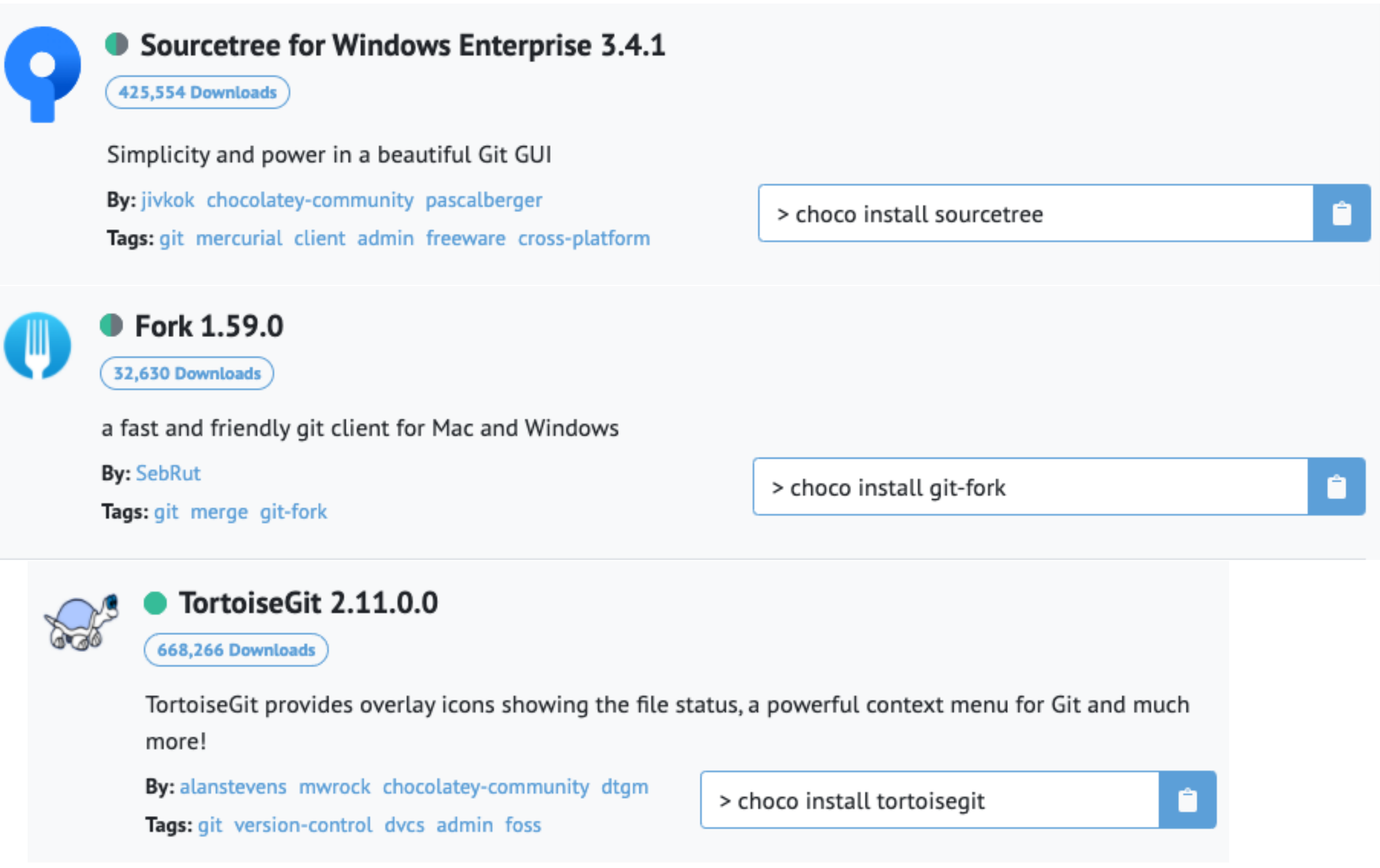
- Repository GIT: creare un account

https://bitbucket.org/

https://github.com/



Setup: GIT Client



Setup: VSCode

Visual Studio Code 1.53.2

1,809,328 Downloads

Visual Studio Code

By: jivkok chocolatey-community jgoemat pascalberger

Tags: microsoft visualstudiocode vscode development editor ide javascript typescript admin foss cross-platform

```
> choco install vscode
```

Setup: Node.JS

Node.js

- node: interprete Javascript “fuori dal browser”
- npm: gestore pacchetti node (per sviluppare applicazione)

- download e install della versione LTS:
  https://nodejs.org/en/download/

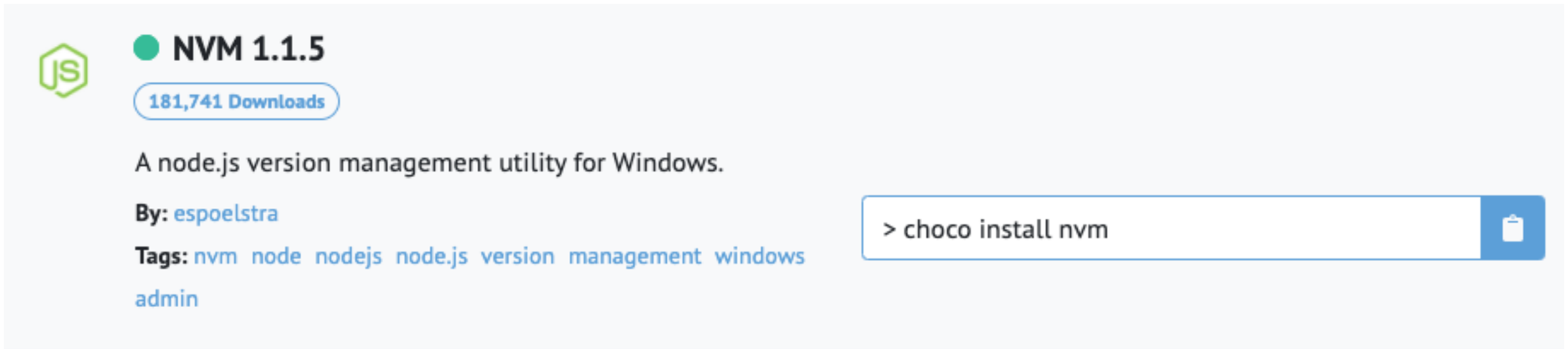
- test:

  c:\> node --version

Setup: NVM

NVM - Node Version Manager

- Consente di installare più versioni di Node.JS
- Consente di passare facilmente da una versione all'altra di Node.JS
- Ogni progetto potrebbe avere una diversa versione di Node.JS



Setup: NVM

```
C:\Users\Utente>nvm list available
```

CURRENT	**LTS**	OLD STABLE	OLD UNSTABLE
19.8.0	**18.15.0**	0.12.18	0.11.16
19.7.0	18.14.2	0.12.17	0.11.15
19.6.1	18.14.1	0.12.16	0.11.14
19.6.0	18.14.0	0.12.15	0.11.13
19.5.0	18.13.0	0.12.14	0.11.12
19.4.0	18.12.1	0.12.13	0.11.11
19.3.0	18.12.0	0.12.12	0.11.10
19.2.0	**16.19.1**	0.12.11	0.11.9
19.1.0	16.19.0	0.12.10	0.11.8
19.0.1	16.18.1	0.12.9	0.11.7
19.0.0	16.18.0	0.12.8	0.11.6
18.11.0	16.17.1	0.12.7	0.11.5
18.10.0	16.17.0	0.12.6	0.11.4
18.9.1	16.16.0	0.12.5	0.11.3
18.9.0	16.15.1	0.12.4	0.11.2
18.8.0	16.15.0	0.12.3	0.11.1
18.7.0	16.14.2	0.12.2	0.11.0
18.6.0	16.14.1	0.12.1	0.9.12
18.5.0	16.14.0	0.12.0	0.9.11
18.4.0	16.13.2	0.10.48	0.9.10

```
This is a partial list. For a complete list, visit https://nodejs.org/download/
release
```

Setup: NVM

```
C:\Users\Utente>nvm install 18.15.0
Downloading node.js version 18.15.0 (64-bit)...
…
C:\Users\Utente>nvm install 16.19.1
Downloading node.js version 16.19.1 (64-bit)...
…
C:\Users\Utente>nvm list
    18.15.0
    16.19.1

C:\Users\Utente>nvm use 18.15.0
Now using node v18.15.0 (64-bit)

C:\Users\Utente>node -v
v18.15.0
```

Installazione Typescript

- I Browser (o node) interpretano il Javascript
- E’ necessario un compilatore (transpiler) per trasformare i file *.ts in *.js

- Dopo aver installato Node, aprire terminale:

```
c:> npm install -g typescript
...
c:> tsc -v
```

Typescript: hello-world

- Creare una nuova directory: hello-world

```
c:> md hello-world
```

- All’interno della directory creare un file: hello-world.ts

hello-world.ts

```
console.log("Hello World!");

function saluto(msg) {
    console.log("Ciao " + msg);
}
```

Typescript: hello-world

- Compilare ed eseguire:

```
c:> cd hello-world
c:hello-world> tsc hello-world.ts
c:hello-world> node hello-world.js
```

Hello World!

Primi passi con i tipi

Nel linguaggio JavaScript non c'è alcun controllo sul tipo di dato assegnato ad una variabile:

```
console.log("Hello World!");

function saluto(msg) {
    console.log("Ciao " + msg);
}
```

Hello World!

```
console.log("Hello World!");

function saluto(msg: String) {
    console.log("Ciao " + msg);
}

saluto(“Daniele”);
```

Hello World!
Ciao Daniele

```
console.log("Hello World!");

function saluto(msg: String) {
    console.log("Ciao " + msg);
}

saluto(4);
```

?

(possibile distrazione!)

Primi passi con i tipi

Con TypeScript il compilatore controlla il tipo di dato assegnato ad una variabile:

```
console.log("Hello World!");

function saluto(msg: String) {
    console.log("Ciao " + msg);
}

saluto(4);
```

$ tsc hello-world.ts
hello-world.ts(7,8): error TS2345:
Argument of type '4' is not assignable to parameter of type 'String'.

TypeScript: tipi base

- Boolean
```
let fatto: boolean = true;
```
- Number
```
let anno: number = 2018;
```
- String
```
let colore: string = “rosso”;
```
- Array
```
let elenco: number[]       = [1, 2, 3, 4];
let elenco: Array<number> = [1, 2, 3, 4];
```
- Tupla (Array composto da tipi non omogenei)
```
let indirizzo: [string, number];
indirizzo = ["Corso d'Augusto", 4];
```
- Enum
```
enum Colore {Rosso, Giallo, Blu};
let c: Colore = Colore.Rosso;
```

TypeScript: tipi base

- Any

```
let nonsaprei: any = 4;
nonsaprei = 'pippo';
nonsaprei = true;
```

- Void

```
function avviso(): void {
    alert(“Premi OK per continuare.”);
}
```

- Null e undefined

```
let nonusabile: null = null;
let nonusabile: undefined = undefined;

Sono dei “sotto-tipi”, assegnabili anche agli altri tipi!
```

TypeScript: tipi custom

```
// è possibile creare un tipo "personalizzato" raggruppando più attributi base

type UserData = {
  id: number;
  name: string;
  surname?: string;
}

type UserAddress {
  address: string;
  city: string;
  state?: string;
  country: string;
}

type User = UserData & UserAddress;

let data: UserData = { id: 1, name: 'Daniele' };

let address: UserAddress =
                { address: 'via Flaminia, 12', city: 'Rimini', country: 'Italy' };

let user: User = { id: 2, name: 'Mario', surname: 'Rossi',
                   address: 'via Dante, 4', city: 'Rimini', country: 'Italy' };
```

Gli attributi con lo '?' in fondo al nome sono facoltativi.

Con l'operatore '&' si uniscono gli attributi, quindi User contiene gli attributi di UserData e di UserAddress

TypeScript: interface

```
// le interface sono "contratti" ai quali gli oggetti devono conformarsi

interface UserData {
  id: number;
  name: string;
  surname?: string;
}

interface UserAddress {
  address: string;
  city: string;
  state?: string;
  country: string;
}

interface User extends UserData, UserAddress {}

let data: UserData = { id: 1, name: 'Daniele' };

let address: UserAddress =
                { address: 'via Flaminia, 12', city: 'Rimini', country: 'Italy' };

let user: User = { id: 2, name: 'Mario', surname: 'Rossi',
                   address: 'via Dante, 4', city: 'Rimini', country: 'Italy' };
```

Gli attributi con lo '?' in fondo al nome sono facoltativi.

Con l'operatore 'extends' si uniscono gli attributi, quindi User contiene gli attributi di UserData e di UserAddress

var

```
var a = 10;
```

```
function f() {
    var a = 10;
    return function g() {
        var b = a + 1;
        return b;
    }
}

var g = f();
g(); // returns '11'
```

```
function f() {
    var a = 1;

    a = 2;
    var b = g();
    a = 3;

    return b;

    function g() {
        return a;
    }
}

f(); // returns '2'
```

var

```
function f(shouldInitialize: boolean) {
    if (shouldInitialize) {
        var x = 10;
    }
    return x;
}

f(true);  // returns '10'
f(false); // returns 'undefined'
```

```
function sumMatrix(matrix: number[][]) {
    var sum = 0;
    for (var i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (var i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }
   return sum;
}
```

var

```
for (var i = 0; i < 10; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
}
```

0
1
2
3
4
5
6
7
8
9

?

10
10
10
10
10
10
10
10
10
10

var

```
for (var i = 0; i < 10; i++) {
    // capture the current state of 'i'
    // by invoking a function with its current value
    (function(i) {
        setTimeout(function() { console.log(i); }, 100 * i);
    })(i);
}
```

IIFE - an Immediately Invoked Function Expression

let

```
let hello = "Hello!";
```

```
function f(input: boolean) {
    let a = 100;

    if (input) {
        // Still okay to reference 'a'
        let b = a + 1;
        return b;
    }

    // Error: 'b' doesn't exist here
    return b;
}
```

let

```
try {
    throw "oh no!";
}
catch (e) {
    console.log("Oh well.");
}

// Error: 'e' doesn't exist here
console.log(e);
```

```
a++; // illegal to use 'a' before it's declared;
let a;
```

let

```
for (let i = 0; i < 10 ; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
}
```

```
0
1
2
3
4
5
6
7
8
9
```

workspace

- Come inizializzare una cartella per lo sviluppo:
- Aprire terminale:

c:> **md workspace**
c:> **cd workspace**
c:workspace> **npm init**
c:workspace> **npm install lite-server --save-dev**
c:workspace> **tsc --init**
c:workspace> **tsc**
c:workspace> **npm start**

esercizio 1

```
let bankAccount = {
    money: 2000,
    deposit(value) {
        this.money += value;
    }
};

let myself = {
    name: "Max",
    bankAccount: bankAccount,
    hobbies: ["Sports", "Cooking"]
};

myself.bankAccount.deposit(3000);

console.log(myself);
```

DA FARE:
Aggiungere i tipi dove mancano!
…e crearne di nuovi per riutilizzarli!

Debugging TypeScript

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": true,
    "sourceMap": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

sourceMap:
Fornisce al debugger JavaScript dentro al Browser i riferimenti ai sorgenti TypeScript

noImplicitAny:
configura come si comporta quando non viene esplicitato il tipo di una variabile

Tutte le opzioni qui:
https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

Classi ed Oggetti

```
class Person {
    name: string;
    private type: string;
    protected age: number = 27;

    constructor(name: string, public username: string) {
        this.name = name;
    }
}

let person = new Person("Max", "max");
```

public: visibile all’esterno
private: NON visibile all’esterno
protected: visibile solo nella classe o sotto

Tipo di dato **Valore del dato**

Classi ed Oggetti

```
class Person {
    name: string;
    private type: string;
    protected age: number = 27;

    constructor(name: string, public username: string) {
        this.name = name;
    }
}

let person = new Person("Max", "max");
```

Il constructor consente anche di definire direttamente proprietà:

```
constructor(name: string, public username: string) {
    this.name = name;
}
```

Ereditarietà

```
class Max extends Person {
    // name = "Max";

    constructor(username: string) {
        super("Max", username);
        this.age = 31;
    }
}
const max = new Max("max");
console.log(max);
```

Max contiene TUTTE le proprietà Person con in più le proprie.

con super() si accede al costruttore di Person

getter e setter

Come posso accedere alle proprietà “private” o “protected”?
Attraverso delle funzioni che mediano l'accesso alla proprietà denominate getter/setter

```
class Plant {
    private _species: string = "Default";

    get species() {                         // getSpecies() {
        return this._species;
    }

    set species(value: string) {     // setSpecies(...) {
        if (value.length > 3) {
            this._species = value;
        } else {
            this._species = "Default";
        }
    }
}
```

Utile per elaborare i valori prima di leggere/scrivere le proprietà!

getter e setter

Esempio:

```
    let plant = new Plant();
(1) console.log(plant.species);
    plant.species = "AB";
(2) console.log(plant.species);
    plant.species = "Green Plant";
(3) console.log(plant.species);
```

(1) “Default”
(2) “Default”
(3) “Green Plant”

metodi e proprietà “static”

- Metodi e proprietà appartenenti alla classe, non legati ad una particolare istanza
- Sono accessibili tramite il nome della classe.

```
// Static Properties & Methods
class Helpers {
    static PI: number = 3.14;
    static calcCircumference(diameter: number): number {
        return this.PI * diameter;
    }
}
console.log(2 * Helpers.PI);
console.log(Helpers.calcCircumference(8));
```

Moduli

- Meccanismo per organizzare un'applicazione in maniera "modulare"
- Suddivisione del codice in più files
- Richiede un "module loader"
- Miglioramento rispetto alla gestione dei JavaScript Namespace
- Dichiarazione esplicita delle dipendenze

```
import * as Circle from "./math/circle";
import calc from "./math/rectangle";

console.log(Circle.PI);
console.log(Circle.calculateCirumcumference(10));
console.log(calc(20, 50));
```

Interfacce

```
function printLabel(labelledObj: { label: string }) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

PROBLEMA:

Ho scritto una funzione che accetta un oggetto (JSON) avente determinate proprietà che vengono elaborate al suo interno (es stampa)

Come posso riusarla per tutti gli oggetti che hanno proprietà “compatibili” anche se non hanno tutta la struttura identica?

```
let prodotto = {
  nome: “libro”, prezzo: “15.00”, label: “Zanna Bianca” };
};
printLabel(prodotto);
```

Interfacce

```
interface LabelledValue {
    label: string;
}

function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);

let prodotto = {
  nome: "libro", prezzo: "15.00", label: "Zanna Bianca"
};
printLabel(prodotto);
```

Interfaccia:
Possibilità di esprimere quali sono le “parti” comuni di oggetti che hanno struttura/tipologia diversa

Consentono di ri-utilizzare procedure che leggono/scrivono le parti comuni

Interfacce

```
function greet(person: {firstName: string}) {
    console.log("Hello, " + person.firstName);
}

function changeName(person: {firstName: string}) {
   person.firstName = “Anna”;
}
```

- **Se l’oggetto “person” avesse molte proprietà?**
- **Se l’oggetto “person” lo usassi tante volte?**
- **Se dovessi modificare l’oggetto “person”?**

Interfacce

```
interface NamedPerson {
    firstName: string;
    age?: number;
    [propName: string]: any;
    greet(lastName: string): void;
}

function greet(person: NamedPerson) {
    console.log("Hello, " + person.firstName);
}
function changeName(person: NamedPerson) {
   person.firstName = “Anna”;
}
```

- **Definizione di “person” in un solo luogo**
- **Controllo di tipo fatto dal compilatore**
- **Molto più flessibile di “type” o “class”**
- **“age?” : proprietà facoltativa di nome “age”**
- **“[propName: …]” : proprietà di nome non noto**
- **“greet(…)” : proprietà funzione**

Interfacce con le classi

```
class Person implements NamedPerson {
    firstName: string;
    lastName: string;

    greet(lastName: string) {
        console.log(“Hi, I am "+this.firstName+" "+lastName);
    };
}

const myPerson = new Person();
myPerson.firstName = "Maximilian";
myPerson.lastName = "Anything";
greet(myPerson);
myPerson.greet(myPerson.lastName);
```

- **Implementare un’interfaccia significa fornire la definizione di tutte le sue proprietà**

Decorators

- Funzioni speciali che si possono “associare” a:
  - classi
  - metodi
  - proprietà
  - parametri di metodi

- Programma sul programma: Meta-programmazione

```
function logged(constructorFn: Function) {
    console.log(constructorFn);
}

@logged
class Person {
    constructor() {
        console.log("Hi!");
    }
}
```

Decorator Factory

- **La funzione decorator ritorna il nome del vero decoratore**

  Utile per parametrizzare i decorator

```
// Factory
function logging(value: boolean) {
    return value ? logged : null;
}

@logging(true)
class Car {

}
```

Decorator Avanzati

- **Attraverso i decorator si possono modificare le classi alle quali sono associati!**
- **Qui la funzione printable(), associata alla classe Plant tramite il decorator @printable, aggiunge un nuovo metodo print() ai metodi della classe**

```
function printable(constructorFn: Function) {
    constructorFn.prototype.print = function () {
        console.log(this);
    }
}

@logging(false)
@printable
class Plant {
    name = "Green Plant";
}
const plant = new Plant();
(<any>plant).print();
```

Funzioni anonime

In TypeScript quando si assegna ad una variabile una funzione (funzione come valore) è possibile utilizzare una forma compatta (ed anonima):

```
let f =
  function dado() { return Math.floor(Math.random()*6+1); };

let g = () => { return Math.floor(Math.random()*6+1); };
let h = () => Math.floor(Math.random()*6+1);


let hello = () => { console.log("hello!"); };
hello();

const printMessage = (msg: string) => { console.log(msg); };
printMessage("Ciao!");


setTimeout(() => { console.log("hello!"); }, 2000);
```

Callback

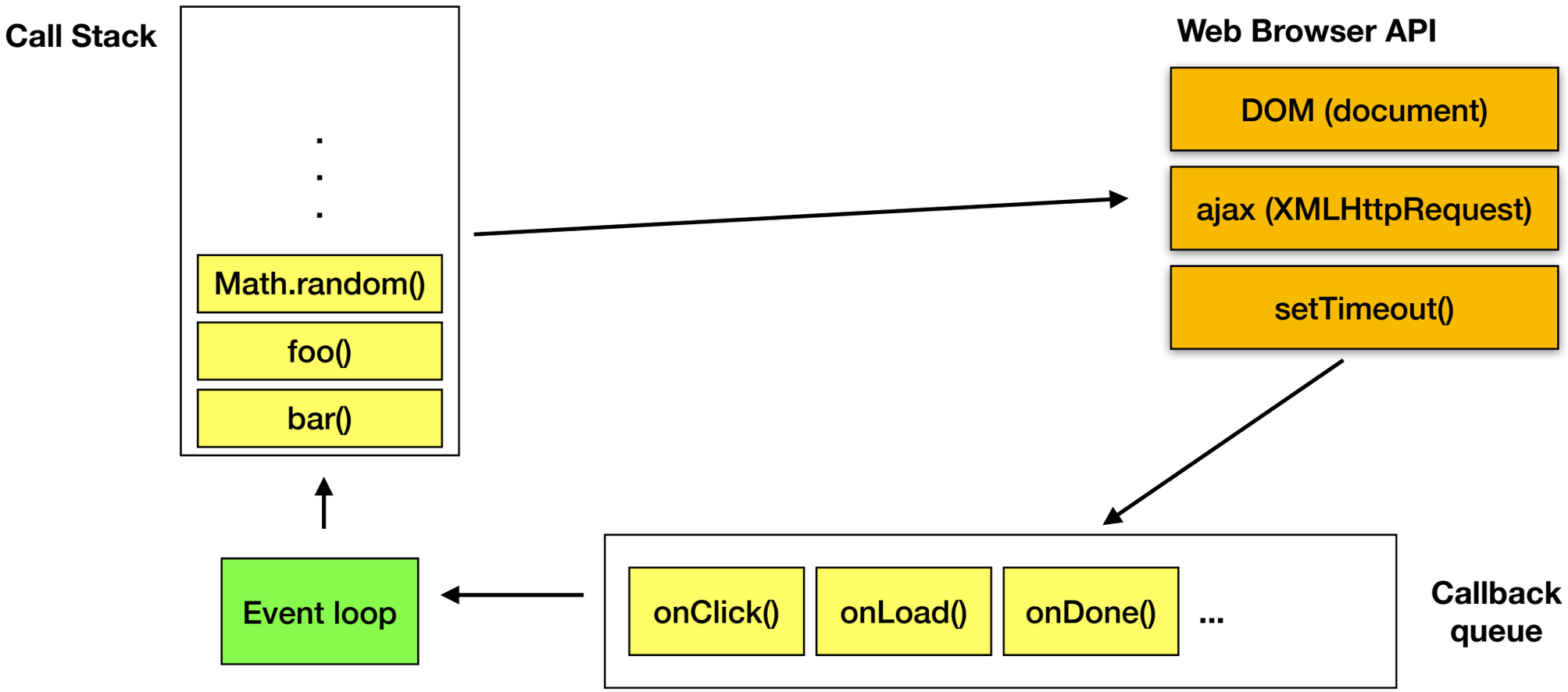
JavaScript è un linguaggio utilizzato in contesti “event driven”:
- eventi asincroni (rete, risposte del server, browser, risultati di elaborazioni)
- interazioni utente

Le funzioni Callback sono funzioni che vengono eseguite in modo asincrono, o in un tempo successivo. L’esecuzione del programma non segue l’ordine “tradizionale” top-down, i programmi con funzioni asincrone possono eseguire codice in esse contenuto in tempi differenti in base all’ordine ed alla velocità di altri fattori esterni (richieste http, letture dal file-system, …)

```
const f = () => { console.log(“hello!”); }
setTimeout(f, 2000);
console.log(“chiamato f()”);
```

JavaScript nei Browser

Single Thread
All'interno dei Browser il JavaScript di uno pagina viene eseguito in un singolo Thread, indipendentemente dal numero di CPU presenti nella macchina il codice viene eseguito come se ci fosse una sola CPU.
Con tale modello, Come è possibile "dare l'impressione" di eseguire più task in parallelo?



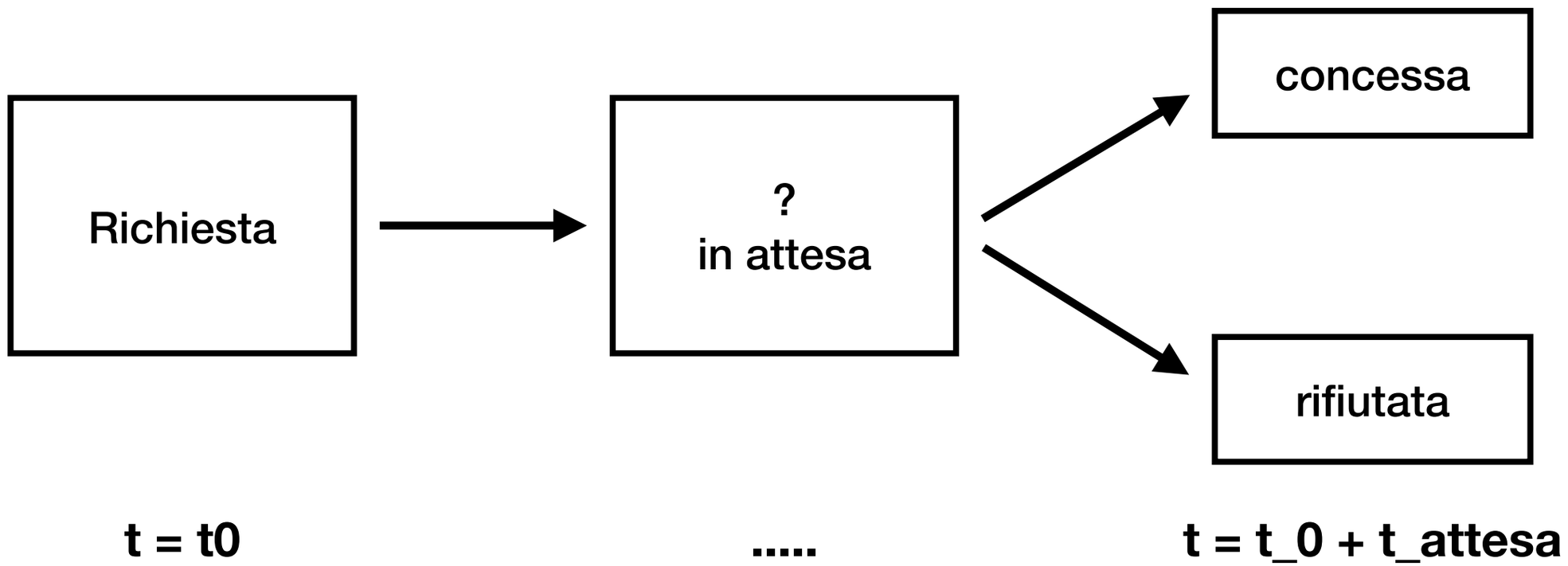
https://www.youtube.com/watch?v=8aGhZQkoFbQ

https://stackblitz.com/edit/javascript-chiamate-bloccanti

Cos'è una Promise

Immagina di essere un ragazzo.
Tua madre ti **Promette** che ti darà un **nuovo telefono** la prossima settimana se passi la verifica.

Non saprai se avrai il telefono fino alla prossima settimana: tua madre potrebbe prendere il telefono oppure no, se non rimane soddisfatta.



Promise

Rappresentano lo standard per la programmazione asincrona in JavaScript (partire dalle specifiche di ECMAScript 2015)

Stato	Descrizione
resolved	Una promise è risolta quando il valore che rappresenta diviene disponibile, cioè quando l'attività asincrona restituisce un valore
rejected	Una promise è rigettata quando l'attività asincrona associata non restituisce un valore o perché si è verificata un'eccezione oppure perché il valore restituito non è considerato valido
pending	Una promise è pendente quando non è né risolta né rigettata, cioè la richiesta di esecuzione di un'attività asincrona è partita ma non abbiamo ancora ricevuto un risultato

Promise

Creazione di una promise:

```
const handler = function(resolve, reject) {
    let condizione = false;

    // calcolo che impiega del tempo
    // …
    // condizione = true;
    // ...

    if (condizione) {
       resolve(valore);
     } else {
       reject("Errore nell'elaborazione");
    }
}

const promise = new Promise(handler);
```

Promise

Uso di una promise:

```
…

const promise = new Promise(handler);

console.log(“ottengo il risultato 1:”);
promise.then((risultato) => {
  console.log(“risultato 1 ottenuto: ”, risultato);
});

console.log(“ottengo il risultato 2:”);
promise.then((risultato) => {
  console.log(“risultato 2 ottenuto: ”, risultato);
});
```

Promise

Gestione degli errori:

```
…

const promise = new Promise(handler);

promise
  .then((risultato) => {
    console.log(“risultato 1 ottenuto: ”, risultato);
  })
  .catch((error) => {
    console.log(“risultato in errore: ”, error);
  });

// console.log(“pippo”);
```

Promise: esercitazione

promise
 https://stackblitz.com/edit/typescript-promise-1

```
promise.then(risultato => {
  console.log("risultato 1 ottenuto: ", risultato);
});

promise.catch(error => {
  console.log("risultato in errore: ", error);
});
```

se ritorno un valore in in un callback della then diventa:

```
promise
  .then(risultato => {
    console.log("risultato 1 ottenuto: ", risultato);
    return true;
  })
  .catch(error => {
    console.log("risultato in errore: ", error);
  });

let promiseBoolean: Promise<boolean> =
  promise.then(risultato => {
    console.log("risultato 1 ottenuto: ", risultato);
    return true;
  });

promiseBoolean.catch(error => {
  console.log("risultato in errore: ", error);
});
```

Promise: in serie

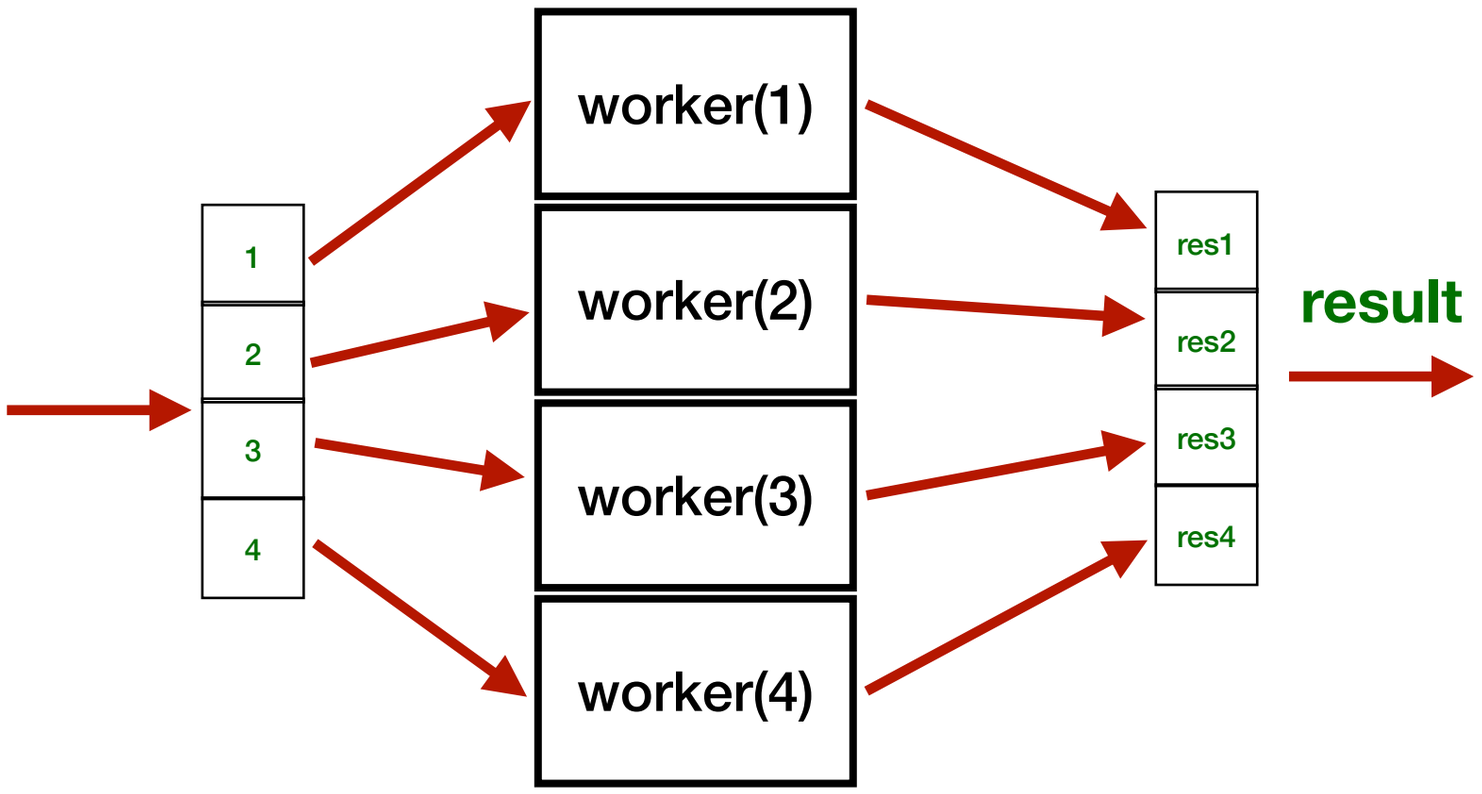
Quando è necessario mettere in sequenza più operazioni asincrone: l'operazione asincrona successiva deve partire dopo che è terminata quella precedente


```
worker(1).then(res1 => worker(2)).then(res2 => worker(3)).then(res3=> worker(4)).then(res4 => result);
```

```
worker(1)
   .then(res1 => worker(2))
   .then(res2 => worker(3))
   .then(res3 => worker(4))
   .then(res4 => result);
```

Promise: in parallelo

Quando le operazioni asincrone possono essere avviate in parallelo perché sono indipendenti tra loro: si avviano le operazioni tutte insieme e si attende che finiscano tutte.




```
let inputList: number[] = [ 1, 2, 3, 4 ];
let promiseList: Promise<number>[] = [ worker(1), worker(2), worker(3), worker(4) ];
Promise.all( promiseList ).then( result => ...);
```

Promise: serie/parallelo /1

https://stackblitz.com/edit/promise-in-sequenza-2

```
/**
 * simula un'attività che impiega del tempo, es. una richiesta http o calcolo
 * in questo caso dopo un ritardo variabile tra 0 e 4 secondi risponde con il
 * valore passato come parametro moltiplicato per 10
 */
function worker(id: number = 0): Promise<number> {
  return new Promise<number>((resolve, reject) => {
    setTimeout(() => {
      try {
        // semplice calcolo
        const result = id * 10;
        console.log('worker('+id+') => '+result);
        resolve(result);
      } catch(err) {
        reject(err);
      }
    }, Math.random() * 4000);
  });
}
```

Promise: serie/parallelo /2

```
/**
 * Quando le operazioni devono essere effettuate in sequenza e ne conosciamo
 * a priori il numero, si concatenano esplicitamente le Promise:
 * quella precedente effettua il return del dato, quella successiva si mette
 * dentro la then() della precedente.
 */
function onSerie(): Promise<number[]> {
  const result: number[] = [];

  return worker(1)
    .then((res1) => {
      console.log("Operation done: ", res);
      result.push(res);
      return worker(2);
    })
    .then((res2) => {
      console.log("Operation done: ", res);
      result.push(res);
      return worker(3);
    })
    .then((res3) => {
      console.log("Operation done: ", res);
      result.push(res);
      return worker(4);
    })
    .then((res4) => {
      console.log("Operation done: ", res);
      result.push(res);
      return result;
    });
}
```

Promise: serie/parallelo /3

```
/**
 * Quando le operazioni devono essere effettuate in sequenza ma non ne conosciamo
 * a priori il numero, ma abbiamo a disposizione un array di operazioni sul quale
 * ciclare, si concatenano le Promise:
 * si cicla sull'array delle operazioni: ad ogni ciclo ci si aggancia alla then()
 * della Promise precedente, al termine si effettua un'unica then().
 */
function onSerieConArray(): Promise<number[]> {
  const result: number[] = [];
  const inputList: number[] = [
    1, 2, 3, 4
  ];

  let p: Promise<number> = null;
  for (let i = 0; i < inputList.length; i++) {
    if ( p == null ) {
      p = worker(inputList[i]);
    } else {
      p = p.then((res) => {
        console.log("Operation done: ", res);
        result.push(res);
        return worker(inputList[i]);
      });
    }
  }

  return p.then((res) => {
      console.log("Operation done: ", res);
      result.push(res);
      return result;
    });
}
```

Promise: serie/parallelo /3

```
/**
 * Quando le operazioni devono essere effettuate in sequenza ma non ne conosciamo
 * a priori il numero, e NON abbiamo neppure a disposizione un array di operazioni sul
 * quale ciclare, diventa necessario utilizzare un ciclo while con una funzione
 * ricorsiva:
 *
 * si chiama la funzione promiseWhile() con il primo valore da elaborare e si
 * chiama la then sul suo risultato.
 * All'interno della promiseWhile() si chiama il worker() e si aspetta il risultato,
 * se non ci sono ulteriori valori allora si ritorna un risultato semplice,
 * altrimenti si richiama ricorsivamente la funzione in modo che il risultato
 * è una promise
 */
function onSerieConWhile(): Promise<number[]> {
  const result: number[] = [];

  const inputStart = 1;
  const inputEnd = 4;

  const promiseWhile = (index: number): Promise<number> => {
    return worker(index)
      .then(res => {
        console.log("Operation done: ", res);
        result.push(res);
        if ( index < inputEnd ) {
          return promiseWhile(index+1);
        } else {
          return res;
        }
      });
  };

  return promiseWhile(inputStart)
    .then((res) => {
      // console.log("Operation done: ", res);
      // result.push(res);
      return result;
    });
}
```

Promise: serie/parallelo /4

```
/**
 * Quando le operazioni devono essere effettuate in parallelo e ne conosciamo
 * a priori il numero, si costruisce l'array delle Promise risultato e poi
 * vengono passate alla Promise.all()
 */
function onParallelo(): Promise<number[]> {
  const inputList: number[] = [
    1, 2, 3, 4
  ];

  const promiseList: Promise<number>[] = [];

  for (let i = 0; i < inputList; i++) {
      promiseList.push( worker(inputList[i]) );
  }

  return Promise.all(promiseList)
    .then((resultList) => {
      console.log("Operation done: ", resultList);
      return resultList;
    });
}
```

https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Promise

https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

Functional Programming

https://it.wikipedia.org/wiki/Functional_programming

https://codewords.recurse.com/issues/one/an-introduction-to-functional-programming

Paradigma di programmazione nel quale il flusso di esecuzione è rappresentato da una serie di valutazioni di **funzioni** matematiche

Es: estrazione dell'elemento più piccolo da un elenco:

ottieniElenco().ordinaElementi().estraiPrimo()

Programmazione funzionale con JavaScript - Anjana Vakil - JSConf

https://www.youtube.com/watch?v=e-5obm1G_FY

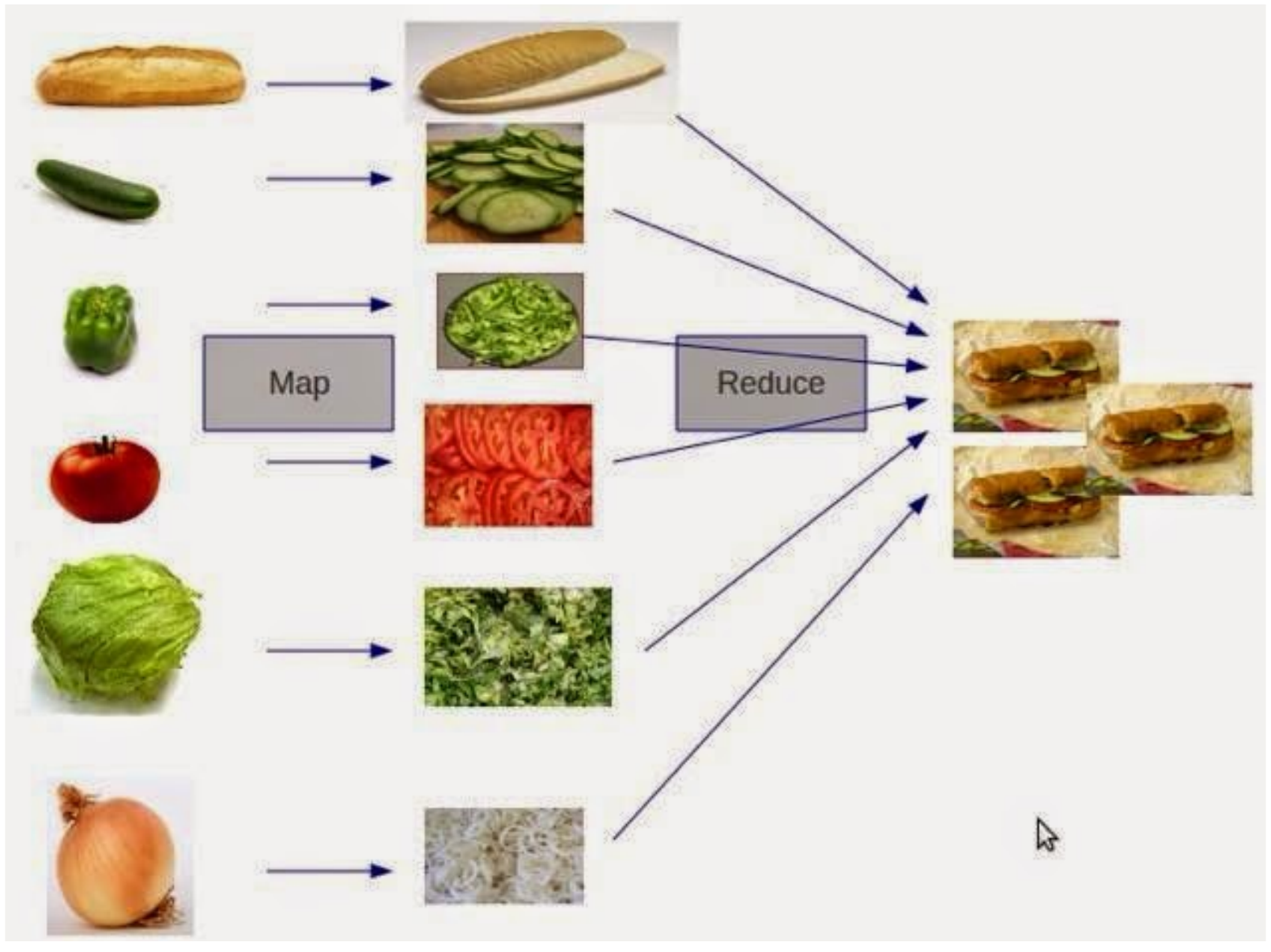
Functional Programming

Paradigma di programmazione nel quale il flusso di esecuzione è rappresentato da una serie di valutazioni di **funzioni** matematiche

Es: estrazione dell'elemento più piccolo da un elenco:

ottieniElenco().ordinaElementi().estraiPrimo()

Functional Programming



https://www.datasciencecentral.com/forum/topics/what-is-map-reduce

RxJS

Reactive eXtensions for JavaScript

Libreria per la creazione/elaborazione/manipolazione/gestione di “eventi” modellati come uno stream (flusso di eventi)

Reactive:
paradigma di programmazione dichiarativa, il programma “reagisce” agli eventi ed il risultato dell’elaborazione viene “elaborato” in tempo reale.

Similarità con il pattern Observable/Observer della programmazione ad oggetti.

Risorse:
https://en.wikipedia.org/wiki/Observer_pattern
https://www.learnrxjs.io
https://rxmarbles.com
https://rxviz.com
https://reactive.how

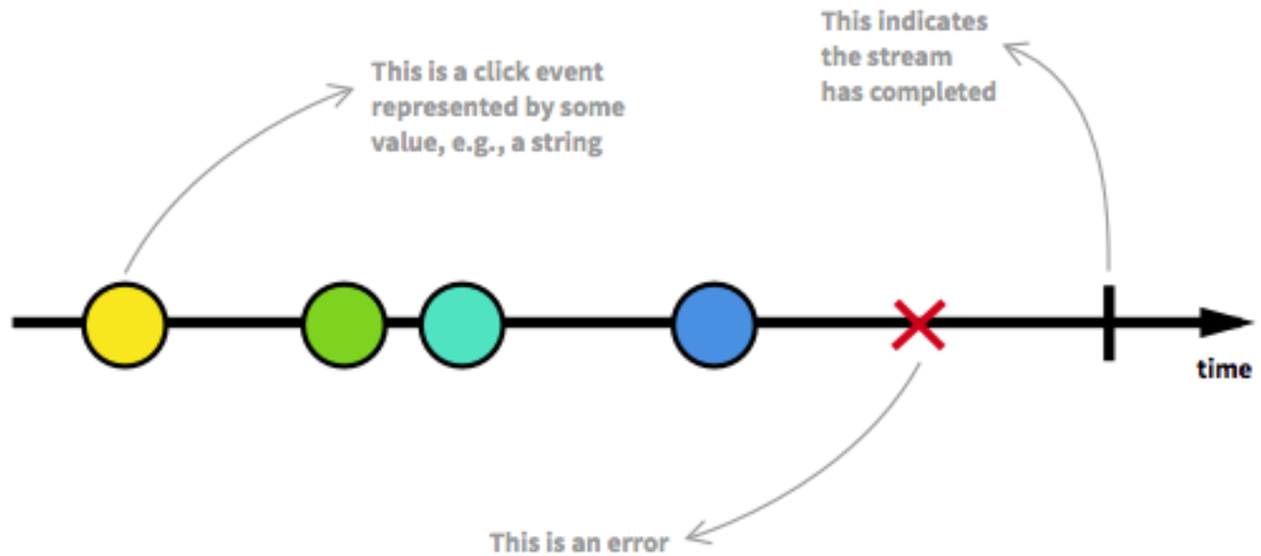
https://training.fabiobiondi.io/2019/09/15/introduzione-rxjs-parte-1-fundamentals/

RxJS: introduzione

Stream:
sequenza di eventi ordinati rispetto al tempo.

Gli eventi possono essere di 3 categorie:

- **un “valore” (di un determinato tipo)**
- **un “errore”**
- **il segnale di sequenza “completata”**



Gli eventi
- **si presentano in maniera asincrona**
- **si “catturano” attraverso delle funzioni di callback, una per ciascun tipo.**

```
const valueCallback =      (x) => { … };
const errorCallback = (error) => { … };
const completeCallback =    () => { … };
```

L’operazione di ascolto della sequenza di eventi per “catturarli”
si definisce iscrizione (subscribe)

```
eventStream$.subscribe(valueCallback, errorCallback, completeCallback);
```

RxJS: introduzione

Esercitazione:

c:\> git clone https://bitbucket.org/rdndnl/rxjs-intro.git
c:\> cd rxjs-intro
c:\> npm install
c:\> npm start

http://localhost:4200

TypeScript: futuro?

- 10 Things I Regret About Node.js
  (Ryan Dahl @ JSConf EU 2018)
    - https://www.youtube.com/watch?v=M3BM9TB-8yA

- Deno (A secure runtime for JavaScript and TypeScript)
    - https://deno.land/

- ~~TypeScript => JavaScript~~
  TypeScript => WebAssembly (https://webassembly.org/)
-