



TYPESCRIPT: INFERENZA E TIPI COMPOSTI

Giovedì 23 Ottobre



RIPASSO: I TIPI SEMPLICI

tipi primitivi	string, number, boolean, null, undefined, any, unknown, never, void
type annotations	<code>const</code> uname: string = "marco";
array e tuple	<code>const</code> nums: number[] = [1, 2, 3]; <code>const</code> xy: [number, number] = [1, 5];
funzioni	<code>function</code> exec(fn: () => string): string { return fn(); }
oggetti	<code>const</code> book: { title: string; author: string } = { title: "1984", author: "George Orwell", };

type alias	<code>type</code> Animal = { species: string; };
union e intersection	<code>type</code> Dirs = "N" "S" "O" "W"; <code>type</code> LandA = Animal & {legs: number};
interfacce	<code>interface</code> Person { name: string; surname: string; }
classi	<code>class</code> Programmer <code>implements</code> Person { name: string; surname: string; constructor(n: string, s: string) { ... } }

TYPE INFERENCE

Per i valori mutabili, il tipo viene esteso dal literal al tipo base (type widening)

```
let personName = "Marco";
```

personName: string

```
const fib = [1, 1, 2, 3, 5, 8];
```

fib: number[]

```
const profile = {  
  name: "Alice",  
  age: 30,  
  isEmployed: true  
};
```

profile: {
 name: string;
 age: number;
 isEmployed: boolean;
}

```
function printValue(value) {  
  console.log(value);  
}
```

value: any

IL TIPO ANY



any disabilita il type checking!

```
let anyValue: any;  
console.log(anyValue.toUpperCase());  
console.log(anyValue.toFixed(2));
```



implicit any

```
function printValue(value) {  
  console.log(value);  
}
```



solo se nel `tsconfig.json` c'è

`noImplicitAny: true`

oppure

`strict: true`

explicit any

```
function printValue(value: any) {  
  console.log(value);  
}
```



anche nel caso siano
disabilitati gli implicit any!

TYPE INFERENCE VS TYPE ANNOTATION

type inference

- Costanti

```
const size = "XL";
```



- Variabili inizializzate

```
let age = 42;
```

- Array/tuple omogenei

```
let seq = [1, 2];
```

- Callback

```
sequence.forEach((n) => n + 1);
```

- Tipo di ritorno delle funzioni (se evidente)

```
function inc(a: number) {  
  return a + 1;  
}
```

type annotation

- Parametri di funzioni standalone

```
const uc = (str: string) => str.toUpperCase();
```

- Array vuoti

- Riferimenti a super-tipi

```
const s: Shape[] = [new Circle(), new Square()];
```

- Funzioni o oggetti complessi

STRUCTURAL TYPING

```
p1 : {  
  x: number;  
  y: number;  
}
```

```
p1 : {  
  x: number;  
  y: number;  
  z: number;  
}
```

Property 'x'
is missing

```
type Point = {  
  x: number;  
  y: number;  
};  
  
function draw(point: Point) {  
  console.log(`Drawing at (${point.x}, ${point.y})`);  
}  
  
const p1 = { x: 10, y: 20 };  
draw(p1);  
  
const p2 = { x: 10, y: 20, z: 30 };  
draw(p2);  
  
const p3 = { y: 20, z: 30 };  
draw(p3);
```

**Un tipo x è compatibile
con un tipo y se y ha
almeno le stesse
proprietà di x**

STRUCTURAL TYPING

```
type Point = {  
  x: number;  
  y: number;  
};  
  
const source = { x: 10, y: 20, z: 30 };  
const p1: Point = source; ✓  
  
const p2: Point = { x: 10, y: 20, z: 30 }; ✗
```

Object literal may only specify known properties, and 'z' does not exist in type 'Point'

**Gli object literals possono
specificare solo proprietà esistenti
(excess property check)**

TYPE NARROWING

```
function setBgColor(color: string | number) {  
  document.body.style.backgroundColor = color;  
}
```

Type 'string | number'
is not assignable to
type 'string'

```
function setBgColor(color: string | number) {  
  const style = document.body.style;  
  if (typeof color === "string") {  
    style.backgroundColor = color;  
    console.log("Color set to string value:", color);  
  } else {  
    style.backgroundColor = `#${color.toString(16)}`;  
    console.log("Color set to numeric value:", color);  
  }  
}
```

type
guard

all'interno di questo
blocco il tipo di color è
string

all'interno di questo
blocco il tipo di color è
number

TYPE NARROWING: TYPE GUARDS

typeof

```
if (typeof users === "string") ...
```

in

```
if ("vatCode" in user) ...
```

instanceof

```
if (animal instanceof Cat) ...
```

truthiness

```
if (optionalValue) ...
```

equality

```
if (optionalValue !== null) ...
```

- string
- number
- boolean
- bigint
- symbol
- function
- object
- undefined

*anche le classi
sono function*

*tutto ciò che
non rientra negli
altri casi
(incluso null!)*

non funziona con le interfacce!

TYPE NARROWING: TYPE PREDICATES

```
function isString(source?: string | number): source is string {  
    return typeof source === "string";  
}
```

type predicate

```
function date(source?: string | number): Date {  
    if (isString(source)) {  
        return new Date(source);  
    } else if (typeof source === "number") {  
        const today = new Date();  
        return new Date(today.getTime() + source * 24 * 60 * 60 * 1000);  
    } else {  
        return new Date();  
    }  
}
```

→ qui source è di tipo string

TYPE NARROWING

union types

equivalents:
name: string | undefined

```
function greet(name?: string) {  
  if (name) {  
    return `Hello, ${name}!`;  
  }  
  return "Hello, World!";  
}
```



unknown

```
function greet(users: string | string[]) {  
  if (typeof users === "string") {  
    return `Hello, ${users}!`;  
  }  
  return `Hello, ${users.join(" and ")}!`;  
}
```

optional fields

```
function greet(name: unknown) {  
  if (typeof name === "string") {  
    return `Hello, ${name}!`;  
  }  
  throw new Error("Invalid name");  
}
```

ANY VS UNKNOWN

	any	unknown
Contenuto	Qualsiasi valore <pre>let x: any = 42; x = "hello";</pre>	Qualsiasi valore <pre>let x: unknown = 42; x = "hello";</pre>
Assegnamento	Si può assegnare a qualsiasi altro tipo <pre>let x: any = 42; const text: string = x;</pre>	Richiede cast o type narrowing <pre>let x: unknown = 42; const <u>text</u>: string = x;</pre>
Operazioni	Qualsiasi <pre>let x: any = 42; x.toUpperCase();</pre>	Nessuna senza cast o type narrowing <pre>let x: unknown = 42; <u>x</u>.toUpperCase();</pre>
Type safety	 Non type safe	 Type safe

Type
'unknown'
is not
assignable
to type
'string'

'x' is of
type
'unknown'

DISCRIMINATED UNION

discriminant property
(tag)

```
type Success = {  
  status: "success";  
  data: string;  
};
```

```
type Failure = {  
  status: "failure";  
  error: string;  
};
```

```
type Result = Success | Failure;
```

```
type Result = {  
  status: "success" | "failure";  
  data?: string;  
  error?: string;  
};
```

qui result è di
tipo Success

```
function handleResult(result: Result) {  
  switch (result.status) {  
    case "success":  
      console.log("Success:", result.data);  
      break;  
    case "failure":  
      console.error("Failure:", result.error);  
      break;  
  }  
}
```

qui result è di tipo Failure

EXHAUSTIVE CHECK (NEVER)

```
function handleResult(result: Result) {  
  switch (result.status) {  
    case "success":  
      console.log("Success:", result.data);  
      break;  
    case "failure":  
      console.error("Failure:", result.error);  
      break;  
    default:  
      const _exhaustiveCheck: never = result;  
      throw new Error("Unhandled case");  
  }  
}
```

```
type Running = {  
  status: "running";  
};
```

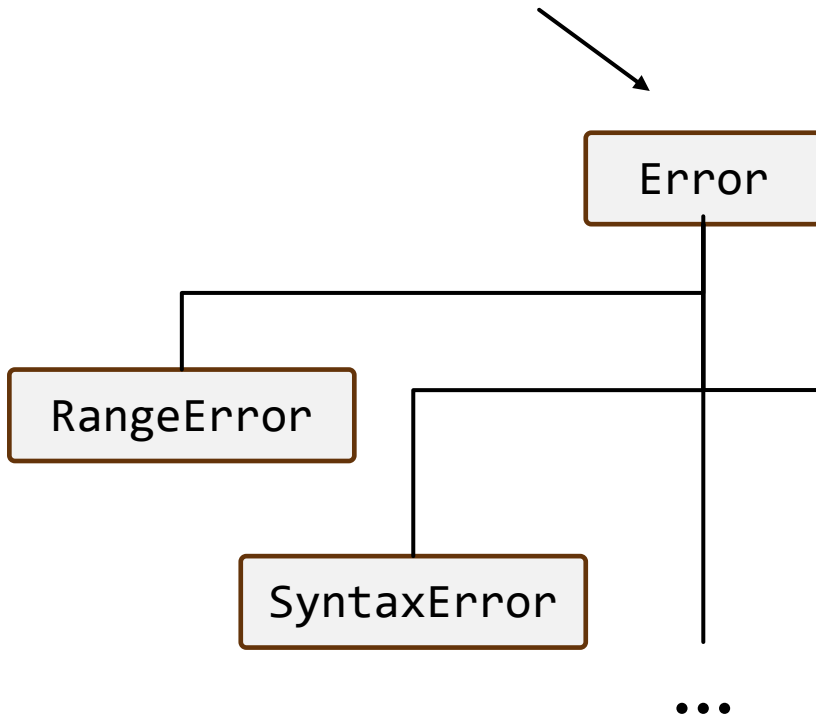
```
type Result =  
  | Success  
  | Failure  
  | Running;
```

exhaustive check: nessun tipo (a parte never)
si può assegnare a una variabile never

Type 'Running' is not assignable to type 'never'

GESTIONE DEGLI ERRORI

throw expression



try { *è di tipo unknown*
code block
} catch (err) {
error handling block
}

```
class CustomError extends Error {  
  constructor(message: string, options?: ErrorOptions) {  
    super(message, options);  
    this.name = "CustomError";  
  }  
}
```

GESTIONE DEGLI ERRORI

```
try {  
  functionThatCanThrow();  
} catch (err) {  
  if (err instanceof CustomError) {  
    console.error("Caught a CustomError:", err.message);  
  } else if (err instanceof Error) {  
    console.error("Caught a generic error:", err.message);  
  } else {  
    console.error("Caught an unknown error:", err);  
  }  
}
```

→ err è di tipo CustomError

→ err è di tipo Error

→ err è di tipo unknown

TYPE ASSERTIONS

```
type Config = {  
  localPath: string;  
  debug: boolean;  
};
```

```
const parsed = JSON.parse(raw);  
const conf = parsed as Config;  
  
console.log("Local path: ", conf.localPath);  
console.log("Verbose:", conf.verbose);
```

parsed **as** Config
<Config>parsed

type assertion

any

conf è di
tipo Config

```
try {  
  ...  
} catch (err: unknown) {  
  const e = err as Error;  
  console.error("Errore: ", e.message);  
}
```

la type assertion forza la conversione di
err in un oggetto di tipo Error

TYPE ASSERTIONS



- Solo tra tipi compatibili
- Può generare errori a runtime!
- Type assertion !== casting
- Si usa a proprio rischio!

```
const a = 42;  
const b = a as string; ❌
```

```
type Vehicle = { wheels: number };  
type Car = { wheels: number; doors: number };  
type Bike = { wheels: number; hasBell: boolean };  
  
const car = { wheels: 4, doors: 4 };  
const vehicle = car as Vehicle; ✅  
const bike = car as Bike; ❌
```

```
const a = 42;  
const b = a as unknown as string; ✅  
console.log(b.toUpperCase());
```




b.toUpperCase is not a function

NON-NULL ASSERTIONS & ASSERTION FUNCTIONS

non-null assertion
variable!

```
const element = document.getElementById("content");  
element!.textContent = "Hello, World";
```



HTMLElement | null

assertion functions
asserts variable is Type

```
type User = {  
  id: number;  
};  
  
function assertUser(value: unknown): asserts value is User {  
  if (!value || typeof value !== "object"  
    || !("id" in value)) {  
    throw new Error("Invalid user");  
  }  
}
```

TYPE PREDICATES VS ASSERTION FUNCTIONS

	type predicate	assertion function
Scopo	Verifica se un oggetto è di un certo tipo	Asserisce che un oggetto è di un certo tipo
Return type	boolean	void (o eccezione)
Quando usarla	In una condizione, come type guard	Per forzare il riconoscimento di un tipo
Esempio	<pre>function isString(value: unknown): value is string { return typeof value === "string"; } function printUpperCase(value: unknown) { if (!isString(value)) { throw new Error("Not a string"); } console.log(value.toUpperCase()); }</pre>	<pre>function assertString(value: unknown): asserts value is string { if (typeof value !== "string") { throw new Error("Not a string"); } } function printUpperCase(value: unknown) { assertString(value); console.log(value.toUpperCase()); }</pre>

CONST ASSERTIONS

```
{  
  readonly name: "Clio";  
  readonly age: 3;  
  readonly vaccines: readonly [  
    "rabbia",  
    "felv"  
  ];  
  readonly owner: {  
    readonly name: "Giuseppe";  
    readonly surname: "Giusti";  
  };  
}
```

```
const cat = {  
  name: "Clio",  
  age: 3,  
  vaccines: ["rabbia", "felv"],  
  owner: { name: "Giuseppe", surname: "Giusti" },  
} as const;  
  
cat.name = "Fluffy";  
cat.vaccines.push("fcv");  
cat.owner.name = "Pippo";
```

readonly è un modificatore che rende una variabile immutabile e si può applicare alle proprietà di un oggetto, ai membri di un type alias, agli array, alle tuple, agli attributi di una classe

CONST ASSERTIONS

- Cosa fanno
 - i tipi literal non vengono espansi (**no widening**)
 - gli oggetti literal hanno tutte le proprietà **readonly** (deep)
 - gli array literal diventano **tuple**

- Quando usarle:
 - Restringere i tipi ai literal
 - Immutability a compile time
 - Al posto degli enum

```
enum Status {  
  Active = "ACTIVE",  
  Deleted = "DELETED",  
}
```



```
const Status = {  
  Active: "ACTIVE",  
  Deleted: "DELETED",  
} as const;
```

- Funzionano solo con i literal!



```
const person = {  
  name: "Arturo",  
  surname: "Rossi"  
};
```



```
const cat = {  
  name: "Clio",  
  owner: person,  
} as const;  
person.name = "Luigi"; ✓
```

BRANDED TYPES

```
function squareRoot(x: number): number {  
    return Math.sqrt(x);  
}  
  
console.log(squareRoot(-1));    // NaN
```

il tipo **Positive** è
un'estensione di
number pertanto
lo structural
typing consente
di assegnarlo a
un **number**

```
type Positive = number & { __brand: 'positive' };  
  
function squareRoot(x: Positive): number {  
    return Math.sqrt(x);  
}  
  
let num = 25 as Positive;  
console.log(squareRoot(num));
```

viene aggiunta una proprietà
literal al tipo per distinguerlo

le type assertion
consentono di trattare i
valori come branded
type (ma non ci sono
controlli!)

BRANDED TYPES

un'altra valida
alternativa è
usare un type
predicate

```
type Positive = number & { __brand: "positive" };

function assertPositive(x: number): asserts x is Positive {
  if (x <= 0) {
    throw new Error("Value is not positive");
  }
}

function positiveNumber(x: number): Positive {
  assertPositive(x);
  return x;
}

let num = 25;
console.log(squareRoot(positiveNumber(num)));
```

l'assert function ci garantisce
che qui x è di tipo Positive

GENERICIS

```
function firstElement(arr: any[]): any {  
    return arr[0];  
}
```

```
const arr1 = [1, 2, 3];  
const arr2 = ['a', 'b', 'c'];
```

→ number[]
→ string[]

any ←

```
const first1 = firstElement(arr1);  
const first2 = firstElement(arr2);
```

```
function firstElementNumber(arr: number[]): number {  
    return arr[0];  
}
```

```
function firstElementString(arr: string[]): string {  
    return arr[0];  
}
```

...

GENERICIS

dichiaro che la funzione utilizzerà un tipo generico rappresentato dalla lettera T

```
function firstElement<T>(arr: T[]): T {  
    return arr[0];  
}
```

```
const arr1 = [1, 2, 3];  
const arr2 = ["a", "b", "c"];
```

```
const first1 = firstElement(arr1);  
const first2 = firstElement(arr2);
```

number[]
string[]

I Generics sono un costrutto del linguaggio che consente di scrivere codice flessibile e riutilizzabile, usando dei placeholder che vengono sostituiti dai tipi attuali quando il codice viene eseguito

GENERICIS

la funzione utilizza due tipi generici,
identificati dalle lettere T e U

```
function mapArray<T, U>(arr: T[], fn: (el: T) => U): U[] {  
  return arr.map(fn);  
}
```

```
const nums = [1, 2, 3, 4, 5];  
const squares = mapArray<number, number>(nums, (x) => (x * x));  
const strings = mapArray<number, string>(nums, (x) => `Number: ${x}`);
```

number[] ←
string[] ←

quando si utilizzano i generics si possono specificare i tipi
attuali, ma spesso non è necessario perché vengono inferiti

GENERICIS: FUNZIONI

equivalente a

async function callApi<Payload>(url: string): Promise<Payload>

```
const callApi = async <Payload>(url: string): Promise<Payload> => {  
  const result = await fetch(url);  
  const data = (await result.json()) as Payload;  
  return data;  
};
```

in TypeScript Promise<T> è un tipo generico, dove T è il tipo del risultato

```
const todo = await callApi<TodoPayload>("https://jsonplaceholder.typicode.com/todos/1");  
const post = await callApi<PostPayload>("https://jsonplaceholder.typicode.com/posts/1");
```

```
type TodoPayload = {  
  userId: number;  
  id: number;  
  title: string;  
  completed: boolean;  
};
```

```
type PostPayload = {  
  userId: number;  
  id: number;  
  title: string;  
  body: string;  
};
```

GENERICSTYPE ALIAS

```
type Page<T> = {  
  elements: T[];  
  totalPages: number;  
  currentPage: number;  
  pageSize: number;  
};
```

```
const p: Page<string> = {  
  elements: ["a", "b", "c", "d", "e"],  
  totalPages: 5,  
  currentPage: 1,  
  pageSize: 5  
};
```

GENERIC: CLASSI & INTERFACCE

```
interface Serializer<Input, Output> {  
    serialize(input: Input): Output;  
    deserialize(output: Output): Input;  
}
```

in questo esempio la classe
fissa uno dei due tipi
generici dell'interfaccia e
lascia generico l'altro

in questo esempio
la classe generica
implementa
un'interfaccia
generica, ma non è
un requisito!

```
class JsonSerializer<T> implements Serializer<T, string> {  
    serialize(input: T): string {  
        return JSON.stringify(input);  
    }  
  
    deserialize(output: string): T {  
        return JSON.parse(output) as T;  
    }  
}
```

```
type User = { name: string; age: number };  
const serializer = new JsonSerializer<User>();
```

GENERIC: CONSTRAINTS

```
interface Identifiable {  
    id: string;  
}  
  
class Repository<T extends Identifiable> {  
    private items: T[] = [];  
  
    add(item: T): void {  
        this.items.push(item);  
    }  
  
    findById(id: string): T | undefined {  
        return this.items.find((i) => i.id === id);  
    }  
}
```

si può fare riferimento a id perché T estende Identifiable

```
type Item = {  
    id: string;  
    name: string;  
};
```

```
new Repository<Item>(); ✓
```

```
type BlobData = {  
    data: string;  
};
```

```
new Repository<BlobData>(); ✗
```

Type 'BlobData' does not satisfy the constraint 'Identifiable'

GENERIC: DEFAULT

```
function wrapInArray<T>(value?: T): T[] {  
  return value !== undefined ? [value] : [];  
}
```

```
const w1 = wrapInArray("b"); —————> string[]  
const w2 = wrapInArray(); —————> unknown[]
```

```
function wrapInArray<T = number>(value?: T): T[] {  
  return value !== undefined ? [value] : [];  
}
```

```
const w1 = wrapInArray("b"); —————> string[]  
const w2 = wrapInArray(); —————> number[]  
const w3 = wrapInArray<string>(); —————> string[]
```

**Type inference e tipi
espliciti hanno
priorità sul default**

RIPASSO: I TIPI SEMPLICI

tipi primitivi	string, number, boolean, null, undefined, any, unknown, never, void
type annotations	<code>const</code> uname: string = "marco";
array e tuple	<code>const</code> nums: number[] = [1, 2, 3]; <code>const</code> xy: [number, number] = [1, 5];
funzioni	<code>function</code> exec(fn: () => string): string { return fn(); }
oggetti	<code>const</code> book: { title: string; author: string } = { title: "1984", author: "George Orwell", };

type alias	<code>type</code> Animal = { species: string; };
union e intersection	<code>type</code> Dirs = "N" "S" "O" "W"; <code>type</code> LandA = Animal & {legs: number};
interfacce	<code>interface</code> Person { name: string; surname: string; }
classi	<code>class</code> Programmer <code>implements</code> Person { name: string; surname: string; constructor(n: string, s: string) { ... } }

RIPASSO: INFERENZA E TIPI COMPOSTI

type inference	<pre>let user = "Marco"; const pi = 3.14;</pre> <div>user: string</div> <div>pi: 3.14</div>
type narrowing typeof in instanceof truthiness equality	<pre>function hello(name?: string) { if (name) { return `HI, \${name.toUpperCase()}!`; } return "Hello, world!"; }</pre> <div>name: string undefined</div> <div>name: string</div>
any unknown never	<pre>let a: any = 123; a.toUpperCase(); ✓ let u: unknown = 123; <u>u</u>.toUpperCase(); ✗ function fail(): never { throw new Error("Boom"); }</pre>

RIPASSO: INFERENZA E TIPI COMPOSTI

discriminated unions	<pre>type ApiResponse = { status: "ok"; data: string } { status: "error"; message: string }; function handle(res: ApiResponse): string { switch (res.status) { case "ok": return res.data; case "error": return `Error: \${res.message}`; } }</pre> <div>res: ApiResponse</div> <div>res: { status: "ok"; data: string }</div> <div>res: { status: "error"; message: string }</div>
type assertions	<pre>let val: string number = "test"; (val as string).toUpperCase(); let user: string undefined = "Marco"; user!.toUpperCase();</pre> <div>val: string number</div> <div>(val as string): string</div> <div>user: string undefined</div> <div>(user!): string</div>

RIPASSO: INFERENZA E TIPI COMPOSTI

const assertions	<pre>const Status = { Active: "ACTIVE", Deleted: "DELETED", } as const;</pre>	<div>Status: { readonly Active: "ACTIVE"; readonly Deleted: "DELETED"; }</div>
type predicates	<pre>function isString(value: unknown): value is string { return typeof value === "string"; }</pre>	
assertion functions	<pre>function assertString(value: unknown): asserts value is string { if (typeof value !== "string") { throw new Error("Not a string"); } }</pre> <pre>let value: unknown = "Hello"; if (isString(value)) { console.log(value.toUpperCase()); } assertString(value); console.log(value.toUpperCase());</pre>	<div>value: unknown</div> <div>value: string</div> <div>value: unknown</div> <div>value: string</div>

RIPASSO: INFERENZA E TIPI COMPOSTI

branded types	<pre>type Positive = number & { __brand: "positive" }; function makePositive(n: number): Positive { if (n <= 0) { throw new Error("Number must be positive"); } return n as Positive; } const positiveNumber: Positive = makePositive(42);</pre>
generics	<pre>function mapArray<T, U>(arr: T[], fn: (el: T) => U): U[] { return arr.map(fn); } const nums = [1, 2, 3, 4, 5]; const squares = mapArray<number, number>(nums, (x) => (x * x)); const strings = mapArray<number, string>(nums, (x) => `Number: \${x}`);</pre>