ITS INFORMATION AND COMMUNICATIONS TECHNOLOGY Academy

NOME MODULO: *WEB*

UNITÀ DIDATTICA: *WEB 1*

ALBERTO BELLEMO BULLO

PHILMARK INFORMATICA

https://youtube.com/@babsevensix

https://www.twitch.tv/babsevensix

Anno Accademico 2022-2023

ITS ICTAcademy

INFORMATION AND COMMUNICATIONS TECHNOLOGY

NB. Le presentazioni devono pervenire alla Segreteria Didattica ITS (segreteria@its-ictacademy.com) possibilmente a inizio Modulo o al termine della singola Unità Didattica. I materiali didattici verranno condivisi con gli Allievi solo al termine del Modulo.

Modulo WEB 1

INDICE DEGLI ARGOMENTI

- HTML
- CSS, Bootstrap
- Typescript e Javascript

Le funzioni

Le funzioni sono i blocchi primari di qualsiasi programma. In JavaScript, le funzioni sono la parte più importante poiché JavaScript è un linguaggio di programmazione funzionale. Con le funzioni, si possono implementare/imitare i concetti della programmazione orientata agli oggetti come classi, oggetti, polimorfismo e astrazione.

Le funzioni assicurano che il programma sia gestibile, riutilizzabile e organizzato in blocchi leggibili. Sebbene TypeScript fornisca il concetto di classi e moduli, le funzioni sono ancora parte integrante del linguaggio.

In TypeScript, le funzioni possono essere di due tipi: denominate e anonime.

Una funzione con nome è quella in cui si dichiara e si chiama una funzione con il suo nome.

```
function Sum(x: number, y: number) : number {
    return x + y;
}

Sum(2,3); // returns 5
```

Una funzione anonima è una funzione definita come un'espressione. Questa espressione è memorizzata in una variabile. Quindi, la funzione stessa non ha un nome. Queste funzioni vengono richiamate utilizzando il nome della variabile in cui è memorizzata la funzione.

```
let Sum = function(x: number, y: number) : number {
    return x + y;
}

Sum(2,3); // returns 5
```

Parametri di una funzioni

I parametri sono valori o argomenti passati a una funzione. In TypeScript, il compilatore si aspetta che una funzione riceva il numero esatto e il tipo di argomenti come definito nella firma della funzione. Se la funzione prevede tre parametri, il compilatore verifica che l'utente abbia passato i valori per tutti e tre i parametri, ovvero controlla le corrispondenze esatte. Questo è diverso da JavaScript, dove è accettabile passare meno argomenti di quanto si aspetta la funzione. I parametri che non ricevono un valore dall'utente sono considerati come *undefined*.

```
function Greet(greeting: string, name: string) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // Compiler Error: Expected 2 arguments, but got 1.
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```

TypeScript ha una funzionalità di parametro facoltativo. I parametri che possono ricevere o meno un valore possono essere aggiunti con un '?' per contrassegnarli come facoltativi.

```
function Greet(greeting: string, name?: string) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // OK, returns "Hi undefined!".
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```

Nota: Tutti i parametri facoltativi devono seguire i parametri obbligatori e devono trovarsi alla fine.

Parametri di una funzioni

TypeScript offre la possibilità di aggiungere valori predefiniti ai parametri. Quindi, se l'utente non fornisce un valore a un argomento, TypeScript inizializzerà il parametro con il valore predefinito. I parametri predefiniti hanno lo stesso comportamento dei parametri facoltativi. Se non viene passato un valore per il parametro predefinito in una chiamata di funzione, il parametro predefinito deve seguire i parametri richiesti nella firma della funzione. Pertanto, i parametri predefiniti possono essere omessi durante la chiamata di una funzione. Tuttavia, se una firma di funzione ha un parametro predefinito prima di un parametro obbligatorio, la funzione può comunque essere chiamata, a condizione che al parametro predefinito venga passato un valore *undefined*.

```
function Greet(name: string, greeting: string = "Hello") : string {
    return greeting + ' ' + name + '!';
}

Greet('Steve');//OK, returns "Hello Steve!"
Greet('Steve', 'Hi'); // OK, returns "Hi Steve!".
Greet('Bill'); //OK, returns "Hello Bill!"
```

Nota: Quando i parametri predefiniti precedono i parametri richiesti in una funzione, possono essere chiamati passando *undefined*.

```
function Greet(greeting: string = "Hello", name: string) : string {
    return greeting + ' ' + name + '!';
}

Greet(undefined, 'Steve');//returns "Hello Steve!"
Greet("Hi", 'Steve'); //returns "Hi Steve!".
Greet(undefined, 'Bill'); //returns "Hello Bill!"
```

Arrow functions

Le **arrow functions** sono utilizzate per le funzioni anonime, ad esempio per le espressioni di funzioni. Sono anche chiamate funzioni lambda in altri linguaggi.

```
let sum = (x: number, y: number) : number => {
    return x + y;
}

sum(10, 20); //returns 30
```

Quella che segue è una arrow function senza parametri.

```
let Print = () => console.log("Hello TypeScript");

Print(); //Output: "Hello TypeScript"
```

Inoltre, se il corpo della funzione è costituito da una sola istruzione, non sono necessarie le parentesi graffe e la parola chiave *return*, come mostrato di seguito.

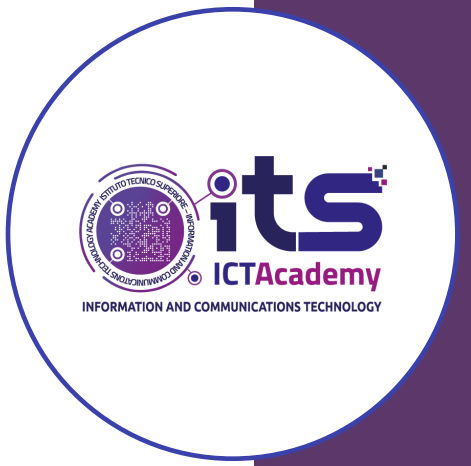
```
let sum = (x: number, y: number) => x + y;

sum(3, 4); //returns 7
```

Overloading delle funzioni

TypeScript fornisce il concetto di overloading di funzioni. È possibile avere più funzioni con lo stesso nome, ma diversi tipi di parametri e tipi restituiti. Tuttavia, il numero di parametri dovrebbe essere lo stesso.

```
function add(a: string, b: string) : string;
function add(a: number, b: number) : number;

function add(a: any, b: any) : any {
    return a + b;
}

add("Hello ", "Steve"); // returns "Hello Steve"
add(10, 20); // returns 30
```

Nota: l'overload della funzione con un numero diverso di parametri e tipi con lo stesso nome non è supportato.



Parametri rest

TypeScript ha introdotto i parametri rest per accogliere facilmente un numero n di parametri.

Quando il numero di parametri che una funzione riceverà non è noto o può variare, possiamo utilizzare i parametri rest. In JavaScript, questo si ottiene con la variabile "arguments". Tuttavia, con TypeScript, possiamo usare il parametro rest indicato da ellipsis ...

Possiamo passare zero o più argomenti al parametro rest. Il compilatore creerà un array di argomenti con il nome del parametro rest fornito da noi.

```
function Greet(greeting: string, ...names: string[]) {
    return greeting + " " + names.join(", ") + "!";
}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"
Greet("Hello");// returns "Hello !"
```

I parametri rest possono essere usati in funzioni, arrow function o classi.

Nota: i parametri rest devono essere gli ultimi nella definizione della funzione, altrimenti il compilatore TypeScript mostrerà un errore.

Interfacce

L'interfaccia è una struttura che definisce un contratto nell'applicazione. Definisce la sintassi per le classi da seguire. Le classi derivate da un'interfaccia devono seguire la struttura fornita dalla loro interfaccia.

Un'interfaccia è definita con la parola chiave *interface* e può includere proprietà e dichiarazioni di metodo utilizzando una funzione o una arrow function.

```
interface IEmployee {
    empCode: number;
    empName: string;
    getSalary: (number) => number; // arrow function
    getManagerName(number) : string;
}
```

L'interfaccia in TypeScript può essere utilizzata per definire un tipo e anche per implementarlo nella classe.

La seguente interfaccia `KeyPair` definisce un tipo di variabile.

```
interface KeyPair {
    key: number;
    value: string;
}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK
let kv2: KeyPair = { key:1, val:"Steve" }; // Compiler Error: 'val' doesn't exist in type 'KeyPair'
let kv3: KeyPair = { key:1, value:100 }; // Compiler Error: Type 'number' is not assignable to type 'string'
```

Interfacce

L'interfaccia in TypeScript viene utilizzata anche per definire un tipo di una funzione. Ciò garantisce la firma della funzione.

```
interface KeyValueProcessor {
    (key: number, value: string) : void;
};

function addKeyValue(key: number, value: string) : void {
    console.log('addKeyValue: key = ' + key + ', value = ' + value);
}

function updateKeyValue(key: number, value: string) : void {
    console.log('updateKeyValue: key = ' + key + ', value = ' + value);
}

let kvp: KeyValueProcessor = addKeyValue;
kvp(1, 'Bill'); //Output: addKeyValue: key = 1, value = Bill

kvp = updateKeyValue;
kvp(2, 'Steve'); //Output: updateKeyValue: key = 2, value = Steve
```

Il tentativo di assegnare una funzione con una firma diversa causerà un errore.

```
function delete(key: number) : void {
    console.log('Key deleted.');
}

let kvp: KeyValueProcessor = delete; //Compiler Error
```

Interfacce

Un'interfaccia può anche definire il tipo di un array in cui è possibile definire il tipo di indice e i valori.

```
interface NumList {
    [index: number] : number
}

let numArr: NumList = [1, 2, 3];
numArr[0];
numArr[1];

interface IStringList { [index: string] : string }

let strArr : IStringList = {};
strArr["TS"] = "TypeScript";
strArr["JS"] = "JavaScript";
```

Interfacce

A volte, possiamo dichiarare un'interfaccia con proprietà in eccesso ma non aspettarci che tutti gli oggetti definiscano tutte le proprietà dell'interfaccia specificate. Possiamo avere proprietà opzionali, contrassegnate da un "?". In tali casi, gli oggetti dell'interfaccia possono definire o meno queste proprietà.

```
interface IEmployee {
    empCode: number;
    empName: string;
    empDept?: string;
}

let empObj1 : IEmployee = { // OK
    empCode: 1,
    empName: "Steve"
}

let empObj2 : IEmployee = { // OK
    empCode: 1,
    empName: "Bill",
    empDept: "IT"
}
```

Interfacce

TypeScript fornisce un modo per contrassegnare una proprietà come in sola lettura. Ciò significa che una volta che a una proprietà viene assegnato un valore, non può essere modificato!

```
interface Citizen {
    name: string;
    readonly SSN: number;
}

let personObj: Citizen = {
    SSN: 110555444,
    name: 'James Bond'
}

personObj.name = 'Steve Smith'; // OK
personObj.SSN = 333666888; // Compiler Error
```

Interfacce

Le interfacce possono estendere una o più interfacce. Ciò rende le interfacce flessibili e riutilizzabili.

```
interface IPerson {
    name: string;
    gender: string;
}

interface IEmployee extends IPerson {
    empCode: number;
}

let empObj : IEmployee = {
    empCode: 1,
    name: "Bill",
    gender: "Male"
}
```

Interfacce

Analogamente a linguaggi come Java e C#, le interfacce in TypeScript possono essere implementate con una classe. La classe che implementa l'interfaccia deve essere rigorosamente conforme alla struttura dell'interfaccia.

```
interface IEmployee {
    empCode: number;
    name: string;
    getSalary: (empCode: number) => number;
}

class Employee implements IEmployee {
    empCode: number;
    name: string;

    constructor(code: number, name: string) {
        this.empCode = code;
        this.name = name;
    }

    getSalary(empCode: number) : number {
        return 20000;
    }
}

let emp = new Employee(1, "Steve");
```

Classi

Nei linguaggi di programmazione orientati agli oggetti come Java e C#, le classi sono le entità fondamentali utilizzate per creare componenti riutilizzabili. Le funzionalità vengono trasmesse alle classi e gli oggetti vengono creati dalle classi. Tuttavia, fino a ECMAScript 6 (noto anche come ECMAScript 2015), questo non era il caso di JavaScript. JavaScript è stato principalmente un linguaggio di programmazione funzionale in cui l'ereditarietà è basata su prototipi. Le funzioni vengono utilizzate per creare componenti riutilizzabili. In ECMAScript 6 è stato introdotto un approccio basato sulla classe orientato agli oggetti. TypeScript ha introdotto le classi per sfruttare i vantaggi delle tecniche orientate agli oggetti come l'incapsulamento e l'astrazione. La classe in TypeScript viene compilata in semplici funzioni JavaScript dal compilatore TypeScript per funzionare su piattaforme e browser.

Una classe può includere quanto segue:

- Costruttore
- Proprietà
- Metodi

```
class Employee {
    empCode: number;
    empName: string;

    constructor(code: number, name: string) {
        this.empName = name;
        this.empCode = code;
    }

    getSalary() : number {
        return 10000;
    }
}
```

Classi

Il costruttore è un tipo speciale di metodo che viene chiamato durante la creazione di un oggetto. In TypeScript, il metodo costruttore è sempre definito con il nome *constructor*.

```
class Employee {
    empCode: number;
    empName: string;

    constructor(empcode: number, name: string ) {
        this.empCode = empcode;
        this.name = name;
    }
}
```

Nota: non è necessario che una classe abbia un costruttore.

Classi

Un oggetto della classe può essere creato utilizzando la parola chiave *new*.

```
class Employee {
    empCode: number;
    empName: string;
}

let emp = new Employee();
```

Se la classe include un costruttore parametrizzato, possiamo passare i valori durante la creazione dell'oggetto.

```
class Employee {
    empCode: number;
    empName: string;

    constructor(empcode: number, name: string ) {
        this.empCode = empcode;
        this.name = name;
    }
}

let emp = new Employee(100, "Steve");
```

Classi

Proprio come i linguaggi orientati agli oggetti come Java e C#, le classi TypeScript possono essere estese per creare nuove classi con ereditarietà, utilizzando la parola chiave *extends*.

```
class Person {
    name: string;

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

class Employee extends Person {
    empCode: number;

    constructor(empcode: number, name: string) {
        super(name);
        this.empCode = empcode;
    }

    displayName() : void {
        console.log("Name = " + this.name + ", Employee Code = " + this.empCode);
    }
}

let emp = new Employee(100, "Bill");
emp.displayName(); // Name = Bill, Employee Code = 100
```

Nota: dobbiamo chiamare il metodo `super()` prima di assegnare valori alle proprietà nel costruttore della classe derivata.

Classi

Una classe può implementare interfacce singole o multiple.

```
interface IPerson {
    name: string;
    display(): void;
}
interface IEmployee {
    empCode: number;
}
class Employee implements IPerson, IEmployee {
    empCode: number;
    name: string;

    constructor(empcode: number, name: string) {
        this.empCode = empcode;
        this.name = name;
    }

    display(): void {
        console.log("Name = " + this.name + ", Employee Code = " + this.empCode);
    }
}

let per: IPerson = new Employee(100, "Bill");
per.display(); // Name = Bill, Employee Code = 100

let emp: IEmployee = new Employee(100, "Bill");
emp.display(); //Compiler Error: Property 'display' does not exist on type 'IEmployee'
```

Classi

Un'interfaccia può anche estendere una classe per rappresentare un tipo.

```
class Person {
    name: string;
}

interface IEmployee extends Person {
    empCode: number;
}

let emp: IEmployee = {
    empCode : 1,
    name: "James Bond"
}
```

Classi

Quando una classe figlia definisce la propria implementazione di un metodo dalla classe genitore, si parla di **override** del metodo.

```
class Car {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    run(speed: number = 0) {
        console.log("A " + this.name + " is moving at " + speed + " mph!");
    }
}

class Mercedes extends Car {
    constructor(name: string) {
        super(name);
    }
    run(speed = 150) {
        console.log('A Mercedes started');
        super.run(speed);
    }
}

class Honda extends Car {
    constructor(name: string) {
        super(name);
    }
    run(speed = 100) {
        console.log('A Honda started');
        super.run(speed);
   }
}

let mercObj = new Mercedes("Mercedes-Benz GLA");
let hondaObj = new Honda("Honda City")
mercObj.run(); // A Mercedes started A Mercedes-Benz GLA is moving at 150 mph!
hondaObj.run(); // A Honda started A Honda City is moving at 100 mph!
```

Classi astratte

Si definisce una classe astratta in Typescript usando la parola chiave *`abstract`*. Le classi astratte sono principalmente utilizzate per l'ereditarietà, in cui altre classi possono derivare da esse. Non possiamo creare un'istanza di una classe astratta.
Una classe astratta in genere include uno o più metodi astratti o dichiarazioni di proprietà. La classe che estende la classe astratta deve definire tutti i metodi astratti.
La seguente classe astratta dichiara un metodo astratto *`find`* e include anche un metodo normale *`display`*.

Classi astratte

```
abstract class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    display(): void{
        console.log(this.name);
    }
    abstract find(string): Person;
}

class Employee extends Person {
    empCode: number;
    constructor(name: string, code: number) {
        super(name); // deve chiamare super()
        this.empCode = code;
    }
    find(name: string): Person { // esegue una chiamata al db per trovare l'employee
        return new Employee(name, 1);
    }
}

let emp: Person = new Employee("James", 100);
emp.display(); //James
let emp2: Person = emp.find('Steve');
```

Nota: la classe che implementa una classe astratta deve chiamare il costruttore `super()`.

Modificatori di accesso

Nella programmazione orientata agli oggetti, il concetto di incapsulamento è usato per rendere pubblici o privati i membri di una classe, cioè una classe può controllare la visibilità dei suoi membri di dati. Questo viene fatto usando i modificatori di accesso.

Esistono tre tipi di modificatori di accesso in TypeScript: *public*, *private* e *protected*.

Per impostazione predefinita, tutti i membri di una classe in TypeScript sono *public*. È possibile accedere a tutti i membri pubblici ovunque senza alcuna restrizione.

```
class Employee {
    public empCode: string;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123;
emp.empName = "Swati";
```

Modificatori di accesso

Il modificatore di accesso *private* garantisce che i membri della classe siano visibili solo a quella classe e non siano accessibili al di fuori della classe che li contiene.

```
class Employee {
    private empCode: number;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123; // Compiler Error
emp.empName = "Swati"; //OK
```

Modificatori di accesso

Il modificatore di accesso *protected* è simile al modificatore di accesso *private*, ad eccezione del fatto che è possibile accedere ai membri protetti utilizzando le relative classi di derivazione.

```
class Employee {
    public empName: string;
    protected empCode: number;
    constructor(name: string, code: number){
        this.empName = name;
        this.empCode = code;
    }
}

class SalesEmployee extends Employee{
    private department: string;
    constructor(name: string, code: number, department: string) {
        super(name, code);
        this.department = department;
    }
}

let emp = new SalesEmployee("John Smith", 123, "Sales");
emp.empCode; //Compiler Error
```

readonly

TypeScript include la parola chiave *readonly* che rende una proprietà di sola lettura nella classe, nel tipo o nell'interfaccia.

Il prefisso *readonly* viene utilizzato per rendere una proprietà di sola lettura. È possibile accedere ai membri di sola lettura all'esterno della classe, ma il loro valore non può essere modificato. Poiché i membri di sola lettura non possono essere modificati all'esterno della classe, devono essere inizializzati alla dichiarazione o inizializzati all'interno del costruttore della classe.

```
class Employee {
    readonly empCode: number;
    empName: string;
    constructor(code: number, name: string) {
        this.empCode = code;
        this.empName = name;
    }
}

let emp = new Employee(10, "John");
emp.empCode = 20; //Compiler Error
emp.empName = 'Bill';
```

Nota: anche un'interfaccia può avere proprietà di membro di sola lettura.

readonly

Allo stesso modo puoi usare *Readonly<T>* per creare un tipo di sola lettura, come mostrato di seguito.

```
interface IEmployee {
    empCode: number;
    empName: string;
}

let emp1: Readonly<IEmployee> = {
    empCode: 1,
    empName: "Steve"
}

emp1.empCode = 100; // Compiler Error: Cannot change readonly 'empCode'
emp1.empName = 'Bill'; // Compiler Error: Cannot change readonly 'empName'

let emp2: IEmployee = {
    empCode: 1,
    empName: "Steve"
}

emp2.empCode = 100; // OK
emp2.empName = 'Bill'; // OK
```

static

ES6 include membri statici, così come TypeScript. Si accede ai membri statici di una classe utilizzando il nome della classe e la notazione con punto, senza creare un oggetto, ad esempio <ClassName>.<StaticMember>.
I membri statici possono essere definiti utilizzando la parola chiave *static*. Si consideri il seguente esempio di una classe con proprietà statica.

```
class Circle {
    static pi: number = 3.14;
    static calculateArea(radius: number) {
        return this.pi * radius * radius;
    }
}

Circle.pi; // returns 3.14
Circle.calculateArea(5); // returns 78.5
```

Si consideri ora l'esempio seguente con membri statici e non statici.

```
class Circle {
    static pi = 3.14;
    pi = 3;
}

Circle.pi; // returns 3.14

let circleObj = new Circle();
circleObj.pi; // returns 3
```

Nota: le classi e i costruttori non possono essere statici in TypeScript.

Moduli

Il codice TypeScript che scriviamo è nell'ambito globale per impostazione predefinita. Se abbiamo più file in un progetto, le variabili, le funzioni, ecc. scritte in un file sono accessibili in tutti gli altri file.

Ad esempio, considera i seguenti file TypeScript: file1.ts e file2.ts

file1.ts

```
var greeting: string = "Hello World!";
```

file2.ts

```
console.log(greeting); //Stampa Hello World!
greeting = "Hello TypeScript"; // consentito
```

La precedente variabile `greeting`, dichiarata in file1.ts è accessibile anche in file2.ts. Non solo è accessibile, ma è anche modificabile. Chiunque può facilmente sovrascrivere le variabili dichiarate nell'ambito globale senza nemmeno sapere che lo sta facendo! Questo è uno spazio pericoloso in quanto può portare a conflitti/errori nel codice.

TypeScript fornisce moduli e spazi dei nomi per impedire l'ambito globale predefinito del codice e anche per organizzare e mantenere un'ampia base di codice.

Moduli

I moduli sono un modo per creare un ambito locale nel file. Quindi, tutte le variabili, classi, funzioni, ecc. dichiarate in un modulo non sono accessibili al di fuori del modulo. Un modulo può essere creato utilizzando la parola chiave *export* e può essere utilizzato in un altro modulo utilizzando la parola chiave *import*.

In TypeScript, i file contenenti un'esportazione o un'importazione di primo livello sono considerati moduli. Ad esempio, possiamo creare i file sopra come moduli come di seguito.

file1.ts

```
export var greeting : string = "Hello World!";
```

file2.ts

```
console.log(greeting); //Error: cannot find 'greeting'
greeting = "Hello TypeScript";
```

Moduli

Un modulo può essere definito in un file .ts separato che può contenere funzioni, variabili, interfacce e classi. Usa l'esportazione del prefisso con tutte le definizioni che vuoi includere in un modulo e vuoi accedere da altri moduli.

Employee.ts

```
export let age: number = 20;

export class Employee {
    empCode: number;
    empName: string;

    constructor(name: string, code: number) {
        this.empName = name;
        this.empCode = code;
    }

    displayEmployee() {
        console.log ("Employee Code: " + this.empCode + ", Employee Name: " + this.empName);
    }
}

let companyName: string = "XYZ";
```

Moduli

Un modulo può essere utilizzato in un altro modulo utilizzando un'istruzione *import*.
Abbiamo esportato una variabile e una classe nel file *Employee.ts*. Tuttavia, possiamo solo importare il modulo di esportazione che useremo. Il codice seguente importa solo la classe `Employee` da *Employee.ts* in un altro modulo presente nel file *EmployeeProcessor.ts*.

EmployeeProcessor.ts

```
import { Employee } from "./Employee";

let empObj = new Employee("Steve Jobs", 1);
empObj.displayEmployee(); //Output: Employee Code: 1, Employee Name: Steve Jobs
```

Puoi importare tutte le esportazioni in un modulo come mostrato di seguito.

EmployeeProcessor.ts

```
import * as Emp from "./Employee";

console.log(Emp.age); // 20
let empObj = new Emp.Employee("Bill Gates" , 2);
empObj.displayEmployee(); //Output: Employee Code: 2, Employee Name: Bill Gates
```

È possibile modificare il nome di un'esportazione come mostrato di seguito.

EmployeeProcessor.ts

```
import { Employee as Associate } from "./Employee";

let obj = new Associate("James Bond" , 3);
obj.displayEmployee();//Output: Employee Code: 3, Employee Name: James Bond
```

Namespace

Lo spazio dei nomi viene utilizzato per il raggruppamento logico delle funzionalità. Uno spazio dei nomi può includere interfacce, classi, funzioni e variabili per supportare una singola o un gruppo di funzionalità correlate.

È possibile creare uno spazio dei nomi utilizzando la parola chiave *namespace* seguita dal nome dello spazio dei nomi. Tutte le interfacce, classi ecc. possono essere definite tra parentesi graffe { }.

```
namespace StringUtility {
    function ToCapital(str: string) : string {
        return str.toUpperCase();
    }

    function SubString(str: string, from: number, length: number = 0) : string {
        return str.substr(from, length);
    }
}
```

Namespace

Per impostazione predefinita, i componenti dello spazio dei nomi non possono essere utilizzati in altri moduli o spazi dei nomi. È necessario esportare ciascun componente per renderlo accessibile all'esterno, utilizzando la parola chiave *export* come mostrato di seguito.

StringUtility.ts

```
namespace StringUtility {
    export function ToCapital(str: string) : string {
        return str.toUpperCase();
    }

    export function SubString(str: string, from: number, length: number = 0) : string {
        return str.substr(from, length);
    }
}
```

Usiamo lo spazio dei nomi `StringUtility` nel modulo `Employee`, come mostrato di seguito.

```
/// <reference path="StringUtility.ts" />
export class Employee {
    empCode: number;
    empName: string;
    constructor(name: string, code: number) {
        this.empName = StringUtility.ToCapital(name);
        this.empCode = code;
    }

    displayEmployee() {
        console.log ("Employee Code: " + this.empCode + ", Employee Name: " + this.empName );
    }
}
```

Generics

Quando si scrivono programmi, uno degli aspetti più importanti è costruire componenti riutilizzabili. Ciò garantisce che il programma sia flessibile e scalabile a lungo termine.
I generics offrono un modo per creare componenti riutilizzabili. I generics forniscono un modo per far funzionare i componenti con qualsiasi tipo di dati e non limitarsi a un tipo di dati. Pertanto, i componenti possono essere chiamati o utilizzati con una varietà di tipi di dati. I generics in TypeScript sono quasi simili ai generics C#.

```
function getArray(items: any[]) : any[] {
    return new Array().concat(items);
}

let myNumArr = getArray([100, 200, 300]);
let myStrArr = getArray(["Hello", "World"]);

myNumArr.push(400); // OK
myStrArr.push("Hello TypeScript"); // OK
myNumArr.push("Hi"); // OK
myStrArr.push(500); // OK

console.log(myNumArr); // [100, 200, 300, 400, "Hi"]
console.log(myStrArr); // ["Hello", "World", "Hello TypeScript", 500]
```

Nell'esempio precedente, la funzione `getArray()` accetta un array di tipo *any*. Viene creato un nuovo array di tipo *any*, vengono concatenati gli elementi e restituito il nuovo array. Poiché si è utilizzato il tipo *any* per gli argomenti, è possibile passare qualsiasi tipo di array alla funzione. Tuttavia, questo potrebbe non essere il comportamento desiderato. Potremmo voler aggiungere i numeri all'array di numeri o le stringhe all'array di stringhe ma non i numeri all'array di stringhe o viceversa.

Generics

La funzione precedente può essere riscritta come funzione generica come di seguito.

```
function getArray<T>(items: T[] ) : T[] {
    return new Array<T>().concat(items);
}

let myNumArr = getArray<number>([100, 200, 300]);
let myStrArr = getArray<string>(["Hello", "World"]);

myNumArr.push(400); // OK
myStrArr.push("Hello TypeScript"); // OK
myNumArr.push("Hi"); // Compiler Error
myStrArr.push(500); // Compiler Error
```

La variabile di tipo `T` è specificata con la funzione tra parentesi angolari `getArray<T>`. La variabile di tipo `T` viene utilizzata anche per specificare il tipo degli argomenti e il valore restituito. Ciò significa che il tipo di dati che verrà specificato al momento di una chiamata di funzione sarà anche il tipo di dati degli argomenti e del valore restituito.

Non è consigliato, ma è possibile anche chiamare una funzione generica senza specificare la variabile di tipo. Il compilatore utilizzerà l'inferenza del tipo per impostare il valore di `T` sulla funzione in base al tipo di dati dei valori dell'argomento.

```
let myNumArr = getArray([100, 200, 300]); // OK
let myStrArr = getArray(["Hello", "World"]); // OK
```

I generics possono essere applicati all'argomento della funzione, al tipo restituito di una funzione e ai campi o metodi di una classe.

Generics

Possiamo specificare più variabili di tipo con nomi diversi come mostrato di seguito.

```
function displayType<T, U>(id: T, name: U) : void {
    console.log(typeof(id) + ", " + typeof(name));
}

displayType<number, string>(1, "Steve"); // number, string
```

Quando si utilizzano variabili di tipo per creare componenti generici, TypeScript ci obbliga a utilizzare solo metodi generali disponibili per ogni tipo.

```
function displayType<T, U>(id: T, name: U) : void {

    id.toString(); // OK
    name.toString(); // OK

    id.toFixed(); // Compiler Error: 'toFixed' does not exists on type 'T'
    name.toUpperCase(); // Compiler Error: 'toUpperCase' does not exists on type 'U'

    console.log(typeof(id) + ", " + typeof(name));
}
```

È possibile utilizzare metodi di matrice per la matrice generica.

```
function displayNames<T>(names: T[]) : void {
    console.log(names.join(", "));
}

displayNames<string>(["Steve", "Bill"]); // Steve, Bill
```

Generics

Come accennato in precedenza, il tipo generico consente qualsiasi tipo di dati. Tuttavia, possiamo limitarlo a determinati tipi utilizzando i vincoli. Considera il seguente esempio:

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

    constructor(fname: string, lname: string) {
        this.firstName = fname;
        this.lastName = lname;
    }
}

function display<T extends Person>(per: T) : void {
    console.log(`${per.firstName} ${per.lastName}` );
}

var per = new Person("Bill", "Gates");
display(per); //Output: Bill Gates
display("Bill Gates"); //Compiler Error
```

Nell'esempio precedente, la funzione `display` è una funzione generica con vincoli. Un vincolo viene specificato dopo il tipo generico nelle parentesi angolari. Il vincolo `<T extends Person>` specifica che il tipo generico T deve estendere la classe `Person`. Quindi, la classe `Person` o qualsiasi altra classe che estende la classe `Person` può essere impostata come tipo generico durante la chiamata alla funzione `display`, altrimenti il compilatore darà un errore.

Interfaccia Generics

Il tipo generico può essere utilizzato anche con l'interfaccia. Quella che segue è un'interfaccia generics.

```
interface IProcessor<T> {
    result: T;
    process(a: T, b: T) => T;
}
```

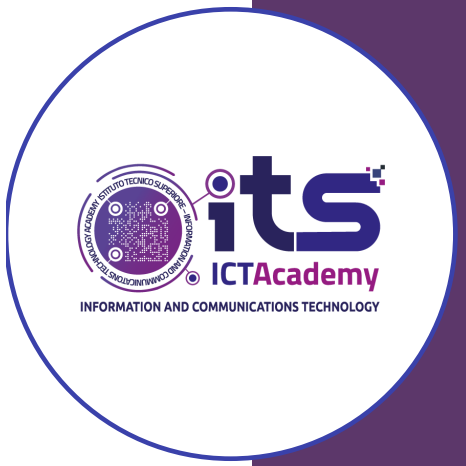
Quanto sopra `IProcessor` è un'interfaccia generics perché abbiamo usato il tipo di variabile `<T>`.
L'interfaccia `IProcessor` include il campo generico `result` e il metodo generico `process()` che accetta due parametri di tipo generico e restituisce un tipo generico.
L'interfaccia generics può essere utilizzata come tipo, come mostrato di seguito.

```
interface KeyPair<T, U> {
    key: T;
    value: U;
}

let kv1: KeyPair<number, string> = { key: 1, value: "Steve" }; // OK
let kv2: KeyPair<number, number> = { key: 1, value: 12345 }; // OK
```

Interfaccia Generics

L'interfaccia generics può essere utilizzata anche come tipo di funzione.

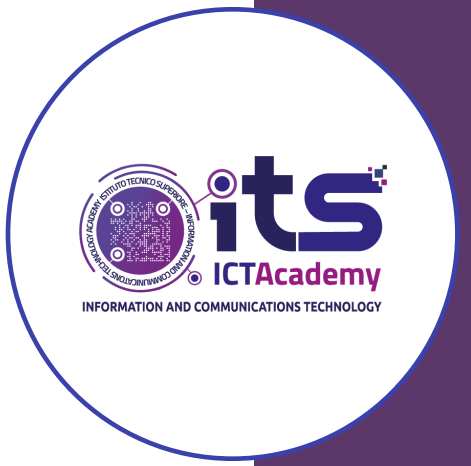
```
interface KeyValueProcessor<T, U> {
    (key: T, val: U) : void;
};

function processKeyPairs<T, U>(key: T, value: U) : void {
    console.log(`processKeyPairs: key = ${key}, value = ${value}`);
}

let numKVProcessor: KeyValueProcessor<number, number> = processKeyPairs;
numKVProcessor(1, 12345); //Output: processKeyPairs: key = 1, value = 12345

let strKVProcessor: KeyValueProcessor<number, string> = processKeyPairs;
strKVProcessor(1, "Bill"); //Output: processKeyPairs: key = 1, value = Bill
```



Interfaccia Generics

L'interfaccia generics può anche essere implementata nella classe, come l'interfaccia non generics, come mostrato di seguito.

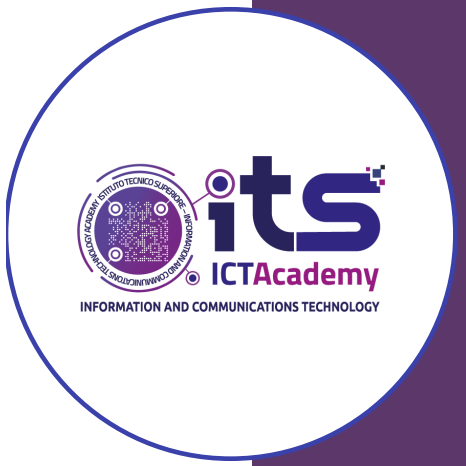
```
interface IKeyValueProcessor<T, U> {
    process(key: T, val: U) : void;
};

class kvProcessor implements IKeyValueProcessor<number, string> {
    process(key: number, val: string) : void {
        console.log(`Key = ${key}, val = ${val}`);
    }
}

let proc: IKeyValueProcessor<number, string> = new kvProcessor();
proc.process(1, 'Bill'); //Output: processKeyPairs: key = 1, value = Bill
```



Classe Generics

TypeScript supporta classi generics. Il parametro di tipo generico è specificato tra parentesi angolari dopo il nome della classe. Una classe generics può avere campi generici (variabili membro) o metodi.

```
class KeyValuePair<T, U> {
    private key: T;
    private val: U;

    setKeyValue(key: T, val: U) : void {
        this.key = key;
        this.val = val;
    }

    display() : void {
        console.log(`Key = ${this.key}, val = ${this.val}`);
    }
}

let kvp1 = new KeyValuePair<number, string>();
kvp1.setKeyValue(1, "Steve");
kvp1.display(); //Output: Key = 1, Val = Steve

let kvp2 = new KayValuePair<string, string>();
kvp2.SetKeyValue("CEO", "Bill");
kvp2.display(); //Output: Key = CEO, Val = Bill
```



Classe Generics

La classe generica può anche implementare un'interfaccia generica. Considera il seguente esempio.

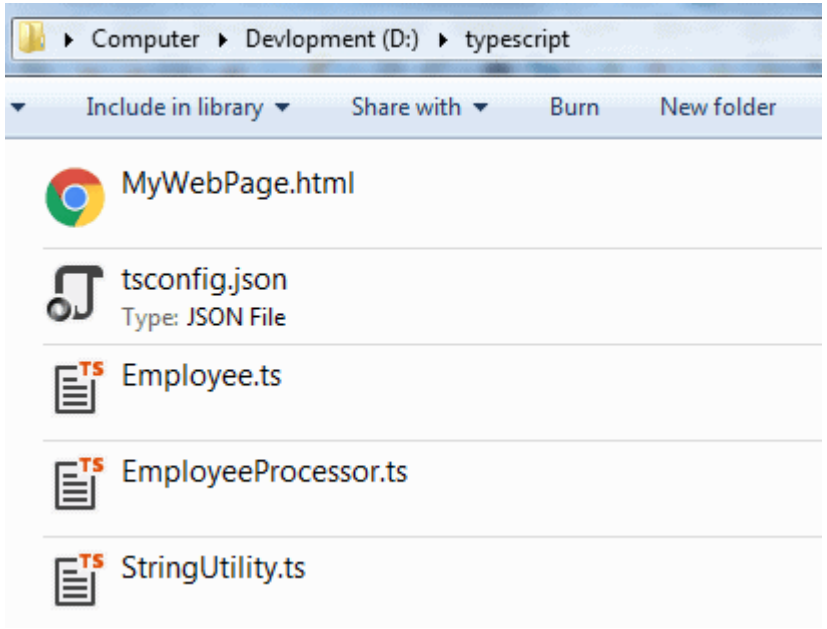
```
interface IKeyValueProcessor<T, U> {
    process(key: T, val: U): void;
};

class kvProcessor<T, U> implements IKeyValueProcessor<T, U> {
    process(key: T, val: U) : void {
        console.log(`Key = ${key}, val = ${val}`);
    }
}

let proc: IKeyValueProcessor<number, string> = new kvProcessor();
proc.process(1, 'Bill'); //Output: key = 1, value = Bill
```

Compilazione

Qui imparerai come compilare un progetto TypeScript e imparerai anche a conoscere *tsconfig.json*.

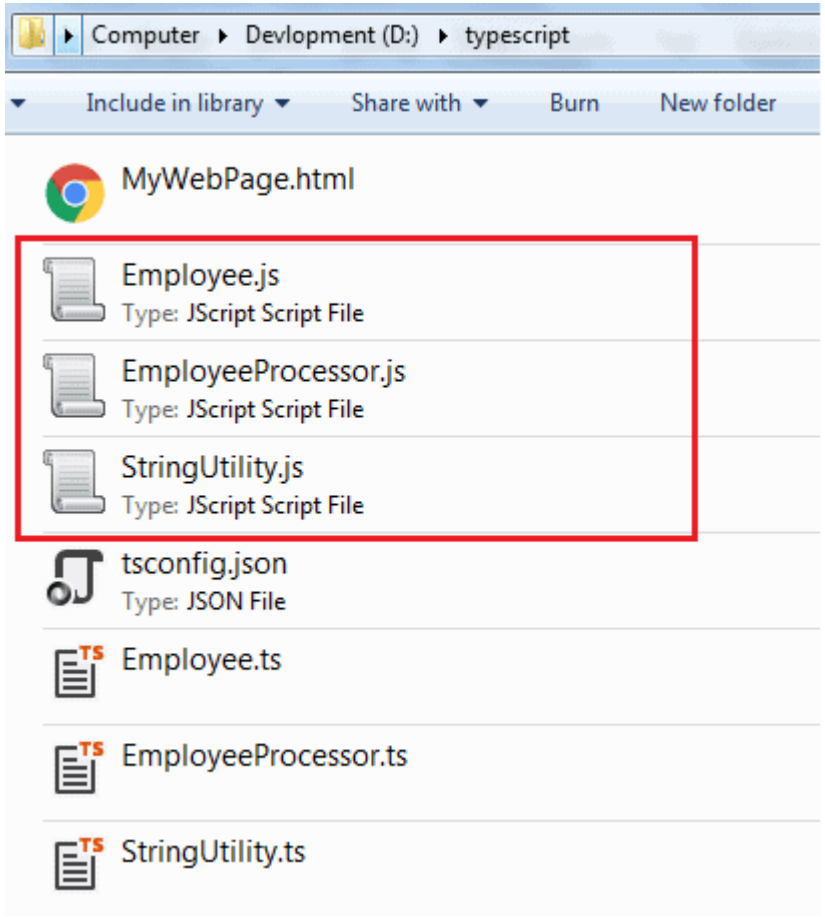
Come sai, i file TypeScript possono essere compilati usando il comando `tsc <file name>.ts`. Sarà noioso compilare più file .ts in un progetto di grandi dimensioni. Quindi, TypeScript fornisce un'altra opzione per compilare tutti o alcuni file .ts del progetto.

TypeScript supporta la compilazione di un intero progetto contemporaneamente includendo il file *tsconfig.json* nella directory radice.
Il file *tsconfig.json* è un semplice file in formato JSON in cui possiamo specificare varie opzioni per dire al compilatore come compilare il progetto corrente.
Considera il seguente semplice progetto che include due file di modulo, un file dello spazio dei nomi, *tsconfig.json* e un file html.



Compilazione

Il file *tsconfig.json* precedente include parentesi graffe vuote { } e non include alcuna opzione. In questo caso, il comando `tsc` considererà i valori predefiniti per le opzioni del compilatore e compilerà tutti i file .ts in una directory radice e nelle sue sottodirectory.
Il comando *tsc* genererà file .js per tutti i file .ts, come mostrato di seguito.



Compilazione

Quando si utilizza il comando `tsc` per compilare i file, se non viene specificato un percorso per *tsconfig.json*, il compilatore cercherà il file nella directory corrente. Se non viene trovato nella directory corrente, cercherà il file *tsconfig.json* nella directory padre. Il compilatore non compilerà un progetto se un file *tsconfig.json* è assente.

Se il file *tsconfig.json* non viene trovato nella directory principale, puoi specificare il percorso utilizzando l' opzione `-project` o `-p`, come mostrato di seguito.

```
tsc -p <percorso di tsconfig.json>
```

È possibile impostare diverse opzioni del compilatore nella proprietà `compilerOptions` nel file *tsconfig.json*, come mostrato di seguito.

```
{
    "compilerOptions": {
        "module": "amd",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "sourceMap": true
    }
}
```

Potete trovare tutte le opzioni disponibile per la sezione `compilerOptions` al seguente indirizzo https://www.typescriptlang.org/docs/handbook/compiler-options.html.
Sono le stesse opzioni del comando `tsc`, che puoi usare durante la compilazione di un file.

Compilazione

È inoltre possibile specificare file specifici da compilare utilizzando l'opzione "file". La proprietà files fornisce un elenco di tutti i file da compilare.

```
{
    "compilerOptions": {
        "module": "amd",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "sourceMap": true
    },
    "files": {
        "Employee.ts"
    }
}
```

Qui, il compilatore compilerà solo il file `Employee.ts`.

Compilazione

Esistono due proprietà aggiuntive che possono essere utilizzate per includere o omettere determinati file: `include` ed `exclude`. Verranno compilati tutti i file specificati in `include`, ad eccezione di quelli specificati nella proprietà `exclude`.

```
{
    "compilerOptions": {
        "module": "amd",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "outFile": "../../built/local/tsc.js",
        "sourceMap": true
    },
    "include": [
         "src/**/*"
    ],
    "exclude": [
        "node_modules",
        "**/*.spec.ts"
    ]
}
```