COESIONE ITALIA 21-27 PIEMONTE
Cofinanziato dall'Unione europea
REGIONE PIEMONTE

Web Developer

Programmazione - Javascript e Typescript

Docente: Shadi Lahham

Intervento realizzato da ITS TECNOLOGIE DELL'INFORMAZIONE E DELLA COMUNICAZIONE

Typescript

language features

Shadi Lahham - Web development

Typescript

Typescript

Typescript is a superset of Javascript

- valid JavaScript code is also valid TypeScript code
- builds on top of JavaScript adding new features
- maintains full compatibility with JavaScript
- intuitive and easy to learn, minimal learning curve
- incremental upgrade is a major benefit
    - easy to gradually migrate code from JavaScript to TypeScript
    - developers can start by gradually adding types to JavaScript code
    - additional Typescript features can be added when and as needed

Quickstart

Quick Typescript

Quickly try in the
TS Playground

Quick introduction
TypeScript for JavaScript Programmers and The Basics

Quick setup
TypeScript Tooling in 5 minutes

Setup & config

Typescript setup

To use TypeScript in your project, you need to

- set up a TypeScript compiler
- configure it using a tsconfig.json file

Install globally

You need to have Node.js installed on your machine and use command prompt

install Typescript globally with npm

```
npm install -g typescript
```

create a folder and a blank file

```
mkdir test
cd test
type nul > main.ts
```

run Typescript compiler

```
tsc main.ts
OR
tsc --watch main.ts
```

Install locally

You need to have Node.js installed on your machine and use command prompt

create a folder and a blank file

```
mkdir test
cd test
type nul > main.ts
```

install locally with npm

```
npm init -y
npm i -D typescript
```

run Typescript compiler locally

```
npx tsc main.ts
OR
npx tsc --watch main.ts
```

TypeScript Projects Compilation

The tsconfig.json file is a configuration file for TypeScript projects that specifies various options for the TypeScript compiler

When a tsconfig.json file is present in the folder, invoking tsc without any parameters will compile all the files included in the configuration

The tsconfig.json file:

- indicates the root of a TypeScript project
- specifies the root files and compiler options for project compilation

Run from the command line:

tsc

Sample tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES2015",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "outDir": "dist"
  },
  "$schema": "https://json.schemastore.org/tsconfig",
  "display": "Recommended",
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

Explaining tsconfig.json

"compilerOptions"
contains various options for the TypeScript compiler

"target": "ES2015"
specifies the ECMAScript version that the compiled JavaScript code should target; ES2015, ES5, etc.

"module": "commonjs"
specifies the module format that the compiled JavaScript code should use

"strict": true
enables strict type-checking options in TypeScript, which helps catch more errors at compile-time

"esModuleInterop": true
enables compatibility with CommonJS and AMD modules
Allows for interoperability between TypeScript and JavaScript code

Explaining tsconfig.json

"skipLibCheck": true
skips type-checking of declaration files in third-party libraries which means faster compilation

"forceConsistentCasingInFileNames": true
file references must use consistent casing; prevents issues when deploying to case-sensitive systems

"outDir": "dist"
specifies the output directory for compiled JavaScript files

"$schema": "https://json.schemastore.org/tsconfig"
enables editors and other tools to ensure tsconfig.json file is correct and follows expected format

"display": "Recommended"
indicates that the configuration options specified in this file are recommended by the TypeScript team

Explaining tsconfig.json

"include": ["src//*"]**
specifies files included in the compilation process
In this case all files in the src directory and its subdirectories

"exclude": ["node_modules"]
specifies the files and directories excluded from compilation

"noEmitOnError": false
generate Javascript file even when there are Typescript errors - noEmitOnError reference
change to true for safer and secure code once developers gain more experience with Typescript

Complete tsconfig.json options

Handbook - What is a tsconfig.json
@tsconfig/recommended
TSConfig option - target
TSConfig option - outDir
TSConfig option - include
All TSConfig options

Simple types

primitives

Inferred Types

When a variable is initialized without an explicit type declaration, TypeScript will try to infer the type based on the assigned value

```
// inferred type: string
let str = 'hello';

// inferred type: number
let num = 42;

// inferred type: boolean
let bool = true;

// sometimes typescript can't infer so it assigns the type of the variable to 'any'
const json = JSON.parse('false');
const json2 = JSON.parse('18');

// type of json and json2 is 'any' since typescript can't know until the code is executed
```

Explicit Types

Developers can also explicitly declare the types of variables in TypeScript
Explicitly declaring types can be helpful in situations where TypeScript cannot infer the type correctly, or when the developer wants to provide additional information to improve the clarity and readability of the code
However, if the type can be inferred correctly, there is no need to declare it explicitly

Explicit Types

main.ts

```ts
let str: string = 'hello';
let num: number = 42;
let bool: boolean = true;
```

main.js

```js
"use strict";
let str = 'hello';
let num = 42;
let bool = true;
```

Protection from type errors

```
let clientName: string = 'james';
clientName = 88;
// TypeScript compiler throws a type assignment error

// function param example
function printMessage(message: string) {
  console.log(message);
}

// Correct usage
printMessage('Good morning!'); // prints "Good morning!"

// Incorrect usage
printMessage(42);

// TypeScript will throw a compilation error
// Error: Argument of type 'number' is not assignable to parameter of type 'string'
```

Protection from type errors

```
function isPalindrome(word: string): boolean {
  const reversedWord = word.split('').reverse().join('');
  return word === reversedWord;
}

// Correct usage
const result1 = isPalindrome('racecar'); // returns true
const result2 = isPalindrome('hello'); // returns false

// Incorrect usage
const result3 = isPalindrome(42);

// TypeScript will throw a compilation error
// Error: Argument of type 'number' is not assignable to parameter of type 'string'
```

Special types

Undefined & null

Undefined and null can be useful for representing missing or uninitialized values

strictNullChecks off
Undefined and null are considered to be subtypes of all other types, which means that a variable of any type can also be assigned a value of undefined or null. This can lead to runtime errors and bugs

strictNullChecks on (part of "strict":true)
TypeScript will consider null and undefined values as distinct types, separate from all other types. The developer must explicitly check for these values before using them to avoid potential runtime errors

Undefined & null

```
let x: number | undefined = undefined; // variable can be a number or undefined
let y: string | null = null; // variable can be a string or null

function printName(name?: string) {
  console.log(name || 'Anonymous');
}

printName(); // prints "Anonymous"
printName('Alice'); // prints "Alice"
printName(undefined); // prints "Anonymous"

// Incorrect usage
printName(null);

// TypeScript will throw a compilation error
// Error: Argument of type 'null' is not assignable to parameter of type 'string | undefined'
```

Any

In Typescript, any is a special type that allows a variable to have any type

When a variable is declared with the any type, the TypeScript compiler assumes that it can have any possible value and disables all type checking on that variable Using any might convenient in some situations when dealing with data of unknown or varying types, but it is still not recommended since it disables any type safety and static checking features and risks runtime errors

Using any is basically like writing Javascript with Typescript, losing almost all of the advantages and introducing a big risk factor

Any

```
// x can be any type
let x: any = 'hello';

// valid even though x is declared as any
console.log(x.toUpperCase());

// also valid, even though x is no longer a string
x = 42;
console.log(x.toFixed());
console.log(x.toUpperCase()); // will result in a JS errors when run

// still valid, even though x is now an object
x = { prop: 'value' };
console.log(x.prop);
```

Note: removing the type x: any will re-enable type checking and show errors where x is reassigned

Unknown

The unknown type is similar to any, but provides more type safety and requires type checks or type assertions before using the value

Unlike any, it is not possible to call methods or access properties on an unknown variable without performing a type check or casting (type assertion)

The unknown type is best used when the type of value dealt with is not known

Unknown

```
// x can be any type
let x: unknown = 'hello';

// Compile-time error: Object is of type 'unknown'
console.log(x.toUpperCase());

// Compile-time error: Object is of type 'unknown'
x = 42;
console.log(x.toFixed());

// Compile-time error: Object is of type 'unknown'
x = { prop: 'value' };
console.log(x.prop);
```

Unknown with type checking

```ts
let x: unknown = 'hello';

// Type checking with typeof
if (typeof x === 'string') {
  console.log(x.toUpperCase()); // OK
}

// Type checking with typeof
x = 42;
if (typeof x === 'number') {
  console.log(x.toFixed(2)); // OK
}

// Type checking with typeof, null check, and 'in' operator
x = { prop: 'value' };
if (typeof x === 'object' && x !== null && 'prop' in x) {
  console.log((x as { prop: string }).prop); // OK
}
```

Unknown with casting

```
// examples of direct type assertion
let x: unknown = 'hello';

// assumes x is a string
console.log((x as string).toUpperCase());

x = 42;
// assumes x is a number
console.log((x as number).toFixed(2));

x = { prop: 'value' };
// assumes x is an object with 'prop'
console.log((x as { prop: string }).prop);
```

Unknown vs any

```ts
const json = JSON.parse('false');
json.doSomthing(); // implicit any

const json2: any = JSON.parse('false');
json2.doSomthing(); // explicit any

const json3: unknown = JSON.parse('false');
json3.doSomthing(); // compiler error: 'json3' is of type 'unknown'
```

any
allows any operation without errors, which can lead to runtime issues

unknown
requires a type check (e.g., typeof, instanceof) or casting before accessing properties, calling methods, or performing operations

Void & never

In TypeScript, void and never both denote functions that don't return a value. However, they have different meanings

Void indicates the absence of a value and is commonly used as a return type for functions that don't return a value or variables that aren't expected to have one

Never indicates a function that never completes normally, due to either throwing an error or entering an infinite loop. It's usually used as the return type for a function that always throws an exception or never returns

Void

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

const result: void = logMessage('Hello, world!'); // result is undefined
```

Never

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

function infiniteLoop(): never {
  while (true) {
    // do something
  }
}
```

Void & never

```ts
// this is ok
let x: void = undefined;

// Error: Type 'undefined' is not assignable to type 'never'
let y: never = undefined;
```

Arrays

Arrays

There are multiple ways to declare arrays in TypeScript

- Using the type[] syntax
- Using the Array<type> syntax

```ts
let myNumbers: number[] = [1, 2, 3];
let myStrings: Array<string> = ['hello', 'world'];

// same result
let moreNumbers: Array<number> = [1, 2, 3];
let moreStrings: string[] = ['hello', 'world'];
```

```ts
// the first is more common, because it's shorter
// the second will become clearer after discussing generics
```

Arrays - type protection

```
let myArray = [1, 2, 3]; // myArray is inferred to be of type number[]
myArray.push(16); // OK
myArray.push('16'); // error: argument of type 'string' not assignable to parameter of type 'number'
myArray = [1, 2, 3, 'hello', 'world']; // error: Type 'string[]' is not assignable to type 'number[]'

// explicit types
let myArray1: number[] = [1, 2, 3];
let myArray2: Array<string> = ['hello', 'world'];
myArray1 = myArray2; // error: Type 'string[]' is not assignable to type 'number[]'.
```

Readonly Arrays

```
// using readonly on an array variable
const myArray: readonly number[] = [1, 2, 3];
myArray.push(4); // error: property 'push' does not exist on type 'readonly number[]'
myArray[0] = 4; // error: index signature in type 'readonly number[]' only permits reading
myArray = [4, 5, 6, 7]; // error: cannot assign to 'myArray' because it is a constant

// using ReadonlyArray on an array value
const myReadonlyArray: ReadonlyArray<number> = [1, 2, 3];
myReadonlyArray.push(4); // error: property 'push' does not exist on type 'ReadonlyArray<number>'
myReadonlyArray[0] = 4; // error: index signature in type 'readonly number[]' only permits reading
myReadonlyArray = [4, 5, 6, 7]; // error: cannot assign to 'myReadonlyArray' because it is a constant

// using readonly in function parameters
function foo(colors: readonly string[]) {
  colors.push('olive'); // error: property 'push' does not exist on type 'readonly string[]'
}
```

Tuples

Tuples

Tuples are commonly used in TypeScript when working with fixed arrays of known types, enabling stricter type checking and providing clarity to data structure

Tuples are particularly useful when dealing with functions that return multiple values, as they ensure a fixed number and type of values

Tuples can also be helpful when working with APIs that return data in a specific format, as they ensure proper parsing and typing of the data

Tuples

In React, tuples define prop types, ensuring components receive the correct and consistent data structure

For instance, tuples can specify the expected object shape representing user data in a user profile component

Tuples

```
// define a tuple - a typed array
let myTuple: [boolean, number, string];

// initialize
myTuple = [true, 18, 'working'];

// number and type of the values matters
myTuple = ['baking', false, 26]; // throws type errors since order is incorrect
myTuple = [true, 18]; // error: type '[true, number]' is not assignable to type '[boolean, number, string]'
```

Tuples

```
// can modify values in the correct positions
myTuple[2] = 'jogging';
myTuple[1] = 'jogging'; // error: type 'string' is not assignable to type 'boolean'.(2322)

// can't exceed the index, but can still push and other methods
myTuple[3] = 'jogging';
myTuple.push('jogging');

// to prevent changes via methods use readonly
let noPushTuple: readonly [boolean, number, string] = [true, 18, 'working'];
noPushTuple.push('jogging'); // error: property 'push' does not exist on type 'readonly [number,
boolean, string]'
noPushTuple = [false, 56, 'eating']; // declared with let, so OK

// and const to prevent reassignment
const noChangeTuple: readonly [boolean, number, string] = [true, 18, 'working'];
// error: cannot assign to 'noChangeTuple' because it is a constant
noChangeTuple = [false, 56, 'eating'];
```

Tuples

```
// tuple as function parameter
function addEmployee(employee: readonly [string, number]) {
  // error: cannot assign to '0' because it is a read-only property
  employee[0] = 'sam';

  // error: tuple type '[string, number]' of length '2' has no element at index '2'
  const [name, id, gender] = employee;

  console.log(name); // const name: string
  console.log(id); // const id: number
  console.log(gender); // const gender: undefined
}

addEmployee(['jane', 1234]);

// error: argument of type '[string]' is not assignable to param of type 'readonly [string, number]'
addEmployee(['jane']);
```

Object Types

Basic object type

```
// Object with specific properties
let entity: { name: string; age: number } = {
  name: 'John',
  age: 30
};

// Type checking
entity = { name: 'Mary', age: '40' }; // error: type 'string' is not assignable to type 'number'
```

Passing objects to functions

```
// Function that takes an object with specific properties
function printPerson(person: { name: string; age: number }) {
  console.log(`Name: ${person.name}, Age: ${person.age}`);
}

// Type checking
printPerson({ name: 'John', age: 30 });
printPerson({ name: 'Mary' }); // error: property 'age' is missing in type '{ name: string; }'
```

Optional properties

```
// Object with optional properties
let person: { name: string; age?: number } = {
  name: 'John'
};

// Type checking
person = { name: 'Ann', age: 34 }; // OK
person = { name: 'Mary', age: '40' }; // error: type 'string' is not assignable to type 'number'
```

Readonly properties

```
// Object with readonly properties
let user: { readonly name: string; age: number } = {
  name: 'John',
  age: 30
};

// Type checking
user.name = 'Mary'; // error: cannot assign to 'name' because it is a read-only property
```

Index signature

```
// We might not know the object's property names, but know the index signature
let stats: { [key: string]: number } = {
  views: 1000,
  clicks: 200,
  conversions: 10
};

// Type checking
stats.distance = 200; // ok - correct signature
stats.standard = 'UK'; // error: type 'string' is not assignable to type 'number' - wrong signature
```

Type inference

```
// book has inferred type { title: string, author: string, year: number, category: string }
let book = {
  title: 'The Great Gatsby',
  year: 1925,
  category: 'Fiction'
};

book = { title: 'To Kill a Mockingbird', year: 1960, category: 'Fiction' }; // ok

book = {
  title: '1984',
  author: 'George Orwell',
  year: 1949
};
// error: property 'category' is missing
// error: property 'author' does not exist in type
```

Functions

Function parameter type annotations

```
// parameter type annotations
function greet(name: string) {
  console.log(`Hello, ${name}!`);
}

const alternateGreet = (name: string) => console.log(name); // as arrow function
```

Function return type annotations

```ts
// return type annotations
function multiply(a: number, b: number): number {
  return a * b;
}

const mul = (a: number, b: number): number => a * b; // as arrow function

// use void for functions with no documented return value
function speak(word: string): void {
  console.log(word);
}

const talk = (word: string): void => console.log(word); // as arrow function
```

Function parameters

```
// optional parameters
const fullGreet = (first: string, last?: string): void => {
  const name = last ? `${first} ${last}` : `${first}`;
  console.log(`Hello ${name}`);
};
fullGreet('sam');
fullGreet('sam', 'altman');

// default parameters
const defaultGreet = (first: string, last: string = 'unknown'): void => {
  console.log(`Hello ${first} ${last}`);
};
defaultGreet('sam');
defaultGreet('sam', 'altman');
```

Function parameters

```
// rest parameters are Array types
function sumWithMultiplier(multiplier: number, ...numbers: number[]): number {
  return multiplier * numbers.reduce((total, num) => total + num, 0);
}
console.log(sumWithMultiplier(2, 1, 2, 3)); // Output: 12
console.log(sumWithMultiplier(3, 4, 5, 6, 7)); // Output: 66
```

Union types

Union types

In TypeScript, union types allow to specify that a variable or parameter can have multiple types

A union type is defined using the | operator to separate the types

Union types

```
// defining a union type for a variable
let employeeId: number | string;
employeeId = 'S188D7LM';
employeeId = 1927599;
employeeId = false; // error: type 'boolean' is not assignable to type 'string | number'

// defining a union type for a function parameter
function printID(id: number | string): void {
  console.log(`ID: ${id}`);
}
printID(1927599); // output: "ID: 1927599"
printID('S188D7LM'); // output: "ID: S188D7LM"
```

Union types

```
// using a type guard to narrow down the type of a variable
function addOrConcat(a: number | string, b: number | string) {
  if (typeof a === 'number' && typeof b === 'number') {
    return a + b; // output type: number
  } else if (typeof a === 'string' && typeof b === 'string') {
    return a.concat(b); // output type: string
  } else {
    throw new Error('Invalid arguments');
  }
}
console.log(addOrConcat(1, 2)); // output: 3
console.log(addOrConcat('hello', 'world')); // output: "helloworld"
console.log(addOrConcat(2, 'world')); // error: Invalid arguments
```

Interface

Interface

In TypeScript, interfaces define a contract that describes the shape of an object by specifying the names and types of its properties

This feature enforces type-checking and ensures object conformity to a specific structure

Interfaces can be extended to create new interfaces that inherit the properties of the parent interface, making them even more powerful for writing maintainable code

Interface example

```
interface Entity {
  name: string;
  age: number;
}

// Object with specific properties
let entity: Entity = {
  name: 'John',
  age: 30
};

// Type checking
entity = { name: 'Mary', age: '40' }; // error: Type 'string' is not assignable to type 'number'
```

Interface used with functions

```
// Interface with specific properties
interface Person {
  name: string;
  age: number;
}

// Object that implements the Person interface
const john: Person = { name: 'John', age: 30 };

// Function that takes an object of type Person
function printPerson(person: Person) {
  console.log(`Name: ${person.name}, Age: ${person.age}`);
}

// Type checking
printPerson(john);
printPerson({ name: 'John', age: 30 });
printPerson({ name: 'Mary' }); // error: property 'age' is missing in type '{ name: string; }'
```

Interface with optional properties

```
// Object with optional properties using interface
interface Person {
  name: string;
  age?: number; // optional
}

let person: Person = {
  name: 'John'
};

// Type checking
person = { name: 'Ann', age: 34 }; // OK
person = { name: 'Mary', age: '40' }; // error: type 'string' is not assignable to type 'number'
```

Interface with readonly properties

```
// Object with readonly properties using interface
interface User {
  readonly name: string;
  age: number;
}

let user: User = {
  name: 'John',
  age: 30
};

// Type checking
user.name = 'Mary'; // error: cannot assign to 'name' because it is a read-only property
```

Interface with index signature

```
// Using an interface to define an object type with an index signature
interface Stats {
  [key: string]: number;
}

// Object with index signature defined by interface
let stats: Stats = {
  views: 1000,
  clicks: 200,
  conversions: 10
};

// Type checking
stats.distance = 200; // ok - correct signature
stats.standard = 'UK'; // error: type 'string' is not assignable to type 'number' - wrong signature
```

Interface and type inference

```ts
// Interface with specific properties
interface Person {
  name: string;
  age: number;
}

// Function that takes an object of type Person
function printPerson(person: Person) {
  console.log(`Name: ${person.name}, Age: ${person.age}`);
}

// Type inference for object literals
const james = { name: 'James', age: 14 };
printPerson(james); // james is not of type 'Person' but its shape matches 'Person'
const mary = { name: 'Mary', age: 40 } as Person;
printPerson(mary); // mary is of type 'Person'
```

See also: Duck typing - Wikipedia

Extending interfaces

Extending interfaces

TypeScript provides the ability to extend types with interfaces

This allows us to add additional properties or methods to an existing type definition, without modifying the original definition

Extending interfaces

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

interface Employee extends Person {
  id: number;
  department: string;
}

const john: Employee = {
  name: 'John',
  age: 30,
  id: 123,
  department: 'IT'
};
```

```
function createEmployee(name: string, age:
number, id: number, department: string): Employee
{
  return { name, age, id, department };
}

const jane: Employee = createEmployee('Jane', 25,
456, 'HR');

function printPerson(person: Person) {
  console.log(`Name: ${person.name}`);
  console.log(`Age: ${person.age}`);
}

printPerson(jane);
```

Extending multiple interfaces

```ts
interface Animal {
  species: string;
  legs: number;
}

interface Pet {
  name: string;
  owner: string;
}

interface Dog extends Animal, Pet {
  bark(): void;
}
```

```ts
const fido: Dog = {
  species: 'Canis familiaris',
  legs: 4,
  name: 'Fido',
  owner: 'Jane',
  bark: () => console.log('Woof!')
};
```

Type aliases

Type aliases

A type alias is a way to create a new name for an existing type

It allows to give a descriptive name to a complex or custom type, which can make code easier to read and understand

They are similar to interfaces, but can also be used with primitive types or to define a set of string or number literals

Type aliases

```
// alias for primitive types
type MyNumber = number;
let x: MyNumber = 42;
let y: number = x; // this is allowed, x and y are the same type

// alias type unions
type Id = number | string;
let employeeId: Id = 'S188D7LM';
employeeId = 1927599;
employeeId = false; // error: type 'boolean' is not assignable to type 'Id'

// a slightly more complex type union
type StringLike = string | (() => string);
let friend: StringLike = 'sam';
let getFriend: StringLike = () => {
  return 'adam';
};
```

Type aliases - literals

```
// alias a set of string or number literals
type LuckyNumbers = 18 | 27 | 333;
type Direction = 'up' | 'down' | 'left' | 'right';
let going: Direction = 'down';
let heading: Direction = 'sideways'; // error: Type '"sideways"' is not assignable to type 'Direction'
```

Type aliases - object types

```ts
// alias Object types (similar to interface)
type Person = {
  name: string;
  readonly age: number; // readonly
};

// intersection '&' (similar to extend but with differences)
type Employee = Person & {
  id: number;
  department?: string; // optional
};

const john: Employee = {
  name: 'John',
  age: 30,
  id: 123,
  department: 'IT'
};
```

Type aliases - object types

```ts
// alias Object types (similar to interface)
type Point = {
  x: number;
  y: number;
};
const p: Point = { x: 10, y: 10 };

// intersection '&' (similar to extend but with differences)
type Point3D = Point & {
  z: number;
};
const p3: Point3D = { x: 2, y: 17, z: 21 };


// could have also done
type HasZAxis = { z: number; };
type Point3D = Point & HasZAxis; // intersection
const p3: Point3D = { x: 2, y: 17, z: 21 };
```

Type aliases - functions

```
// using Type aliases with functions
const transform = (p: Point | Point3D): Point => {
  // flatten p then
  return p;
};

type SuperPoint = Point | Point3D;
const altTransform = (p: SuperPoint): Point => {
  // do something with p then
  return p;
};
```

Type aliases - functions

```
// can define types for functions
type PointTransformer = (p: Point | Point3D) => Point;

// transform2D is a PointTransformer that accepts a Point
const transform2D: PointTransformer = (p: Point): Point => {
  // do something with p then
  return p;
};

// transform3D is a PointTransformer that accepts a SuperPoint = Point | Point3D
const transform3D: PointTransformer = (p: SuperPoint): Point => {
  // do something with p then
  return p;
};
```

Type casting

Also called type assertions

Type casting

Type casting using the 'as' keyword is a way to tell the compiler that a value should be treated as a different type than its original type

This can be useful when working with APIs or third-party libraries that may return data in unexpected formats but it can be dangerous if used incorrectly, overriding the type system and leading to errors

TypeScript calls type casting "type assertions"

Type casting

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

const data = '{"name": "John", "age": 30}'; // data from an external service: API, library, etc.
const john = JSON.parse(data) as Person;

console.log(john.name); // "John"
console.log(john.age); // 30
```

Enums

Enums

An enum in TypeScript is a data type that allows for defining a set of named constants

It is a way to give more meaning to values in code by assigning them names, making them more readable and maintainable

The values in an enum are usually integers, but they can also be strings
Enums are also useful because most editors provide autocompletion of values

Numeric example

```
enum DayOfWeek {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday
}

console.log(DayOfWeek.Monday); // Output: 0
console.log(DayOfWeek.Friday); // Output: 4
console.log(DayOfWeek.Sunday); // Output: 6

// enum values are assigned automatically if not initialized
// numeric enums start at 0 and increase by 1 for each following value
```

Initialized numeric enum

```
// fully initialized enum
enum StatusCode {
  OK = 200,
  NotFound = 404,
  ServerError = 500
}

console.log(StatusCode.OK); // Output: 200
console.log(StatusCode.NotFound); // Output: 404
console.log(StatusCode.ServerError); // Output: 500
```

String enum

```
// string enum
enum LogLevel {
  Error = 'ERROR',
  Warn = 'WARN',
  Info = 'INFO',
  Debug = 'DEBUG'
}

console.log(LogLevel.Error); // Output: 'ERROR'
console.log(LogLevel.Info); // Output: 'INFO'
console.log(LogLevel.Debug); // Output: 'DEBUG'
```

Note: enums should be numeric or string; don't mix types if you don't have a good reason

Partially initialized enum

```
// partial enum
enum Feeling {
  Happy,
  Sad = 12,
  Bored,
  Giddy
}

console.log(Feeling.Happy); // Output: 0
console.log(Feeling.Sad); // Output: 12
console.log(Feeling.Bored); // Output: 13
```

Generics

Generics

The usage of generics facilitates the creation of flexible and reusable code by defining placeholders for types that can be specified later

This means that functions, classes, and interfaces can operate with various types without the need to specify them directly

TypeScript: Documentation - Generics

Generics

TypeScript uses generics for example for arrays using the syntaxes **type[]** and **Array<type>**

- Array<string> and string[] are used to declare an array of strings
- Array<number> and number[] are used to declare an array of numbers
- string[] and number[] are more commonly used because they are easier to read

```
// Array<string>
const fruits1: Array<string> = ['apple', 'banana', 'orange'];

// string[]
const fruits2: string[] = ['apple', 'banana', 'orange'];

// Array<number>
const numbers1: Array<number> = [1, 2, 3, 4];

// number[]
const numbers2: number[] = [1, 2, 3, 4];
```

Generic function

```
function identity<T>(arg: T): T {
  return arg;
}

// usage
let output = identity<string>('hello');
console.log(output); // "hello"

let output2 = identity<number>(5);
console.log(output2); // 5
```

Generic function with interface

```
function identity<T>(arg: T): T {
  return arg;
}

interface Identity<T> {
  (arg: T): T;
}

// specific type functions based on the generic
const stringIdentityFn: Identity<string> = identity;
const numberIdentityFn: Identity<number> = identity;

console.log(stringIdentityFn('example')); // Output: "example"
console.log(numberIdentityFn(42)); // Output: 42
```

Generic interface

```
interface Container<T> {
  value: T | null;
  getValue(): T | null;
  setValue(value: T): void;
}

let sam: Container<string> = {
  value: null,
  setValue(v: string): void {
    this.value = v;
  },
  getValue() {
    return this.value;
  }
};

sam.setValue('altman');
console.log(sam.getValue());
```

Generic class

```
class ContainerClass<T> implements Container<T> {
  value: T | null;

  constructor(value?: T) {
    this.value = value || null;
  }

  setValue(value: T): void {
    this.value = value;
  }

  getValue(): T | null {
    return this.value;
  }
}
```

```
// using the same interface as before
interface Container<T> {
  value: T | null;
  getValue(): T | null;
  setValue(value: T): void;
}
```

Generic class

```
// example with type string and constructor parameter
let sam: Container<string> = new ContainerClass<string>('altman');
console.log(sam.getValue()); // "altman"

// example with type number and no constructor parameter
let numberBox: Container<number> = new ContainerClass<number>();
numberBox.setValue(42);
console.log(numberBox.getValue()); // 42
```

Utility Types

Utility types are built-in type operators that allow manipulating and transforming existing types in various ways

A few examples

- Awaited<Type>
- Partial<Type>
- Required<Type>
- Readonly<Type>

TypeScript: Documentation - Utility Types

Your turn

1. Rewrite

- Create a basic typescript project that contains
  - src/ dist/ folders
  - a tsconfig.json file
  - a main.ts file
  - an index.html file that links to the compiled .js file in the dist/ folder

- Choose any of your previous Javascript exercises or projects
  - Rewrite it using typescript
  - Think where typescript can improve the safety of your code or simplify it
  - Compile your code and make sure it works like the original version, or maybe better

References

Documentation - Everyday Types
Documentation - Object Types
Documentation - More on Functions
Handbook - Enums

References

Unions and Intersections
Classes
Documentation - Generics