TypeScript

TypeScript Classes: -

TypeScript adds types and visibility modifiers to JavaScript classes.

Members: Types

The members of a class (properties & methods) are typed using type annotations, similar to variables.

```
class Person {
  name: string;
}
const person = new Person();
person.name = "Jane";
```

Members: Visibility

Class members also be given special modifiers which affect visibility.

public - (default) allows access to the class member from anywhere

private - only allows access to the class member from within the class

protected - allows access to the class member from itself and any classes that inherit it.

```
class Person {
  private name: string;
  public constructor(name: string) {
    this.name = name;
  }
  public getName(): string {
    return this.name;
  }
}
const person = new Person("Jane");
console.log(person.getName()); // person.name isn't accessible from outside the class since it's private.
```

Parameter Properties: -

TypeScript provides a convenient way to define class members in the constructor, by adding a visibility modifiers to the parameter.

```
class Person {
  // name is a private member variable
  public constructor(private name: string) {}
  public getName(): string {
    return this.name;
  }
}
const person = new Person("Jane");
console.log(person.getName());
```

Like arrays, the **readonly** keyword can prevent class members from being changed.

```
class Person {
  private readonly name: string;
  public constructor(name: string) {
    // name cannot be changed after this initial definition, which has to be either at it's declaration or in the constructor.
    this.name = name;
  }
  public getName(): string {
    return this.name;
  }
}
const person = new Person("Jane");
console.log(person.getName());
```

Inheritance: Implements

```
interface Shape {
  getArea: () => number;
}
class Rectangle implements Shape {
  public constructor(protected readonly width: number, protected readonly height: number) {}

  public getArea(): number {
```

```
    return this.width * this.height;
  }
}
const myRect = new Rectangle(10,20);
console.log(myRect.getArea()); //200
```

A class can implement multiple interfaces by listing each one after implements, separated by a comma.
ex: - class Rectangle implements Shape, Colored {

Inheritance: Extends

Classes can extend each other through the extends keyword. A class can only extends one other class.

```
interface Shape {
  getArea: () => number;
}
class Rectangle implements Shape {
  public constructor(protected readonly width: number, protected readonly height: number) {}
  public getArea(): number {
    return this.width * this.height;
  }
}
class Square extends Rectangle {
  public constructor(width: number) {
    super(width, width);
  }
  // getArea gets inherited from Rectangle
}
const mySq = new Square(20);
console.log(mySq.getArea()); // 400
```

Override: -

When a class extends another class, it can replace the members of the parent class with the same name.

Newer versions of TypeScript allow explicitly marking this with the override keyword.

```
interface Shape {
  getArea: () => number;
}
class Rectangle implements Shape {
  // using protected for these members allows access from classes that extend from this class, such as Square
  public constructor(protected readonly width: number, protected readonly height: number) {}
  public getArea(): number {
    return this.width * this.height;
  }
  public toString(): string {
    return `Rectangle[width=${this.width}, height=${this.height}]`;
  }
}
class Square extends Rectangle {
  public constructor(width: number) {
    super(width, width);
  }
  // this toString replaces the toString from Rectangle
  public override toString(): string {
    return `Square[width=${this.width}]`;
  }
}
```

By default the override keyword is optional when overriding a method, and only helps to prevent accidentally overriding a method that does not exist. Use the setting noImplicitOverride to force it to be used when overriding.

Abstract Classes: -

Classes can be written in a way that allows them to be used as a base class for other classes without having to implement all the members. This is done by using the abstract keyword. Members that are left unimplemented also use the abstract keyword.

```
abstract class Polygon {
  public abstract getArea(): number;
```

```
  public toString(): string {
    return `Polygon[area=${this.getArea()}]`;
  }
}
class Rectangle extends Polygon {
  public constructor(protected readonly width: number, protected readonly height: number) {
    super();
  }
  public getArea(): number {
    return this.width * this.height;
  }
}
```

Abstract classes cannot be directly instantiated, as they do not have all their members implemented.

TypeScript Basic Generics: -

Generics allow creating 'type variables' which can be used to create classes, functions & type aliases that don't need to explicitly define the types that they use.

Generics makes it easier to write reusable code.

Functions

Generics with functions help make more generalized methods which more accurately represent the types used and returned.

```
function createPair<S, T>(v1: S, v2: T): [S, T] {
  return [v1, v2];
}
console.log(createPair<string, number>('hello', 42)); // ['hello', 42]
```

Classes

```
class NamedValue<T> {
  private _value: T | undefined;
  constructor(private name: string) {}
  public setValue(value: T) {
    this._value = value;
  }
```

```
  public getValue(): T | undefined {
    return this._value;
  }
  public toString(): string {
    return `${this.name}: ${this._value}`;
  }
}
let value = new NamedValue<number>('myNumber');
value.setValue(10);
console.log(value.toString()); // myNumber: 10
```

Type Aliases

Generics in type aliases allow creating types that are more reusable.

```
type Wrapped<T> = { value: T };
const wrappedValue: Wrapped<number> = { value: 10 };
```

This also works with interfaces with the following syntax: interface Wrapped<T> {

Default Value

Generics can be assigned default values which apply if no other value is specified or inferred.

```
class NamedValue<T = string> {
  private _value: T | undefined;
  constructor(private name: string) {}
  public setValue(value: T) {
    this._value = value;
  }
  public getValue(): T | undefined {
    return this._value;
  }
  public toString(): string {
    return `${this.name}: ${this._value}`;
  }
}
```

```
let value = new NamedValue('myNumber');

value.setValue('myValue');

console.log(value.toString()); // myNumber: myValue
```

Extends

Constraints can be added to generics to limit what's allowed. The constraints make it possible to rely on a more specific type when using the generic type.

```
function createLoggedPair<S extends string | number, T extends string | number>(v1: S, v2: T): [S, T] {
  console.log(`creating pair: v1='${v1}', v2='${v2}'`);
  return [v1, v2];
}
```

This can be combined with a default value.

TypeScript Utility Types: -

TypeScript comes with a large number of types that can help with some common type manipulation, usually referred to as utility types.

Partial

Partial changes all the properties in an object to be optional.

```
interface Point {
  x: number;
  y: number;
}

let pointPart: Partial<Point> = {}; // `Partial` allows x and y to be optional

pointPart.x = 10;
```

Required

Required changes all the properties in an object to be required.

```
interface Car {
  make: string;
  model: string;
  mileage?: number;
}

let myCar: Required<Car> = {
  make: 'Ford',
```

```
  model: 'Focus',
  mileage: 12000 // `Required` forces mileage to be defined
};
```

Record

Record is a shortcut to defining an object type with a specific key type and value type.

```
const nameAgeMap: Record<string, number> = {
  'Alice': 21,
  'Bob': 25
};
```

Record<string, number> is equivalent to { [key: string]: number }

Omit

Omit removes keys from an object type.

```
interface Person {
  name: string;
  age: number;
  location?: string;
}
const bob: Omit<Person, 'age' | 'location'> = {
  name: 'Bob'
  // `Omit` has removed age and location from the type and they can't be defined here
};
```

Pick

Pick removes all but the specified keys from an object type.

```
interface Person {
  name: string;
  age: number;
  location?: string;
}
const bob: Pick<Person, 'name'> = {
  name: 'Bob'
  // `Pick` has only kept name, so age and location were removed from the type and they can't be defined here
```

```
};
```

Exclude

Exclude removes types from a union.

```
type Primitive = string | number | boolean
const value: Exclude<Primitive, string> = true; // a string cannot be used here since Exclude removed it from the type.
```

ReturnType

ReturnType extracts the return type of a function type.

```
type PointGenerator = () => { x: number; y: number; };
const point: ReturnType<PointGenerator> = {
  x: 10,
  y: 20
};
```

Parameters

Parameters extracts the parameter types of a function type as an array.

```
type PointPrinter = (p: { x: number; y: number; }) => void;
const point: Parameters<PointPrinter>[0] = {
  x: 10,
  y: 20
};
```

Readonly

Readonly is used to create a new type where all properties are readonly, meaning they cannot be modified once assigned a value.

TypeScript will prevent this at compile time, but in theory since it is compiled down to JavaScript you can still override a readonly property.

```
interface Person {
  name: string;
  age: number;
}
const person: Readonly<Person> = {
  name: "Dylan",
  age: 35,
};
person.name = 'Israel'; // prog.ts(11,8): error TS2540: Cannot assign to 'name' because it is a read-only property.
```

TypeScript Keyof: -

keyof is a keyword in TypeScript which is used to extract the key type from an object type.

keyof with explicit keys

When used on an object type with explicit keys, keyof creates a union type with those keys.

```
interface Person {
  name: string;
  age: number;
}
// `keyof Person` here creates a union type of "name" and "age", other strings will not be
allowed
function printPersonProperty(person: Person, property: keyof Person) {
  console.log(`Printing person property ${property}: "${person[property]}"`);
}
let person = {
  name: "Max",
  age: 27
};
printPersonProperty(person, "name"); // Printing person property name: "Max"
```

keyof with index signatures

keyof can also be used with index signatures to extract the index type.

```
type StringMap = { [key: string]: unknown };
// `keyof StringMap` resolves to `string` here
function createStringPair(property: keyof StringMap, value: string): StringMap {
  return { [property]: value };
}
```

TypeScript Null & Undefined

By default null and undefined handling is disabled, and can be enabled by setting strictNullChecks to true.

null and undefined are primitive types and can be used like other types, such as string.

```
let value: string | undefined | null = null; // object
value = 'hello'; // string
value = undefined; // undefined
```

When strictNullChecks is enabled, TypeScript requires values to be set unless undefined is explicitly added to the type.

Optional Chaining: -

Optional Chaining is a JavaScript feature that works well with TypeScript's null handling. It allows accessing properties on an object, that may or may not exist, with a compact syntax. It can be used with the ?. operator when accessing properties.

```
interface House {

  sqft: number;

  yard?: {
```

```
    sqft: number;
  };
}
function printYardSize(house: House) {
  const yardSize = house.yard?.sqft;
  if (yardSize === undefined) {
    console.log('No yard');
  } else {
    console.log(`Yard is ${yardSize} sqft`);
  }
}
let home: House = {
  sqft: 500
};
printYardSize(home); // Prints 'No yard'
```

Nullish Coalescence: -

Nullish Coalescence is another JavaScript feature that also works well with TypeScript's null handling. It allows writing expressions that have a fallback specifically when dealing with null or undefined.

This is useful when other falsy values can occur in the expression but are still valid. It can be used with the ?? operator in an expression, similar to using the && operator.

```
function printMileage(mileage: number | null | undefined) {
  console.log(`Mileage: ${mileage ?? 'Not Available'}`);
}
printMileage(null); // Prints 'Mileage: Not Available'
printMileage(0); // Prints 'Mileage: 0'
```

Null Assertion: -

TypeScript's inference system isn't perfect, there are times when it makes sense to ignore a value's possibility of being null or undefined.

An easy way to do this is to use casting, but TypeScript also provides the ! operator as a convenient shortcut.

```
function getValue(): string | undefined {
  return 'hello';
}
let value = getValue();
console.log('value length: ' + value!.length);
```

Array bounds handling: -

Even with strictNullChecks enabled, by default TypeScript will assume array access will never return undefined (unless undefined is part of the array type).

The config noUncheckedIndexedAccess can be used to change this behavior.

```
let array: number[] = [1, 2, 3];
let value = array[0]; // with `noUncheckedIndexedAccess` this has the type `number | undefined`
```

TypeScript Definitely Typed: -

NPM packages in the broad JavaScript ecosystem doesn't always have types available.

Using non-typed NPM packages in TypeScript

Using untyped NPM packages with TypeScript will not be type safe due to lack of types.

Definitely Typed is a project that provides a central repository of TypeScript definitions for NPM packages which do not have types.

Ex: - npm install --save-dev @types/jquery

This command installs the TypeScript type definitions for jQuery, allowing you to use jQuery in a TypeScript project with type safety and autocompletion features.

Editors such as Visual Studio Code will often suggest installing packages like these when types are missing.

Template Literal Types: -

Template Literal Types now allows us to create more precise types using template literals. We can define custom types that depend on the actual values of strings at compile time.

```
type Color = "red" | "green" | "blue";

type HexColor<T extends Color> = `#${string}`;

// Usage:

let myColor: HexColor<"blue"> = "#0000FF";
```

Index Signature Labels: -

Index Signature Labels allows us to label index signatures using computed property names. It helps in providing more descriptive type information when working with dynamic objects.

```
type DynamicObject = { [key: string as `dynamic_${string}`]: string };

let obj: DynamicObject = { dynamic_key: "value" }; // Usage
```