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

Cos'è Typescript?

TypeScript è un linguaggio open source orientato agli oggetti sviluppato e gestito da Microsoft, concesso in licenza.

TypeScript estende JavaScript aggiungendo tipi di dati, classi e altre funzionalità orientate agli oggetti con il controllo del tipo. È un superset tipizzato di JavaScript che viene compilato in semplice JavaScript.

JavaScript è un linguaggio di programmazione dinamico non tipizzato.

Fornisce tipi primitivi come string, number, object, ecc., ma non controlla i valori assegnati. Le variabili JavaScript vengono dichiarate utilizzando la parola chiave *var* e possono puntare a qualsiasi valore. JavaScript non supporta le classi e altre funzionalità orientate agli oggetti (ECMA2015 lo supporta). Quindi, senza il sistema di tipi, non è facile utilizzare JavaScript per creare applicazioni complesse con team di grandi dimensioni che lavorano sullo stesso codice.

Il sistema di tipi aumenta la qualità del codice, la leggibilità e semplifica la manutenzione e il refactoring dello stesso. Ancora più importante, gli errori possono essere rilevati in fase di compilazione piuttosto che in fase di esecuzione.

Quindi, il motivo per utilizzare TypeScript è che rileva gli errori in fase di compilazione, in modo da poterlo correggere prima di eseguire il codice. Supporta funzionalità di programmazione orientate agli oggetti come tipi di dati, classi, enum e così via, consentendo l'utilizzo di JavaScript su larga scala.

TypeScript si compila in un semplice JavaScript. Il compilatore TypeScript è implementato anche in TypeScript e può essere utilizzato con qualsiasi browser o motore JavaScript come Node.js. TypeScript necessita di un ambiente compatibile con ECMAScript 3 o superiore per la compilazione. Questa è una condizione soddisfatta oggi da tutti i principali browser e motori JavaScript.

Installazione

Esistono tre modi per installare TypeScript:

- Come pacchetto NPM sul tuo computer locale o nel tuo progetto.
- Tramite NuGet nei progetti .NET o .NET Core.
- Come plug-in nel tuo IDE (Integrated Development Environment).

Installazione tramite NPM

NPM (gestore pacchetti Node.js) viene utilizzato per installare il pacchetto TypeScript sul computer locale o su un progetto. È necessario assicurarsi di aver installato Node.js sul proprio computer locale. Se si utilizzano framework JavaScript per le proprie applicazione, si consiglia vivamente di installare Node.js.

Per installare o aggiornare l'ultima versione di TypeScript, aprire il prompt dei comandi/terminale e digitare il seguente comando:

```
npm install -g typescript
```

Il comando precedente installerà TypeScript a livello globale in modo da poterlo utilizzare in qualsiasi progetto. Controllare la versione installata di TypeScript usando il seguente comando:

```
tsc -v
```

Eseguire il comando seguente per installare TypeScript nel progetto locale come dipendenza dev.

```
npm install typescript --save-dev
```

Typescript Playground

TypeScript fornisce un playground online https://www.typescriptlang.org/play per scrivere e testare il codice al volo senza la necessità di scaricare o installare nulla.

Questo è un ottimo posto per i principianti per imparare TypeScript e provare diverse funzionalità di TypeScript. C'è anche la possibilità di condividere il proprio codice tramite un link condivisibile fornito dal playground.

Un primo programma

Creare un nuovo file nel proprio editor di codice, assegnargli il nome `add.ts` e scrivere il codice seguente:

```
function addNumbers(a: number, b: number) {
    return a + b;
}
var sum: number = addNumbers(10, 15);
console.log('La somma dei due numeri è : ' + sum);
```

Il codice TypeScript precedente definisce la funzione `addNumbers()`, la chiama e visualizza il risultato nella console del browser.
Ora aprire il prompt dei comandi su Windows (o un terminale sulla tua piattaforma), andare al percorso in cui si è salvato `add.ts` e compilare il programma usando il seguente comando:

```
tsc add.ts
```

Il comando precedente compilerà il file TypeScript `add.ts` e creerà il file Javascript denominato `add.js` nella stessa posizione. Il file `add.js` contiene il seguente codice.

```
function addNumbers(a, b) {
    return a + b;
}
var sum = addNumbers(10, 15);
console.log('La somma dei due numeri è : ' + sum);
```

Sostituisci `var sum:number = addNumbers(10, 15)` in `add.ts` con `var sum:number = addNumbers(10,'abc')` e ricompila il file `add.ts` nel terminale. Otterrai un errore di compilazione!

Type annotation

TypeScript è un linguaggio tipizzato, in cui possiamo specificare il tipo di variabili, parametri di funzione e proprietà dell'oggetto.

Possiamo specificare il tipo usando `:Type` dopo il nome della variabile, parametro o proprietà. Ci può essere uno spazio dopo i due punti. TypeScript include tutti i tipi primitivi di JavaScript: `number`, `string` e `boolean`.

```
var age: number = 32; // variabile number
var name: string = "John"; // variabile string
var isUpdated: boolean = true; // variabile boolean
```

Non è obbligatorio in TypeScript utilizzare le annotazioni di tipo. Tuttavia, le annotazioni di tipo aiutano il compilatore a controllare i tipi e aiutano a evitare errori durante la gestione dei tipi di dati.

```
function display(id:number, name:string) {
    console.log("Id = " + id + ", Name = " + name);
}

var employee : {
    id: number;
    name: string;
};

employee = {
    id: 100,
    name : "John"
}
```

Variabili

In Typescript le variabili possono essere dichiarate usando: `var`, `let` e `const`.

```
var age: number = 32;
```

```
let employeeName = "John";
// oppure
let employeeName:string = "John";
```

A differenza delle variabili dichiarate con `var`, le variabili dichiarate con `let` hanno un ambito di blocco. Ciò significa che l'ambito delle variabili let è limitato al blocco che le contiene, ad esempio funzione, blocco `if else` o blocco `loop`.

Variabili

Si consideri il seguente esempio:

```
let num1:number = 1;

function letDeclaration() {
    let num2:number = 2;
    if (num2 > num1) {
        let num3: number = 3;
        num3++;
    }
    while(num1 < num2) {
        let num4: number = 4;
        num1++;
    }
    console.log(num1); //OK
    console.log(num2); //OK
    console.log(num3); //Compiler Error: Cannot find name 'num3’
    console.log(num4); //Compiler Error: Cannot find name 'num4’
}

letDeclaration();
```

`num3` è dichiarato nel blocco `if`, quindi il suo ambito è limitato al blocco `if` e non è possibile accedervi al di fuori del blocco `if`. Allo stesso modo, `num4` viene dichiarato nel blocco `while`. Pertanto, durante l'accesso a `num3` e `num4` altrove, verrà visualizzato un errore del compilatore.

Variabili

I vantaggi nell'utilizzo di `let` al posto di var sono:

1. Le variabili `let` con ambito blocco non possono essere lette o scritte prima di essere dichiarate;
2. Le variabili `let` non possono essere nuovamente dichiarate;
3. Le variabili con lo stesso nome e lo stesso tipo possono essere dichiarate in blocchi diversi

```
let num:number = 1;

function demo() {
    let num:number = 2;
    if(true) {
        let num:number = 3;
        console.log(num); //Output: 3
    }
    console.log(num);//Output: 2
}

console.log(num); //Output: 1
demo();
```

Variabili

Le variabili possono essere dichiarate usando `const`. Il `const` rende una variabile una costante in cui il suo valore non può essere modificato. Le variabili `const` hanno le stesse regole di ambito delle variabili `let`.

```
const num:number = 100;
num = 200; //Compiler Error: Cannot assign to 'num' because it is a constant or read-only property
```

Le variabili `const` consentono di modificare le sottoproprietà di un oggetto ma non la struttura dell'oggetto.

```
const playerCodes = {
    player1 : 9,
    player2 : 10,
    player3 : 13,
    player4 : 20
};

playerCodes.player2 = 11; // OK

playerCodes = { //Compiler Error: Cannot assign to playerCodes because it is a constant or read-only
    player1 : 50, // valore modificato
    player2 : 10,
    player3 : 13,
    player4 : 20
};
```

Tipi di dati - number

Come JavaScript, anche TypeScript supporta il tipo di dati numerico. Tutti i numeri vengono memorizzati come numeri in virgola mobile. Questi numeri possono essere decimali (base 10), esadecimali (base 16) o ottali (base 8).

```
let first: number = 123; // number
let second: number = 0x37CF; // hexadecimal
let third: number = 0o377; // octal
let fourth: number = 0b111001; // binary

console.log(first); // 123
console.log(second); // 14287
console.log(third); // 255
console.log(fourth); // 57
```

Tipi di dati - number

Di seguito alcuni dei metodi applicabili su variabili di tipo `number`.

Metodo	Descrizione
toExponential()	Restituisce la notazione esponenziale in formato stringa.
toFixed()	Restituisce la notazione a virgola fissa in formato stringa.
toLocaleString()	Converte il numero in una rappresentazione specifica locale del numero.
toPrecision()	Restituisce la rappresentazione di stringa in esponenziale o a virgola fissa con la precisione specificata.
toString()	Restituisce la rappresentazione di stringa del numero nella base specificata.
valueOf()	Restituisce il valore primitivo del numero.

Tipi di dati - number

Il metodo `toExponential()` restituisce la notazione esponenziale di un numero in formato stringa, in base alle cifre decimali specificate

```
let myNumber: number = 123456;

myNumber.toExponential(); // returns 1.23456e+5
myNumber.toExponential(1); // returns 1.2e+5
myNumber.toExponential(2); // returns 1.23e+5
myNumber.toExponential(3); // returns 1.235e+5
```

Il metodo `toFixed` restituisce la notazione a virgola fissa di un numero in formato stringa.

```
let myNumber: number = 10.8788;

myNumber.toFixed(); // returns 11
myNumber.toFixed(1); //returns 10.9
myNumber.toFixed(2); //returns 10.88
myNumber.toFixed(3); //returns 10.879
myNumber.toFixed(4); //returns 10.8788
```

Tipi di dati - number

Il metodo `toLocaleString()` converte il numero in una rappresentazione specifica locale del numero.

```
let myNumber: number = 10667.987;

myNumber.toLocaleString(); // returns 10,667.987 - US English
myNumber.toLocaleString('de-DE'); // returns 10.667,987 - German
myNumber.toLocaleString('ar-EG'); // returns 10667.987 in Arabico
```

Il metodo `toPrecision` restituisce la rappresentazione di stringa in esponenziale o a virgola fissa alla precisione specificata.

```
let myNumber: number = 10.5679;

myNumber.toPrecision(1); // returns 1e+1
myNumber.toPrecision(2); // returns 11
myNumber.toPrecision(3); // returns 10.6
myNumber.toPrecision(4); // returns 10.57
```

Tipi di dati - number

Il metodo `toString()` restituisce una rappresentazione in forma di stringa del numero nella base specificata.

```
let myNumber: number = 123;

myNumber.toString(); // returns '123’
myNumber.toString(2); // returns '1111011’
myNumber.toString(4); // returns '1323’
myNumber.toString(8); // returns '173’
myNumber.toString(16); // returns '7b’
myNumber.toString(36); // returns '3f'
```

Il metodo `valueOf` restituisce il valore primitivo del numero.

```
let num1 = new Number(123);

console.log(num1) //Output: a number object with value 123
console.log(num1.valueOf()) //Output: 123
console.log(typeof num1) //Output: object

let num2 = num1.valueOf()

console.log(num2) //Output: 123
console.log(typeof num2) //Output: number
```

Tipi di dati - string

`string` è un altro tipo di dati primitivo utilizzato per archiviare dati di testo. I valori stringa sono racchiusi tra virgolette singole o doppie.

```
let employeeName: string = 'John Smith';
// Oppure
let employeeName: string = "John Smith";
```

Dalla versione 1.4 di TypeScript, TypeScript ha incluso il supporto per le stringhe modello ES6. Le stringhe modello vengono utilizzate per incorporare espressioni nelle stringhe.

```
let employeeName: string = "John Smith";
let employeeDept: string = "Finance";

// Pre-ES6
let employeeDesc1: string = employeeName + " works in the " + employeeDept + " department.";

// Post-ES6
let employeeDesc2: string = `${employeeName} works in the ${employeeDept} department.`;

console.log(employeeDesc1); // John Smith works in the Finance department.
console.log(employeeDesc2); // John Smith works in the Finance department.
```

Tipi di dati - string

Di seguito alcuni dei metodi applicabili su variabili di tipo `string`.

Metodo	Descrizione
charAt()	Restituisce il carattere all'indice dato
concat()	Restituisce una combinazione delle due o più stringhe specificate
indexOf()	Restituisce un indice della prima occorrenza della sottostringa specificata da una stringa (-1 se non trovata)
replace()	Sostituisce la sottostringa corrispondente con una nuova sottostringa
split()	Divide la stringa in sottostringhe e restituisce un array
toUpperCase()	Converte tutti i caratteri della stringa in maiuscolo
toLowerCase()	Converte tutti i caratteri della stringa in minuscolo

Tipi di dati - string

Il metodo `charAt()` restituisce un carattere all'indice specificato da una stringa.

```
let str: string = 'Hello TypeScript';

str.charAt(0); // returns 'H'
str.charAt(2); // returns 'l'
"Hello World".charAt(2); // returns 'l'
```

Il metodo `concat()` concatena due o più stringhe specificate.

```
let str1: string = 'Hello';
let str2: string = 'TypeScript';

str1.concat(str2); // returns 'HelloTypeScript'
str1.concat(' ', str2); // returns 'Hello TypeScript'
str1.concat(' Mr. ', 'Bond'); // returns 'Hello Mr. Bond'
```

Tipi di dati - string

Il metodo `indexOf()` restituisce un indice della prima occorrenza della sottostringa specificata da una stringa. L'indice parte da 0. Restituisce -1 se non trovato. Il metodo di ricerca `indexOf()` fa distinzione tra maiuscole e minuscole, quindi 't' e 'T' sono diversi.

```
let str: string = 'TypeScript';

str.indexOf('T'); // returns 0
str.indexOf('p'); // returns 2
str.indexOf('e'); // returns 3
str.indexOf('T', 1); // returns -1
str.indexOf('t', 1); // returns 9
```

Il metodo `replace()` sostituisce la sottostringa corrispondente con la stringa specificata. L'espressione regolare può essere utilizzata anche per la ricerca.

```
let str1: string = 'Hello Javascript';
let str2: string = 'TypeScript';

str1.replace('Java', 'Type'); // returns 'Hello TypeScript’
str1.replace('JavaScript', str2); // returns 'Hello TypeScript’
str1.replace(/Hello/gi, 'Hi'); // returns 'Hi TypeScript'
```

Tipi di dati - string

Il metodo `split()` suddivide una stringa in sottostringhe in base a un carattere separatore specificato e restituisce un array di sottostringhe. Questo metodo accetta due argomenti: una stringa di separazione e un limite facoltativo che specifica il numero di voci da trovare.

```
let str1: string = 'Apple, Banana, Orange';
let str2: string = ',';

str1.split(str2) // returns [ 'Apple', ' Banana', ' Orange' ]
str1.split(',') // returns [ 'Apple', ' Banana', ' Orange' ]
str1.split(',', 2) // returns [ 'Apple', ' Banana' ]
str1.split(',', 1) // returns [ 'Apple']
```

Il metodo `toUpperCase()` restituisce una rappresentazione maiuscola della stringa su cui è chiamato.

```
let str: string = 'Hello Typescript';

str.toUpperCase(); // returns 'HELLO TYPESCRIPT'
'hello typescript'.toUpperCase(); // returns 'HELLO TYPESCRIPT'
```

Il metodo `toLowerCase()` restituisce una rappresentazione minuscola della stringa su cui è chiamato.

```
let str: string = 'Hello Typescript';

str.toLowerCase(); // returns 'hello typescript'
'HELLO TYPESCRIPT'.toLowerCase(); // returns 'hello typescript'
```

Tipi di dati - boolean

I valori booleani sono supportati sia da JavaScript che da TypeScript e archiviati come valori true/false.

Si noti che il booleano `Boolean` è diverso dal tipo minuscolo `boolean`. Il maiuscolo `Boolean` è un tipo di oggetto mentre il minuscolo `boolean` è un tipo primitivo.

Si consiglia di utilizzare il tipo primitivo `boolean` nel codice perché, mentre JavaScript costringe un oggetto al suo tipo primitivo, il sistema di tipi TypeScript non lo fa. TypeScript lo tratta come un tipo di oggetto.

Tipi di dati - Array

Un array è un tipo speciale di tipo di dati che può memorizzare più valori di diversi tipi di dati in sequenza utilizzando una sintassi speciale.

TypeScript supporta gli array, simili a JavaScript. Ci sono due modi per dichiarare un array:

1. Usando parentesi quadre. Questo metodo è simile a come dichiareresti gli array in JavaScript.
2. Utilizzando un tipo di matrice generico, Array<elementType>.

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];
```

```
let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
```

Un array in TypeScript può contenere elementi di diversi tipi di dati utilizzando una sintassi di tipo array generico, come mostrato di seguito.

```
let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
// oppure
let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
```

Tipi di dati - Array

È possibile accedere agli elementi dell'array utilizzando l'indice di un elemento, ad es `ArrayName[index]`. L'indice dell'array parte da zero, quindi l'indice del primo elemento è zero, l'indice del secondo elemento è uno e così via.

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];

fruits[0]; // returns Apple
fruits[1]; // returns Orange
fruits[2]; // returns Banana
fruits[3]; // returns undefined
```

Utilizzare il ciclo for per accedere agli elementi dell'array come mostrato di seguito.

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];

for(var index in fruits) {
    console.log(fruits[index]); // output: Apple Orange Banana
}

for(var i = 0; i < fruits.length; i++) {
    console.log(fruits[i]); // output: Apple Orange Banana
}
```

Tipi di dati - Array

Di seguito i metodi che possono essere utilizzati con gli array

Metodo	Descrizione
pop()	Rimuove l'ultimo elemento dell'array e restituisce quell'elemento
push()	Aggiunge nuovi elementi all'array e restituisce la nuova lunghezza dell'array
sort()	Ordina tutti gli elementi dell'array
concat()	Unisce due matrici e restituisce il risultato combinato
indexOf()	Restituisce l'indice della prima corrispondenza di un valore nell'array (-1 se non trovato)
copyWithin()	Copia una sequenza di elementi all'interno della matrice
fill()	Riempie la matrice con un valore statico dall'indice iniziale fornito all'indice finale
shift()	Rimuove e restituisce il primo elemento dell'array
splice()	Aggiunge o rimuove elementi dall'array
unshift()	Aggiunge uno o più elementi all'inizio della matrice
include()	Controlla se l'array contiene un determinato elemento
join()	Unisce tutti gli elementi dell'array in una stringa
lastIndexOf()	Restituisce l'ultimo indice di un elemento nell'array
slice()	Estrae una sezione dell'array e restituisce il nuovo array
toString()	Restituisce una rappresentazione di stringa della matrice
toLocaleString()	Restituisce una stringa localizzata che rappresenta la matrice

Tipi di dati - Array

Il seguente esempio mostra alcuni dei metodi elencati

```
var fruits: Array<string> = ['Apple', 'Orange', 'Banana'];

fruits.sort();
console.log(fruits); //output: [ 'Apple', 'Banana', 'Orange' ]

console.log(fruits.pop()); //output: Orange

fruits.push('Papaya');
console.log(fruits); //output: ['Apple', 'Banana', 'Papaya']

fruits = fruits.concat(['Fig', 'Mango']);
console.log(fruits); //output: ['Apple', 'Banana', 'Papaya', 'Fig', 'Mango']
console.log(fruits.indexOf('Papaya')); //output: 2
```

Tipi di dati - Tuple

TypeScript ha introdotto un nuovo tipo di dati chiamato Tuple. Tuple può contenere due valori di diversi tipi di dati.

Si consideri il seguente esempio di variabili di tipo numero, stringa e tupla.

```
var empId: number = 1;
var empName: string = "Steve";

// Tuple
var employee: [number, string] = [1, "Steve"];
```

Una variabile di tipo tupla può includere più tipi di dati come mostrato di seguito.

```
var employee: [number, string] = [1, "Steve"];
var person: [number, string, boolean] = [1, "Steve", true];

var user: [number, string, boolean, number, string];// declare tuple variable
user = [1, "Steve", true, 20, "Admin"];// initialize tuple variable
```

Puoi anche dichiarare un array di tuple.

```
var employee: [number, string][];
employee = [[1, "Steve"], [2, "Bill"], [3, "Jeff"]];
```

Tipi di dati - Tuple

Possiamo accedere agli elementi di tupla usando l'indice, allo stesso modo di un array. Un indice parte da zero.

```
var employee: [number, string] = [1, "Steve"];

employee[0]; // returns 1
employee[1]; // returns "Steve"
```

Puoi aggiungere nuovi elementi a una tupla usando il metodo `push()`.

```
var employee: [number, string] = [1, "Steve"];

employee.push(2, "Bill"); console.log(employee); //Output: [1, 'Steve', 2, 'Bill']
```

La tupla è come un array. Quindi, si possono usare metodi di matrice su tupla come `pop()`, `concat()` ecc.

```
var employee: [number, string] = [1, "Steve"];

// recupera il valore per indice ed esegue un'operazione su di esso
employee[1] = employee[1].concat(" Jobs");
console.log(employee); //Output: [1, 'Steve Jobs']
```

Tipi di dati - Enum

enum o enumerazioni sono un nuovo tipo di dati supportato in TypeScript. La maggior parte dei linguaggi orientati ad oggetti come Java e C# utilizza le enumerazioni. Ora è disponibile anche in TypeScript.

In parole semplici, le enum ci consentono di dichiarare un insieme di costanti denominate, ovvero una raccolta di valori correlati che possono essere valori numerici o stringa.

Esistono tre tipi di enumerazione:

1. Enumerazione numerica
2. Enumerazione stringa
3. Enumerazione eterogenea

Gli enum numerici sono enum basati su numeri, ovvero memorizzano i valori di stringa come numeri.
Gli enum possono essere definiti utilizzando la parola chiave `enum`. Supponiamo di voler archiviare un insieme di tipi di supporti di stampa. L'enumerazione corrispondente in TypeScript sarebbe:

```
enum PrintMedia {
    Newspaper,                          Newspaper = 0
    Newsletter,                         Newsletter = 1
    Magazine,                           Magazine = 2
    Book                                Book = 3
}
```

Tipi di dati - Enum

Abbiamo anche la possibilità di inizializzare noi stessi il primo valore numerico. Ad esempio, possiamo scrivere lo stesso enum di:

```
enum PrintMedia {
    Newspaper = 1,
    Newsletter,
    Magazine,
    Book
}
```

Non è necessario assegnare valori sequenziali ai membri Enum. Possono avere qualsiasi valore.

```
enum PrintMedia {
    Newspaper = 1,
    Newsletter = 5,
    Magazine = 5,
    Book = 10
}
```

Tipi di dati - Enum

Le enumerazioni numeriche possono includere membri con valore numerico calcolato. Il valore di un membro enum può essere una costante o calcolato. L'enumerazione seguente include membri con valori calcolati.

```
enum PrintMedia {
    Newspaper = 1,
    Newsletter = getPrintMediaCode('newsletter'),
    Magazine = Newsletter * 3,
    Book = 10
}

function getPrintMediaCode(mediaName: string): number {
    if (mediaName === 'newsletter') {
        return 5;
    }
}

PrintMedia.Newsetter; // returns 5
PrintMedia.Magazine; // returns 15
```

Tipi di dati - Enum

Le enumerazioni stringa sono simili alle enumerazioni numeriche, tranne per il fatto che i valori enum sono inizializzati con valori stringa anziché valori numerici.

I vantaggi dell'utilizzo delle enumerazioni di stringhe è che le enumerazioni di stringhe offrono una migliore leggibilità. Se dovessimo eseguire il debug di un programma, è più facile leggere valori stringa piuttosto che valori numerici.

```
enum PrintMedia {
    Newspaper = "NEWSPAPER",
    Newsletter = "NEWSLETTER",
    Magazine = "MAGAZINE",
    Book = "BOOK"
}

// Access String Enum
PrintMedia.Newspaper; //returns NEWSPAPER
PrintMedia['Magazine'];//returns MAGAZINE
```

Tipi di dati - Enum

Le enumerazioni eterogenee sono enumerazioni che contengono sia valori stringa che numerici.

```
enum Status {
    Active = 'ACTIVE',
    Deactivate = 1,
    Pending
}
```

Enum in TypeScript supporta il mapping inverso. Significa che possiamo accedere al valore di un membro e anche al nome di un membro dal suo valore.

```
enum PrintMedia {
    Newspaper = 1,
    Newsletter,
    Magazine,
    Book
}

PrintMedia.Magazine; // returns 3
PrintMedia["Magazine"];// returns 3
PrintMedia[3]; // returns Magazine
```

Nota: Il mapping inverso non è supportato per i membri stringa enum. Per l'enumerazione eterogenea, il mapping inverso è supportato solo per i membri di tipo numerico ma non per i membri di tipo stringa.

Tipi di dati - Union

TypeScript ci consente di utilizzare più di un tipo di dati per una variabile o un parametro di funzione. Questo è chiamato tipo di unione:

```
let code: (string | number);
code = 123; // OK
code = "ABC"; // OK
code = false; // Compiler Error

let empId: string | number;
empId = 111; // OK
empId = "E111"; // OK
empId = true; // Compiler Error
```

Il parametro della funzione può anche essere di tipo unione, come mostrato di seguito.

```
function displayType(code: (string | number)) {
    if(typeof(code) === "number")
        console.log('Code is number.')
    else if(typeof(code) === "string")
        console.log('Code is string.')
}

displayType(123); // Output: 'Code is number.'
displayType("ABC"); // Output: 'Code is string.'
displayType(true); //Compiler Error: Argument of type 'true' is not assignable to a parameter of type string | number
```

Tipi di dati - any

TypeScript ha il controllo del tipo e controlli in fase di compilazione.

Tuttavia, non sempre abbiamo una conoscenza preliminare del tipo di alcune variabili, soprattutto quando sono presenti valori inseriti dall'utente da librerie di terze parti. In tali casi, abbiamo bisogno di una disposizione che possa occuparsi di contenuto dinamico. Il tipo any è utile qui.

```
let something: any = "Hello World!";

something = 23;
something = true;
```

Allo stesso modo, puoi creare un array di tipo `any[]` se non sei sicuro dei tipi di valori che possono contenere questo array.

```
let arr: any[] = ["John", 212, true];

arr.push("Smith");
console.log(arr); //Output: [ 'John', 212, true, 'Smith' ]
```

Tipi di dati - void

Simile a linguaggi come Java, `void` viene utilizzato dove non ci sono dati. Ad esempio, se una funzione non restituisce alcun valore, è possibile specificare `void` come tipo restituito.

```
function sayHi(): void {
    console.log('Hi!');
}

let speech: void = sayHi();
console.log(speech); //Output: undefined
```

Non ha senso assegnare `void` a una variabile, poiché solo `null` o `undefined` è assegnabile a `void`.

```
let nothing: void = undefined; // OK
let num: void = 1; // Error
```

Tipi di dati - never

TypeScript ha introdotto un nuovo tipo never, che indica i valori che non si verificheranno mai. Il tipo never viene utilizzato quando sei sicuro che qualcosa non accadrà mai. Ad esempio, scrivi una funzione che non tornerà al suo punto finale o genera sempre un'eccezione.

```
function throwError(errorMsg: string): never {
    throw new Error(errorMsg);
}

function keepProcessing(): never {
    while (true) {
        console.log('Farò sempre qualcosa e non finirò mai.');
    }
}
```

Il tipo `void` può avere `undefined` o `null` come valore mentre never non può avere alcun valore.

```
let something: void = null;
let nothing: never = null; // Error: Type 'null' is not assignable to type 'never'
```

In TypeScript, una funzione che non restituisce un valore (`void`), in realtà restituisce `undefined`. Considera il seguente esempio.

```
function sayHi(): void {
    console.log('Hi!');
}

let speech: void = sayHi();
console.log(speech); // undefined
```

Inferenza di tipo

TypeScript è un linguaggio tipizzato. Tuttavia, non è obbligatorio specificare il tipo di una variabile. TypeScript deduce tipi di variabili quando non sono disponibili informazioni esplicite sotto forma di annotazioni di tipo.

I tipi vengono dedotti dal compilatore TypeScript quando:

- Le variabili vengono inizializzate
- I valori predefiniti sono impostati per i parametri
- Vengono determinati i tipi restituiti dalla funzione

Per esempio,

```
var a = "some text";
```

Qui, poiché non stiamo definendo in modo esplicito `a: string` con un'annotazione di tipo, TypeScript deduce il tipo della variabile in base al valore assegnato alla variabile. Il valore di a è una stringa e quindi il tipo di a viene dedotto come stringa.

```
var a = "some text";
var b = 123;
a = b; // Compiler Error: Type 'number' is not assignable to type 'string'
```

if else

Un'istruzione `if` può includere una o più espressioni che restituiscono un valore booleano. Se l'espressione booleana restituisce `true`, viene eseguito un insieme di istruzioni.

```
if (true) {
    console.log('Questo verrà sempre eseguito.');
}

if (false) {
    console.log('Questo non verrà mai eseguito.');
}
```

L'esempio seguente include più espressioni booleane nella condizione `if`.

```
let x: number = 10, y = 20;

if (x < y) {
    console.log('x è minore di y');
}
```

if else

Una condizione `if else` include due blocchi: un blocco `if` e un blocco `else`. Se la condizione `if` restituisce `true`, il blocco `if` viene eseguito. In caso contrario, il blocco `else` viene eseguito.

```
let let x: number = 10, y = 20;

if (x > y) {
    console.log('x è maggiore di y.');
} else {
    console.log('x è uguale o minore a y.'); //This will be executed
}
```

L'istruzione `else if` può essere utilizzata dopo l'istruzione `if`.

```
let x: number = 10, y = 20;
if (x > y) {
    console.log('x è maggiore di y.');
} else if (x < y) {
    console.log('x è minore di y.'); //This will be executed
} else if (x == y) {
    console.log('x è uguale a y.');
}
```

Un operatore ternario è indicato da '?' ed è usato come scorciatoia per un'istruzione if..else. Verifica la presenza di una condizione booleana ed esegue una delle due istruzioni, a seconda del risultato della condizione booleana.

```
let x: number = 10, y = 20;
x > y? console.log('x è maggiore di y.') : console.log('x è minore o uguale a y.');
```

switch

L'istruzione `switch` viene utilizzata per verificare la presenza di più valori ed esegue insiemi di istruzioni per ciascuno di tali valori. Un'istruzione `switch` ha un blocco di codice corrispondente a ciascun valore e può avere un numero qualsiasi di tali blocchi. Quando viene trovata la corrispondenza con un valore, viene eseguito il blocco di codice corrispondente.

Le seguenti regole vengono applicate all'istruzione switch:

1. L'istruzione `switch` può includere un'espressione costante o variabile che può restituire un valore di qualsiasi tipo di dati.
2. All'interno di uno `switch` può esserci un numero qualsiasi di istruzioni `case`. Il `case` può includere una costante o un'espressione.
3. Dobbiamo usare la parola chiave `break` alla fine di ogni blocco `case` per interrompere l'esecuzione del blocco `case`.
4. Il tipo restituito dell'espressione `switch` e dell'espressione `case` deve corrispondere.
5. Il blocco `default` è facoltativo.

switch

```
let day: number = 4;

switch (day) {
    case 0:
        console.log("E’ domenica.");
        break;
    case 1:
        console.log("E’ lunedì.");
        break;
    case 2:
        console.log("E’ martedì.");
        break;
    case 3:
        console.log("E’ mercoledì.");
        break;
    case 4:
        console.log("E’ giovedì.");
        break;
    case 5:
         console.log("E’ venerdì.");
         break;
    case 6:
         console.log("E’ sabato.");
         break;
    default:
         console.log("Giorno inesistente!");
         break;
}
```

switch

L'istruzione switch può anche includere un'espressione come mostrato di seguito.

```
let x = 10, y = 5;

switch (x-y) {
    case 0:
        console.log("Result: 0");
        break;
    case 5:
        console.log("Result: 5");
        break;
    case 10:
        console.log("Result: 10");
        break;
}
```

Ciclo for

TypeScript supporta i seguenti cicli for:

- ciclo `for`
- ciclo `for..of`
- ciclo `for..in`

Il ciclo `for` viene utilizzato per eseguire un blocco di codice un determinato numero di volte, specificato da una condizione. Qui, la prima espressione viene eseguita prima dell'inizio del ciclo. La seconda espressione è la condizione per l'esecuzione del ciclo. E la terza espressione viene eseguita dopo l'esecuzione di ogni blocco di codice.

```
for (let i = 0; i < 3; i++) {
    console.log ("Blocco istruzione esecuzione n." + i);
}
```

Il risultato dell'esecuzione di questo codice è:

```
Blocco istruzione esecuzione n.0
Blocco istruzione esecuzione n.1
Blocco istruzione esecuzione n.2
```

Ciclo for

TypeScript include il ciclo `for...of` per iterare e accedere agli elementi di una matrice, un elenco o una raccolta di tuple. Il ciclo `for...of` restituisce elementi da una raccolta, ad esempio un array, una lista o una tupla, quindi non è necessario utilizzare il tradizionale ciclo `for` mostrato sopra.

```
let arr = [10, 20, 30, 40];

for (var val of arr) {
    console.log(val); // prints values: 10, 20, 30, 40
}
```

Il ciclo `for...of` può anche restituire un carattere da un valore stringa.

```
let str = "Hello World";

for (var char of str) {
    console.log(char); // prints chars: H e l l o W o r l d
}
```

Ciclo for

Un'altra forma del ciclo for è `for...in`. Questo può essere utilizzato con un array, un elenco o una tupla.
Il ciclo `for...in` scorre un elenco o una raccolta e restituisce un **indice** a ogni iterazione.

```
let arr = [10, 20, 30, 40];

for (var index in arr) {
    console.log(index); // prints indexes: 0, 1, 2, 3

    console.log(arr[index]); // prints elements: 10, 20, 30, 40
}
```

Puoi anche usare `let` al posto di `var`. La differenza è che la variabile dichiarata utilizzando `let` non sarà accessibile al di fuori del ciclo for..in .

Ciclo while

Il ciclo `while` è un altro tipo di ciclo che controlla una condizione specificata prima di iniziare a eseguire il blocco di istruzioni. Il ciclo viene eseguito finché non viene soddisfatto il valore della condizione

```
let i: number = 2;

while (i < 4) {
    console.log("Blocco istruzione esecuzione n." + i );
    i++;
}
```

L'esecuzione di questo codice produce il seguente risultato:

```
Blocco istruzione esecuzione n.2
Blocco istruzione esecuzione n.3
```

Ciclo while

Il ciclo `do..while` è simile al ciclo `while`, tranne per il fatto che la condizione è data alla fine del ciclo.
Il ciclo `do..while` esegue il blocco di codice almeno una volta prima di verificare la condizione specificata. Per il resto delle iterazioni, esegue il blocco di codice solo se la condizione specificata è soddisfatta.

```
let i: number = 4;

do {
    console.log("Blocco istruzione esecuzione n." + i );
    i++;
} while ( i < 4)
```

L'esecuzione di questo codice produce il seguente risultato:

```
Blocco istruzione esecuzione n.4
```