TYPESCRIPT: CONCETTI AVANZATI

Venerdì 24 Ottobre

RIPASSO: I TIPI SEMPLICI

tipi primitivi	`string, number, boolean, null, undefined, any, unknown, never, void`
type annotations	`const uname: string = "marco";`
array e tuple	`const nums: number[] = [1, 2, 3];` `const xy: [number, number] = [1, 5];`

funzioni	`function exec(`
	`  fn: () => string`
	`): string {`
	`  return fn();`
	`}`

oggetti	`const book: {`
	`  title: string; author: string`
	`} = {`
	`  title: "1984",`
	`  author: "George Orwell",`
	`};`

type alias	`type Animal = {`
	`  species: string;`
	`};`

union e intersection	`type Dirs = "N" | "S" | "O" | "W";` `type LandA = Animal & {legs: number};`

interfacce	`interface Person {`
	`  name: string;`
	`  surname: string;`
	`}`

classi	`class Programmer implements Person {`
	`  name: string;`
	`  surname: string;`
	
	`  constructor(n: string, s: string) {`
	`  ...`
	`  }`
	`}`

RIPASSO: INFERENZA E TIPI COMPOSTI

type inference	`let user = "Marco";` `user: string` `const pi = 3.14;` `pi: 3.14`

type narrowing
typeof
in
instanceof
truthiness
equality
	`function hello(name?: string) {` `name: string | undefined`
	`  if (name) {`
	`  return `HI, ${name.toUpperCase()}!`;` `name: string`
	`  }`
	`  return "Hello, world!";`
	`}`

any
unknown
never
	`let a: any = 123;`
	`a.toUpperCase();` ✅
	
	`let u: unknown = 123;`
	`u.toUpperCase();` ❌
	
	`function fail(): never {`
	`  throw new Error("Boom");`
	`}`

RIPASSO: INFERENZA E TIPI COMPOSTI

discriminated unions

```
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}`;
  }
}
```

`res: ApiResponse`

`res: { status: "ok"; data: string }`

`res: { status: "error"; message: string }`

type assertions

```
let val: string | number = "test";
(val as string).toUpperCase();

let user: string | undefined = "Marco";
user!.toUpperCase();
```

`val: string | number`

`(val as string): string`

`user: string | undefined`

`(user!): string`

RIPASSO: INFERENZA E TIPI COMPOSTI

const assertions

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

```
Status: {
  readonly Active: "ACTIVE";
  readonly Deleted: "DELETED";
}
```

type predicates

assertion functions

```
function isString(value: unknown): value is string {
  return typeof value === "string";
}
function assertString(value: unknown): asserts value is string {
  if (typeof value !== "string") {
    throw new Error("Not a string");
  }
}

let value: unknown = "Hello";
if (isString(value)) {
  console.log(value.toUpperCase());
}
assertString(value);
console.log(value.toUpperCase());
```

```
value: unknown
```

```
value: string
```

```
value: unknown
```

```
value: string
```

RIPASSO: INFERENZA E TIPI COMPOSTI

branded types

```
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);
```

generics

```
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}`);
```

TYPE OPERATORS: KEYOF

keyof si può usare solo su tipi o oggetti definiti nel contesto dei tipi

```
type Person = {
  name: string;
  birthYear: number;
  getAge: () => number;
};

function getProperty(person: Person, key: keyof Person): string | number {
  if (key === "getAge") {
    return person.getAge();
  }
  return person[key];
}

const p: Person = { name: "Sara", birthYear: 1985, getAge: () => 39 };
console.log(`Ciao, ${getProperty(p, "name")}`);
console.log(`Hai ${getProperty(p, "getAge")} anni`);
console.log(getProperty(p, "fiscalCode"));
```

"name" | "birthYear" | "getAge"

Argument of type '"fiscalCode"' is not assignable to parameter of type 'keyof Person'

TYPE OPERATORS: KEYOF

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}
```

```
const v: Vector = {};
v[1] = 42;
v["pippo"] = 100;
```

```
const d: Dictionary = {};
d["name"] = "Marco";
d[0] = "Mario";
```

```
type Vector = { [index: number]: number };
type Dictionary = { [key: string]: any };

let index: keyof Vector;
let dictKey: keyof Dictionary;
```

→ number

→ string | number

TYPE OPERATORS: TYPEOF

```
User: {
  name: string;
  age: number;
}
```

```
const user = {
  name: "Gaia",
  age: 28,
};

type User = typeof user;
```

***typeof* si può usare solo su identificatori o proprietà**

```
"name" | "age"
```

```
const user = {
  name: "Gaia",
  age: 28,
};

type UserProps = keyof typeof user;
```

***keyof typeof* estrae le proprietà di un oggetto nel contesto dei tipi**

VALUE-LEVEL

runtime

```
const user = {
  id: 1,
  name: "Alice",
};

function logKey(key) {
  console.log(key);
}

logKey("id");
```

TYPESCRIPT

```
const user = {
  id: 1,
  name: "Alice",
};

type User = typeof user;

type Keys = keyof User;

function logKey(key: Keys) {
  console.log(key);
}

logKey("id");
```

type inference

typeof

TYPE-LEVEL

compile time

```
const user: {
  id: number;
  name: string;
}

type User = {
  id: number;
  name: string;
}

type Keys = "id" | "name"

function logKey(
  key: "id" | "name"): void
```

keyof

TYPE OPERATORS: INDEXED ACCESS TYPES

Con la sintassi *T[K]* si indica il tipo della proprietà *K* appartenente al tipo *T*

```
type Config = {
  version: number;
  apiUrl: string;
  verbose: boolean;
};

type ApiUrlType = Config["apiUrl"];
type UrlOrVerboseType = Config["apiUrl" | "verbose"];
type ConfigValueTypes = Config[keyof Config];
```

tra parentesi quadre va indicato un tipo di chiave valido per `Config`

`string`

`string | boolean`

`number | string | boolean`

`"version" | "apiUrl" | "verbose"`

TYPE OPERATORS: INDEXED ACCESS TYPES

```
number
string
number | string
number | string
```

```
type Pair = [number, string];
type FirstType = Pair[0];
type SecondType = Pair[1];
type PairTypes = Pair[number];

type MyArray = Array<number | string>;
type ElementType = MyArray[number];
```

Nel caso delle tuple *number* va inteso come tutti i possibili indici numerici della tupla

```
"cat" | "dog" | "elephant"
```

```
const animals = ["cat", "dog", "elephant"] as const;
type Animal = (typeof animals)[number];

const people = [
  { name: "Luca", age: 30 },
  { name: "Maria", age: 25 },
];
type Person = (typeof people)[number];
```

```
Person: {
  name: string;
  age: number;
}
```

ESEMPIO: ALTERNATIVA AGLI ENUM

```
200 | 404 | 500
```

```
const HTTP_STATUS = {
  SUCCESS: 200,
  NOT_FOUND: 404,
  INTERNAL_ERROR: 500,
} as const;
type HttpStatus = typeof HTTP_STATUS[keyof typeof HTTP_STATUS];
```

```
"SUCCESS" | "NOT_FOUND" | "INTERNAL_ERROR"
```

```
type ApiResponse = {
  status: HttpStatus;
  content: string;
};
```

→

```
function handleResponse(response: ApiResponse) {
  switch (response.status) {
    case HTTP_STATUS.SUCCESS:
      console.log("Success:", response.content);
      break;
    case HTTP_STATUS.NOT_FOUND:
      console.log("Not Found:", response.content);
      break;
    ...
  }
}
```

CONDITIONAL TYPES

SomeType extends OtherType ? TypeA : TypeB

```
type IdType<T> = T extends { id: unknown } ? T["id"] : never;

const item1 = { id: "12abc", description: "Item 1" };
const item2 = { id: 123, description: "Item 2" };
const box = { content: "some content" };

type Item1IdType = IdType<typeof item1>;
type Item2IdType = IdType<typeof item2>;
type BoxIdType = IdType<typeof box>;
type AllIdType = IdType<typeof item1 | typeof item2 | typeof box>;
```

string ← Item1IdType

number ← Item2IdType

never ← BoxIdType

string | number ← AllIdType

typeof item1 → string

typeof item2 → number

typeof box → never

string | number | never

CONDITIONAL TYPES

```
type IdType<T> = T extends { id: unknown } ? T["id"] : never;
```

```
type IdType<T> = T extends { id: infer U } ? U : never;
```

```
type Return<T> = T extends (...args: any[]) => infer R ? R : never;

type GreetReturnType = Return<typeof greet>;
type SumReturnType = Return<typeof sum>;
type LogReturnType = Return<typeof log>;
```

string ← type GreetReturnType

number ← type SumReturnType

void ← type LogReturnType

```
function greet() {
  return "Hello world";
}
```

```
function log(message: string) {
  console.log(message);
}
```

```
function sum(a: number, b: number) {
  return a + b;
}
```

TEMPLATE LITERAL TYPES

```
type MouseEvents = ["Click", "Hover", "Focus"];
type ClickEventHander = `on${MouseEvents[0]}`;
type MouseEventHandlers = `on${MouseEvents[number]}`;
```

"onClick"

"onClick" | "onHover" | "onFocus"

```
type Elements = "Button" | "Input";
type ElementEvents = `${Elements}${MouseEvents[number]}`;
```

"ButtonClick" | "ButtonHover" | "ButtonFocus" | "InputClick" | "InputHover" | "InputFocus"

```
type ExtractPrefix<T> = T extends `${infer Prefix}_id` ? Prefix : never;

type UserType = ExtractPrefix<"user_id">;
type ProductType = ExtractPrefix<"product_id">
```

"user"

"product"

TEMPLATE LITERAL TYPES: INTRINSIC STRING MANIPULATION TYPES

```
type Resource = "user" | "product" | "order";

type MethodName = `get${Capitalize<Resource>}`; ⟶ "getUser" | "getProduct" | "getOrder"

type TableName = `${Uppercase<Resource>}`; ⟶ "USER" | "PRODUCT" | "ORDER"

type ApiPath = `/api/${Lowercase<TableName>}`; ⟶ "/api/user" | "/api/product" | "/api/order"

type ExtractFieldName<T extends MethodName> =
  T extends `get${infer Field}` ? Field : never;  ⟶ "user" | "product" | "order"

type FieldName = Uncapitalize<ExtractFieldName<MethodName>>;
```

Capitalize<S>

Uppercase<S>

Lowercase<S>

Uncapitalize<S>

MAPPED TYPES

```
type User = {
  name: string;
  level: number;
  isActive: boolean;
};
```

```
type CreateUserPayload = {
  [Property in keyof User]?: User[Property];
};
```

```
CreateUserPayload: {
  name?: string;
  level?: number;
  isActive?: boolean;
}
```

```
type Actions = "read" | "write" | "browse";

type Permission = {
  [A in Actions]: boolean;
};
```

```
Permission: {
  read: boolean;
  write: boolean;
  browse: boolean;
}
```

MAPPED TYPES

```
UserPropertySetter: {
  setName: (value: string) => void;
  setLevel: (value: number) => void;
  setIsActive: (value: boolean) => void;
}
```

```
type UserPropertySetter = {
  [P in keyof User as `set${Capitalize<string & P>}`]: (value: User[P]) => void;
};

type ExcludeProperty<T, K extends string> = {
  [P in keyof T as P extends K ? never : P]: T[P];
};

type UserWithoutLevel = ExcludeProperty<User, "level">;
```

```
UserWithoutLevel: {
  name: string;
  isActive: boolean;
}
```

UTILITY TYPES

Partial<T>

```
type Partial<T> = { [P in keyof T]?: T[P] | undefined; }
```

Rende opzionali tutte le proprietà del tipo *T*

```
type User = {
  name: string;
  level: number;
  isActive: boolean;
};

type PartialUser = Partial<User>;
```

```
PartialUser: {
  name?: string | undefined;
  level?: number | undefined;
  isActive?: boolean | undefined;
}
```

UTILITY TYPES

Required<T>

```
type Required<T> = { [P in keyof T]-?: T[P]; }
```

Rende obbligatorie tutte le proprietà del tipo *T*

```
type User = {
  name: string;
  level?: number;
  isActive?: boolean;
};

type RequiredUser = Required<User>;
```

```
RequiredUser: {
  name: string;
  level: number;
  isActive: boolean;
}
```

UTILITY TYPES

Readonly<T>

```
type Readonly<T> = { readonly [P in keyof T]: T[P]; }
```

Rende readonly tutte le proprietà del tipo *T (shallow)*

```
type User = {
  id: number;
  isActive: boolean;
  personalData: {
    name: string;
    surname: string;
  }
};

type ReadonlyUser = Readonly<User>;
```

```
ReadonlyUser: {
  readonly id: number;
  readonly isActive: boolean;
  readonly personalData: {
    name: string;
    surname: string;
  }
}
```

UTILITY TYPES

Exclude<T, K>

```
type Exclude<T, U> = T extends U ? never : T
```

Esclude dal tipo (unione) *T* tutti gli elementi di tipo *K*

```
type ExcludedString = Exclude<"a" | "b" | "c", "a" | "b">;

type ExcludedObject = Exclude<
  { a: string } | { b: number } | { c: boolean },
  { a: string }
>;
```

```
ExcludedString: "c"
```

```
ExcludedObject: {
  b: number;
} | {
  c: boolean
}
```

UTILITY TYPES

Extract<T, K>

```
type Extract<T, U> = T extends U ? T : never
```

Estrae dal tipo (unione) *T* tutti gli elementi di tipo *K*

```
type ExtractedString = Extract<"a" | "b" | "c", "a" | "b">;

type ExtractedObject = Extract<
  { a: string } | { b: number } | { c: boolean },
  { a: string }
>;
```

```
ExtractedString:
"a" | "b"
```

```
ExtractedObject: {
  a: string;
}
```

UTILITY TYPES

NonNullable<T>

```
type NonNullable<T> = T & {}
```

Esclude null e undefined dal tipo *T*

```
type NonNullableString = NonNullable<string | null>;

type NonNullableMixed = NonNullable<
  string | number | null | undefined
>;
```

NonNullableString: string

NonNullableMixed: string | number

UTILITY TYPES

Pick<T, K>

```
type Pick<T, K extends keyof T> = { [P in K]: T[P]; }
```

Crea un tipo che include solo le proprietà di *T* elencate in *K*

```
type User = {
  name: string;
  level: number;
  isActive: boolean;
};

type UserWithLevel = Pick<User, "name" | "level">;
```

```
UserWithLevel: {
  name: string;
  level: number;
}
```

UTILITY TYPES

Omit<T, K>

```
type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>
```

Crea un tipo che include tutte le proprietà di *T* tranne quelle elencate in *K*

```
type User = {
  name: string;
  level: number;
  isActive: boolean;
};

type UserWithLevel = Omit<User, "isActive">;
```

```
UserWithLevel: {
  name: string;
  level: number;
}
```

UTILITY TYPES

Record<K,T>

```
type Record<K extends keyof any, T> = { [P in K]: T; }
```

Crea un tipo oggetto in cui le chiavi sono del tipo *K* e i valori sono del tipo *T*

```
type Roles = "player1" | "player2";

type Player = {
  id: string;
  username: string;
};

type Game = Record<Roles, Player>;
```

```
Game: {
  player1: {
    id: string;
    username: string;
  };
  player2: {
    id: string;
    username: string;
  };
}
```

UTILITY TYPES

Parameters<T>

```
type Parameters<T extends (...args: any) => any> =
  T extends (...args: infer P) => any ? P : never;
```

Estrae i tipi dei parametri dalla funzione di tipo *T*

```
type ParamsEmpty = Parameters<() => string>;
```

→ `ParamsEmpty: []`

```
type ParamsNumber = Parameters<(value: number) => void>;
```

→ `ParamsNumber: [value: number]`

```
type ParamsMixed = Parameters<(a: string, b: number) => string>;
```

→ `ParamsMixed: [a: string, b: number]`

UTILITY TYPES

ReturnType<T>

```
type ReturnType<T extends (...args: any) => any> =
  T extends (...args: any) => infer R ? R : any;
```

Estrae il tipo di ritorno dalla funzione di tipo *T*

```
type StringReturnType = ReturnType<() => string>;

StringReturnType: string

type VoidReturnType = ReturnType<(a: string) => void>;

VoidReturnType: void

type MixedReturnType = ReturnType<() => string | number>;

MixedReturnType: string | number
```

SATISFIES

```
type Options = {
  apiEndpoint: string;
  outputPath: string;
  verbose: boolean;
};

const config = {
  apiEndpoint: "https://api.example.com",
  fakeOption: "error",
};

console.log(config.apiEndpoint.toLocaleLowerCase());
console.log(config.outputPath.toLocaleLowerCase());
console.log(config.verbose);
```

```
config: {
  apiEndpoint: string;
  fakeOption: string;
}
```

Property 'verbose' does not exist

Property 'outputPath' does not exist

SATISFIES

```
type Options = {
  apiEndpoint: string;
  outputPath: string;
  verbose: boolean;
};

const config: Partial<Options> = {
  apiEndpoint: "https://api.example.com",
  fakeOption: "error",
};

console.log(config.apiEndpoint.toLocaleLowerCase());
console.log(config.outputPath.toLocaleLowerCase());
console.log(config.verbose);
```

```
config: {
  apiEndpoint?: string;
  outputPath?: string;
  verbose?: boolean;
}
```

Possibly undefined

undefined

SATISFIES

```
type Options = {
  apiEndpoint: string;
  outputPath: string;
  verbose: boolean;
};

const config = {
  apiEndpoint: "https://api.example.com",
  fakeOption: "error",
} as Partial<Options>;

console.log(config.apiEndpoint.toLocaleLowerCase());
console.log(config.outputPath.toLocaleLowerCase());
console.log(config.verbose);
```

```
config: {
  apiEndpoint?: string;
  outputPath?: string;
  verbose?: boolean;
}
```

Possibly undefined

undefined

SATISFIES

```
type Options = {
  apiEndpoint: string;
  outputPath: string;
  verbose: boolean;
};

const config = {
  apiEndpoint: "https://api.example.com",
  fakeOption: "error",
} satisfies Partial<Options>;

console.log(config.apiEndpoint.toLocaleLowerCase());
console.log(config.outputPath.toLocaleLowerCase());
console.log(config.verbose);
```

***satisfies* verifica un tipo a compile time ma non altera il tipo inferito**

```
config: {
  apiEndpoint: string;
  fakeOption: string;
}
```

Property 'verbose' does not exist

Property 'outputPath' does not exist

FUNCTIONAL PROGRAMMING IN TYPESCRIPT

- **Funzioni pure** ✗ il linguaggio non fornisce supporto nativo
- **Immutabilità** `const` `readonly` `Readonly<T>` `ReadonlyArray<T>`
- **Funzioni di ordine superiore (HOF)**

```
const values = [1, 2, 3, 4, 5, 6];

const sumOfSquaresOfEven = values
  .filter(n => n % 2 === 0)
  .map(n => n * n)
  .reduce((acc, n) => acc + n, 0);
```

filtra solo i numeri pari
calcola il quadrato dei valori
somma tutti i valori

calcola la somma dei quadrati dei numeri pari

FUNCTIONAL PROGRAMMING IN TYPESCRIPT

- **Composizione di funzioni e currying**

```
const add = (x: number): number => x + 1;
const double = (x: number): number => x * 2;

const compose = <T, U, V>(f: (y: U) => V, g: (x: T) => U) => (x: T): V => f(g(x));
const addThenDouble = compose(double, add);
```

```
const sum = (a: number, b: number): number => a + b;
const curriedSum = (a: number) => (b: number) => a + b;
const add5 = curriedSum(5);
```

USARE TYPESCRIPT SUL FRONTEND: TSCONFIG

React

```
{
  "compilerOptions": {
    ...
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "jsx": "react-jsx",
  },
  ...
}
```

```
npm i --save-dev @types/react @types/react-dom
```

AngularJS

```
{
  "compilerOptions": {
    ...
    "lib": [
      "dom",
      "es2022"
    ],
    "experimentalDecorators": true,
    "angularCompilerOptions": {
      ...
    },
  },
  ...
}
```

USARE TYPESCRIPT SUL FRONTEND

React

- Tipi generici
- Union types
- Type guards

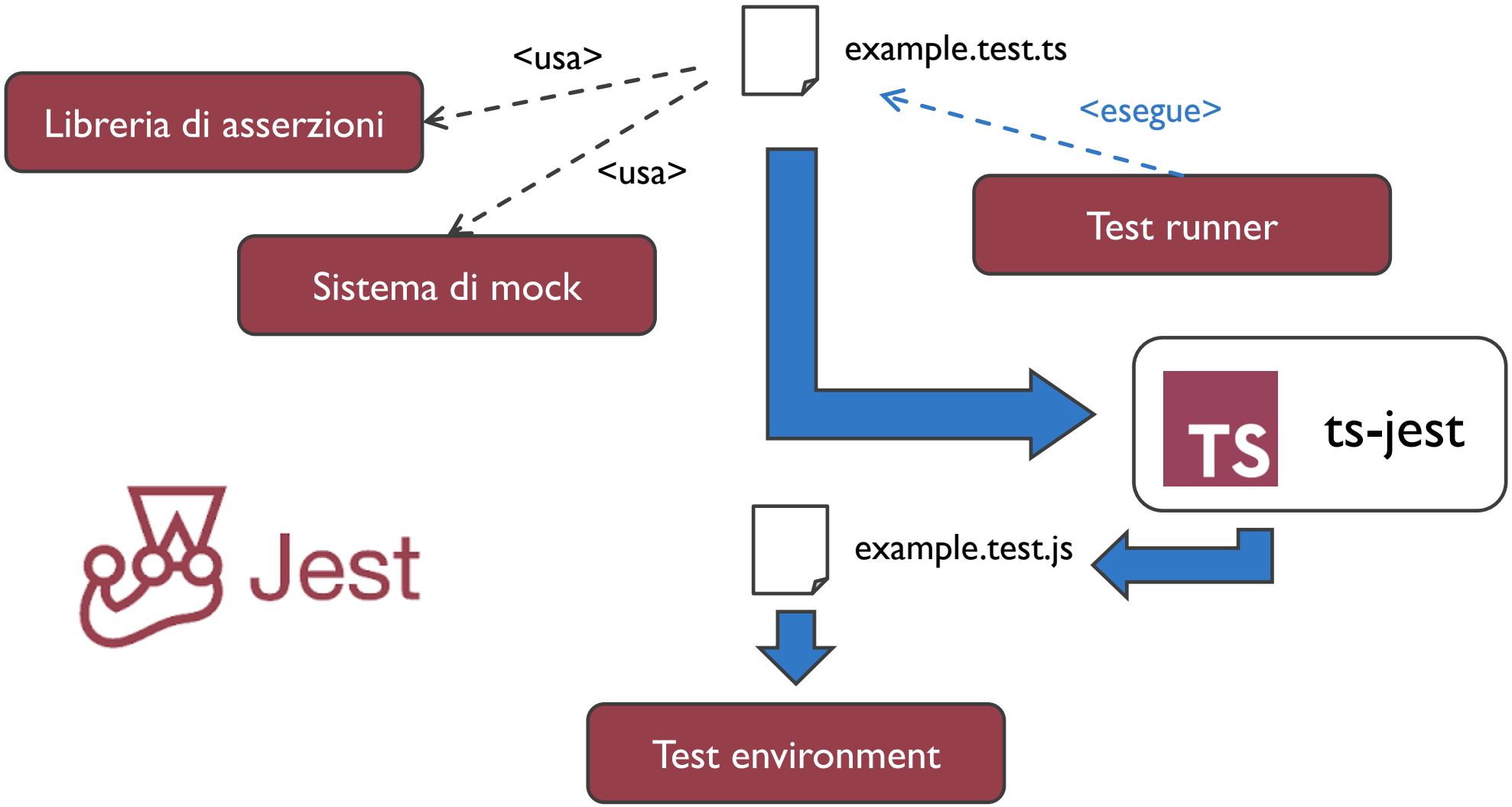
AngularJS

- Classi
- Interfacce
- Decoratori

Uso dei tipi predefiniti per gli oggetti e gli eventi del DOM

Uso intensivo della type inference

TESTING



Jest

TESTING

```
npm install --save-dev jest @types/jest ts-jest
```

jest.config.cjs

```
module.exports = {
  testEnvironment: "node",
  transform: {
    "^.+\\.ts$": "ts-jest",
  },
  moduleNameMapper: {
    "^(.+)\\.js": "$1",
  },
  transformIgnorePatterns: ["node_modules"],
  testMatch: ["**/*.test.ts"],
};
```

ambiente di esecuzione: `node` o `jsdom`

usa `ts-jest` per trasformare i file `.ts`

rimuovi l'estensione dai file `.js` negli import

TESTING

math.ts

```
export function fibonacci(
  n: number
): number {
  if (n < 0) {
    throw new Error("Bad input");
  }
  if (n === 0) {
    return 0;
  }
  if (n === 1) {
    return 1;
  }
  return fibonacci(n - 1) +
         fibonacci(n - 2);
}
```

math.test.ts

```
import { fibonacci } from "../src/math.js";

test("fibonacci of a negative number", () => {
  expect(() => fibonacci(-1)).toThrow();
});

test("fibonacci of 0 should return 0", () => {
  expect(fibonacci(0)).toBe(0);
});

test("fibonacci of a positive number", () => {
  expect(fibonacci(1)).toBe(1);
  expect(fibonacci(2)).toBe(1);
  expect(fibonacci(3)).toBe(2);
  expect(fibonacci(5)).toBe(5);
  expect(fibonacci(10)).toBe(55);
  expect(fibonacci(15)).toBe(610);
});
```

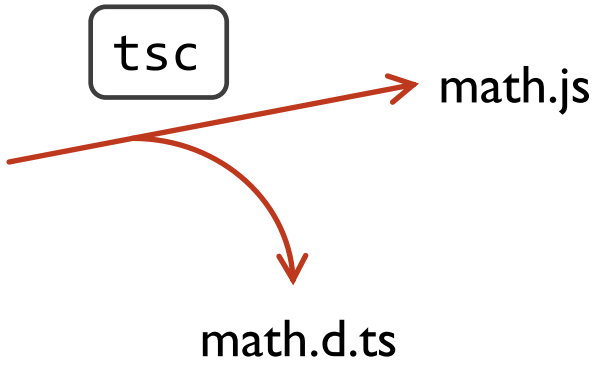
DECLARATION FILES

I file *.d.ts* contengono solo informazioni sui tipi, senza implementazioni

math.ts

```
export function sum(
  a: number,
  b: number
): number {
  return a + b;
}

export function multiply(
  a: number,
  b: number
): number {
  return a * b;
}
```




```
export function sum(a, b) {
    return a + b;
}

export function multiply(a, b) {
    return a * b;
}
```

```
export declare function sum(a: number, b: number):
number;

export declare function multiply(a: number, b: number):
number;
```

RIPASSO: CONCETTI AVANZATI

operatori
keyof
typeof
	`const user = {`
	`  id: 1,`
	`  name: "Alice"`
	`};`
	`typeof user: { id: number; name: string; }`
	`type KeyOfUser = keyof typeof user;`
	`KeyOfUser: "id" | "name"`

indexed access types	`const people = [`
	`  { name: "Greta", age: 30 },`
	`  { name: "Daniele", age: 25 },`
	`];`
	`typeof people: { name: string; age: number; }[]`
	`(typeof people)[number]: { name: string; age: number; }`
	`type Name = (typeof people)[number]["name"];`
	`Name: string`

conditional e template literal types	`const style = {`
	`  background_color: 'blue',`
	`  foreground_color: 'white',`
	`};`
	`StyleKeys: "background" | "foreground"`
	`type ExtractStyle<T> = T extends `${infer Prefix}_color` ? Prefix : never;`
	`type StyleKeys = ExtractStyle<keyof typeof style>;`

RIPASSO: CONCETTI AVANZATI

mapped types e utility types	`type Partial<T> = { [P in keyof T]?: T[P] | undefined };`
	`type Required<T> = { [P in keyof T]-?: T[P] };`
	`type Readonly<T> = { readonly [P in keyof T]: T[P] };`

satisfies	`type Options = {`
	`  apiEndpoint: string;`
	`  outputPath: string;`
	`  verbose: boolean;`
	`};`
	
	`const config = {`
	`  apiEndpoint: "https://api.example.com",`
	`} satisfies Partial<Options>;`
	
	`console.log(config.apiEndpoint.toLowerCase());`
	
	`config:{ apiEndpoint: string }`