

Laboratorio I

a.a. 2026/2027

Introduzione a Typescript & tipi di dato

Contenuti

- Breve storia di JavaScript e Typescript
- Tipi di dato e operatori
- Variabili
- Primi esempi e primi esercizi



JavaScript

- Introdotto nel 1995 per Netscape (browser)
- Web programming, supportato da tutti i browser
- Inizialmente client-side
- Adesso utilizzato sia client che server-side
 - Node.js - piattaforma per sviluppo e esecuzione di programmi in JavaScript
- Vari update negli ultimi 25 anni
- ECMAScript 2026 - ultima versione





JavaScript

- Linguaggio interpretato
 - Interprete: browser, Node.js
- Flessibile, con poche restrizioni
 - **weakly-typed**
 - Svantaggio: difficile trovare gli errori
 - Vantaggio: facile sviluppare soluzioni inedite





JavaScript

JavaScript è

- **linguaggio tipizzato**, ma che
- **non prevede controlli statici** sui tipi di dato ed
- **effettua una conversione implicita tra tipi** (*dynamic type-checking*)



JavaScript

JavaScript è

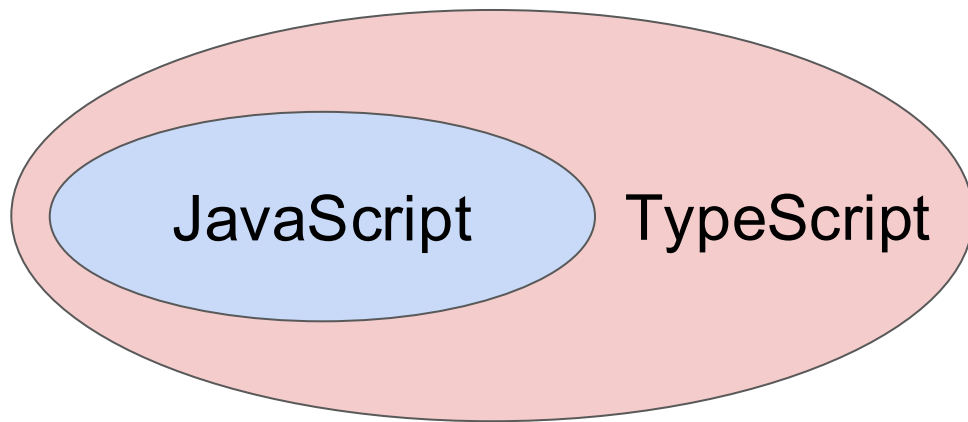
- **linguaggio tipizzato**, ma che
- **non prevede controlli statici** sui tipi di dato ed
- **effettua una conversione implicita tra tipi** (*dynamic type-checking*)



- Per questo JS risulta molto **flessibile** (probabilmente una delle chiavi del suo successo)
- L'assenza di controlli sui tipi durante la *compilazione* può generare **errori difficili** da analizzare a runtime

TypeScript

- Per questo, Microsoft ha definito nel 2012 una **estensione** di JavaScript
- Supporto per il controllo statico dei tipi e altre funzionalità pensate per scrittura di applicazioni complesse



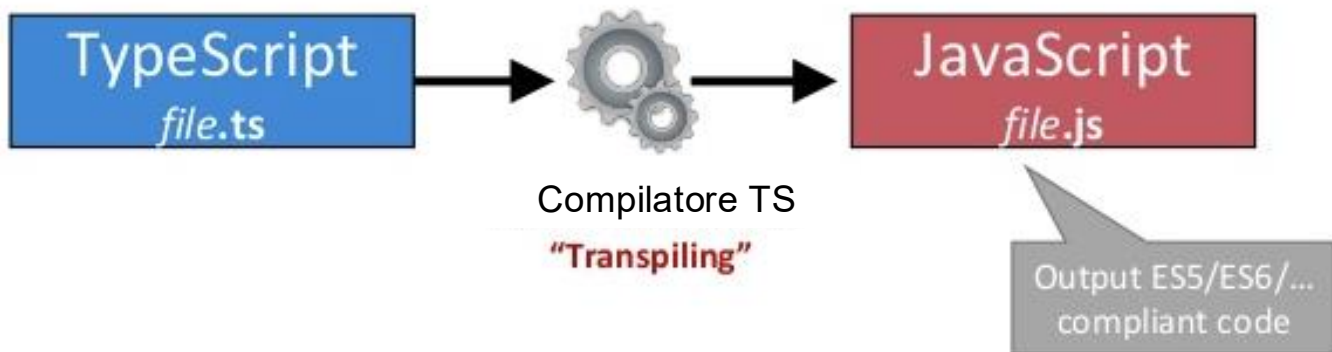
TypeScript



- Introdotto nel 2012 da Microsoft
- “Javascript with syntax for types”
 - Transpiler: traduce il codice in JS
 - Type inference e controlli ulteriori
- Maggiori restrizioni
 - **strongly-typed**
 - Vantaggio: più facile trovare errori

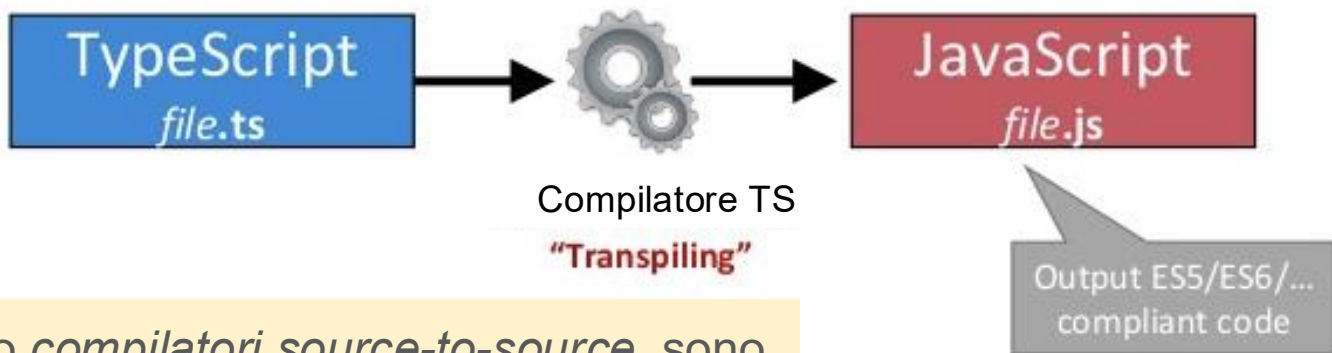
Da JS a TS

- Un'applicazione TypeScript viene tradotta da un "compilatore" (**transpiler**) in un'applicazione JavaScript standard eseguibile su qualsiasi engine



Da JS a TS

- Un'applicazione TypeScript viene tradotta da un "compilatore" (**transpiler**) in un'applicazione JavaScript standard eseguibile su qualsiasi engine



Transpilers, o *compilatori source-to-source*, sono strumenti che leggono codice sorgente in un linguaggio di programmazione e producono codice equivalente in un altro linguaggio che ha un simile livello di astrazione

Struttura di un programma

- Programma

- Implementazione di un algoritmo per risolvere un problema
- Una serie di **comandi** (*statement*) da eseguire in ordine
- Comandi:
 - devono seguire delle regole di **sintassi** - e.g. finiscono con ‘;’
 - automatic semicolon insertion - inserimento automatico del ;
 - vengono interpretati dall'interprete ed eseguiti - hanno una **semantica**
- Esempi di comandi:
 - Calcolo della somma di due numeri
 - Mostrare a video un testo - `console.log`
 - Memorizzare un valore in una **variabile**

Input and output

- Un programma produce la soluzione a un problema
- Il problema può avere dei parametri - dati iniziali su cui lavorare
- INPUT
- Utile passare questi parametri dall'esterno - non serve cambiare il codice per ogni istanza di problema
 - E.g. `prompt()` - prende input da tastiera ← **non disponibile su tutti i sistemi**
- OUTPUT - risultato del programma va mostrato o salvato in qualche modo
 - E.g. `console.log()` - mostra a video nella console

Input and output: esempio

- Scriviamo un programma che risolve un'equazione di primo grado
 - $a*x+b=c$; $x=?$
 - a , b , c sono l'input del programma
 - x è l'output

Tipi di dato

- I comandi fanno delle operazioni con dati in memoria
- I dati possono essere di tipo diverso, ad esempio:
 - Numeri
 - Valori logici (Boolean)
 - Testo
- Le operazioni si fanno usando operatori con i dati
 - ***Espressioni*** - hanno un valore finale
 - Operatore `typeof` - per trovare il tipo di dato (base)

Espressioni vs comandi

- Espressioni

- Hanno un valore finale
- Possono essere usate per comporre altre espressioni, o liste di espressioni
- $5+4$, $x*2$, $a \geq b$

- Comandi

- Espressioni
- Altri comandi che non hanno un valore finale
 - Non possono essere usate per comporre espressioni
 - Hanno un effetto collaterale (di solito)
 - $b=5$, `console.log(5)`

Tipi di dato (primitivi)

- Numeri (`number`)
 - Interi: `2`, `-5`, `0`
 - Reali: `2.5`, `8.3`, `-9867.7`
 - Numeri speciali:
 - `NaN`, `Infinity`, `-Infinity`
 - Operatori
 - `+`, `-`, `*`, `/`, `%` (`a%b` = resto divisione intera tra `a` e `b`)
 - `>`, `<`, `>=`, `<=`, `==`

Tipi di dato (primitivi)

- Valori logici (`boolean`)
 - `true`, `false`
 - Valore di verità di espressioni (e.g. confronti tra valori)
 - Operatori:
 - AND: `&&`, OR: `||`, NOT: `!`
- Esempi: tabella di verità di una funzione Booleana

Tipi di dato (primitivi)

- Testo (`string`)

- Letterali delimitati da `' ' o " " o ` ``

- Esempi:

- `'Siamo al corso di laboratorio',`
`'6', '87.6'`

- `"Ciao", "0", "1425.6"`

Tipi di dato (primitivi)

- Testo (`string`)
 - ``Hello world``
 - Espressioni all'interno della stringa (*template literal*): ``2+5 = ${2+5}``

Template literals: stringhe delimitate da *backticks* (```), che permettono l'inclusione e la valutazione di espressioni (mediante *sostituzioni*)

Tipi di dato (primitivi)

- Testo (`string`)

- Operatori

- + concatenazione

- >, <, ==, <=, >=

- Regole complesse per gestire questi operatori tra stringhe e numeri

- <https://262.ecma-international.org/5.1/#sec-11.8.5>

- <https://262.ecma-international.org/5.1/#sec-9.3>

Variabili (variables, bindings)

- Le espressioni vengono valutate una per una
- Il valore di un'espressione può essere assegnato a un nome - usando `let` o `var` o `const`

Variabili (variables, bindings)

- Le espressioni vengono valutate una per una
- Il valore di un'espressione può essere assegnato a un nome - usando `let` o `var` o `const`

Variabili (variables, bindings)

- Le espressioni vengono valutate una per una
- Il valore di un'espressione può essere assegnato a un nome - usando `let` o `var` o `const`
- Valori `undefined` e `null`

Variabili (variables, bindings)

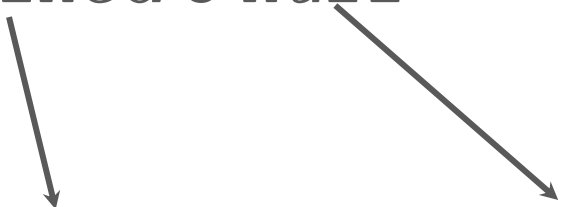
- Le espressioni vengono valutate una per una
- Il valore di un'espressione può essere assegnato a un nome - usando `let` o `var` o `const`
- Valori `undefined` e `null`



Una variabile a cui non è stato assegnato un valore è di tipo **undefined**

Variabili (variables, bindings)

- Le espressioni vengono valutate una per una
- Il valore di un'espressione può essere assegnato a un nome - usando `let` o `var` o `const`
- Valori `undefined` e `null`



Una variabile a cui non è stato assegnato un valore è di tipo **undefined**

Il valore **null** rappresenta l'assenza intenzionale di un valore per un oggetto.

Sintassi TypeScript

- Per **dichiarare** ed **inizializzare** una variabile in TS:

```
let nomeVariabile : tipoVariabile = valoreIniziale
```

- Esempi:
 - `let x:number = 5`
 - `let y,z:number`
 - `let s:string = "ciao"`

Conversione di tipo

- È possibile trasformare una variabile / un valore di un tipo in un altro tipo (**casting**)
 - `4` $\rightarrow$ `'4'`
 - `"65.5"` $\rightarrow$ `65.5`
- `Number()`, `String()`, `Boolean()`
- Attenzione: il cast restituisce sempre un valore (che può essere NaN)

Approfondimento: Inferenza di tipo

- TS applica una strategia di **type inference** quando non viene utilizzata alcuna annotazione esplicita sul tipo utilizzato

```
let x = 3;  
//   ^ = let x: number
```

Attenzione: per i nostri scopi, dovrete sempre specificare **TUTTI** i tipi!

- L'inferenza, se necessaria, avviene ad esempio quando si *inizializzano variabili* o quando si determinano *i valori di ritorno di una funzione*
- Molto spesso l'inferenza è “ovvia”; in altri casi lo è meno, e TS utilizza svariate regole

Operatori di uguaglianza

- `==` e `!=`
 - Uguaglianza e disuguaglianza (del valore)
 - Fa anche **type coercion**: due valori sono confrontati solo dopo conversione a un tipo comune
 - Attenzione, in TS genera errori tranne per `undefined`/`null`
- `===` e `!==`
 - Uguaglianza e disuguaglianza **stretta** (di *valore* **e** *tipo*)

Null vs undefined vs NaN

- `null == null // true`
- `undefined == undefined // true`
- `null == undefined // true`

- `NaN == null // false`
- `NaN == undefined // false`
- `NaN == NaN // false (!!!!)`

Confrontando `null` con qualsiasi altro valore, si otterrà sempre `false`

MA:

- `null === undefined // false (!!!!)`

BigInt e tipi di dato non primitivi

- **bigint:**

- Numeri di lunghezza arbitraria, $-(10^{53}-1) \leq n \leq 10^{53}-1$
- `let a:bigint = 638774856...5645863423n;`

- Tipi di dato non primitivi

- Oggetti (dizionari): `{nome: "Toto", cognome:"Bruno", eta:23}`
- Array: `[3, 6, 2, 8, 9]`
- Funzioni: `x=> { if (x>0) return x; else return -x }`

BigInt e tipi di dato non primitivi

- **bigint:**

- Numeri di lunghezza arbitraria, $-(10^{53}-1) \leq n \leq 10^{53}-1$
- `let a:bigint = 638774856...5645863423n;`

- **Tipi di dato non primitivi**

- Oggetti: `{nome:"Mario", eta:23}`
- Array: `[1,2,3]`
- Funzioni: `x => { if (x > 0) return x; else return -x }`



More on this later

Recap: Tipi di dato primitivi e valori letterali in JS/TS

JavaScript consente di denotare **valori letterali** di determinati tipi.

I seguenti sono i corrispondenti tipi in TypeScript:

- `boolean`: `true`, `false`
- `undefined`: `undefined` ($\rightarrow$ **non è stato definito** nessun valore)
- `null`: `null` ($\rightarrow$ so che **non c'è** un valore)
- `number`: `1`, `5`, `-53.38`, `12.3e4` (numeri “normali”, $-(10^{53}-1) \leq n \leq 10^{53}-1$)
- `bigint`: `90071992547434344169871610992n` (interi a precisione infinita)
- `string`: `“ciao ugo”`, `“हिन्दू”`, `‘hello’`, ``x vale ${x}`` (sequenze di caratteri)

Tipi di dato non primitivi

Ci sono poi tipi di dato non primitivi, che in JavaScript ricadono sotto “Object”, e di cui vedremo più avanti la tipizzazione in TypeScript:

- **Object:** `{ nome: “Ugo”, età: 19 }` (mappe da chiavi a valori - anche dizionari)
 - **Array:** `[ 1, 5, 8, 12, 21, 33 ], [ 1, “ciao!”, { x: 12, y: 22 }, [ “banana” ] ]` (liste con indici numerici)
 - **Function:** `x => 2 * x, (a, b) => a + b, x => { if (x > 0) return x; else return -x }` (funzioni)

Esercizi

1. Scrivere un programma che stampa la [tabella di verità](#) per la funzione Booleana 'OR'
2. Scrivere un programma che, data una temperatura in Celsius, calcola e mostra a video la temperatura in Fahrenheit ($F = C \times 1.8 + 32$)
3. Scrivere un programma che, dato un numero di secondi, calcola e mostra a video il numero di ore, minuti e secondi inclusi.

Q & A

Approfondimento **JS**: Conversione automatica di tipo

- JavaScript permette di fare delle operazioni con dati di tipo diverso
- I valori vengono automaticamente convertiti allo stesso tipo
 - Regole non molto semplici

Approfondimento **JS**: Conversione automatica di tipo

```
let a='5', b=6;  
let c=a+b;  
console.log(c);
```

56

```
let a='5', b=6;  
let c=a-b;  
console.log(c);
```

-1

Approfondimento **JS**: Conversione automatica di tipo

Type coercion: due valori sono confrontati solo dopo conversione a un tipo comune

```
let a='5', b=5;  
console.log(a==b);
```

true

```
let a='5', b=5;  
console.log(a!==b);
```

true

```
let a='5', b=5;  
console.log(a===b);
```

false

Approfondimento **JS**: Conversione automatica di tipo

Per approfondimenti sui casi più complessi di *type coercion*, potete consultare le seguenti risorse:

- in generale: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#type_coercion
- sul trattamento dei numeri: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number#number_coercion
- sul trattamento delle stringhe: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#string_coercion
- sul trattamento dei booleani: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean#boolean_coercion