Top 100 Typescript Interview Questions

You can also find all 100 answers here 👉 Devinterview.io - Typescript

1. What is *TypeScript* and how does it differ from *JavaScript*?

TypeScript is a statically-typed superset of **JavaScript**, developed and maintained by Microsoft. It enables enhanced code maintainability and predictability. After compiling, TypeScript code is transpiled into standard, browser-compatible JavaScript.

Key distinctions between TypeScript and JavaScript include the use of type annotations, the ability to work with existing JavaScript code, and more.

TypeScript Features & Benefits

- **Type System**: Offers static typing, allowing developers to define the type of variables, parameters, and return values. This helps catch errors during development, reducing runtime issues.
- **Advanced Language Features**: Incorporates modern ECMAScript syntax and features, often before they are rolled out in JavaScript. Additionally, TypeScript brings functional programming patterns, classes, and access modifiers (such as `public` and `private`)
- **Compatibility with JavaScript**: TypeScript can interpret existing JavaScript code with minor or no modifications.
- **Tooling and Extra Safety**: Provides enhanced autocompletion, refactoring, and documentation via TypeScript-aware tools. TypeScript helps catch and rectify common programming errors without needing to run the code.
- **ECMAScript Compatibility**: TypeScript allows developers to target different **ECMAScript versions**, ensuring the generated JavaScript is compatible with the targeted browsers.

- **Code Structure & Readability**: Promotes maintainability by enforcing a defined coding structure and fostering code clarity.

TypeScript's Role in Modern Development

- **Workplace Adaptability**: TypeScript is used in an extensive range of projects, from small utilities to large-scale applications.
- **Community Support**: Supported by a vibrant developer community, TypeScript benefits from frequent updates, bug fixes, and useful extensions.
- **On-Going Development**: A robust language server furnishes accurate tooling feedback, such as linting and error suggestions in real time.
- **Rapid Enhancement**: The TypeScript team consistently introduces new features and reinforces existing ones.

2. Can you explain what is meant by "*TypeScript* is a *superset* of *JavaScript*"?

TypeScript is often described as a "**superset of JavaScript**" because every valid JavaScript code is also a valid TypeScript code.

TypeScript is designed in a way that it fully embraces existing **JavaScript syntax** and functionality. This ensures a smooth transition for developers wishing to adopt or migrate to TypeScript.

Key TypeScript Features On Top of JavaScript

- **Type Definitions**: TypeScript introduces static typing through type annotations. These are optional, enabling gradual adoption for existing codebases that might not need them.
- **Newer JavaScript Features**: TypeScript extends JavaScript syntax, providing support for the latest ECMAScript standards more effectively through its compiler, even when the underlying JavaScript engine might not support them yet.
- **Tooling and Error Detection**: TypeScript offers robust type-checking, increased code readability, and stronger compile-time error detection.

Code Demonstration

Here is the TypeScript code:

```
let num: number = 5;
num = "this will raise a type error";
```

3. What are the *basic types* available in *TypeScript*?

TypeScript provides an assortment of basic types for different kinds of data, such as numbers, strings, boolean values, arrays, tuples and more.

Common Basic Types in TypeScript

- **Boolean**: Represents true/false values.
- **Number**: Applies to both integer and floating-point numbers.
- **String**: Refers to textual data.
- **Array**: Offers a flexible way to work with structured data.
- **Tuple**: Enables the definition of arrays with a fixed number of elements, each potentially of a different data type.
- **Enum**: Provides a set of named constants such as days or colors.
- **Any**: Offers a dynamic type, which can be used to bypass type-checking. It's typically best to be avoided, as it defeats the purpose of using TypeScript, which is primarily focused on static typing. However, there are certain use cases where it becomes necessary.
- **Void**: Typically used as the return type for functions that don't return a value.

- **Null** and **Undefined**: Allow for the assignment of null and undefined values, respectively. However, this isn't enabled by default, and these are probably better handled using the `strict` mode settings in TypeScript.
- **Never**: Represents the type of values that never occur. For instance, the return type of a function that doesn't reach its end or always throws an error.
- **Object**: Any JavaScript object.
- **Function**: Denotes a function type.

Code Example: Basic TypeScript Types

Here is the TypeScript code:

```
// Boolean
let isActive: boolean = true;

// Number
let age: number = 30;

// String
let title: string = "Manager";

// Array
let scores: number[] = [85, 90, 78];
// or use a compact form: let scores: Array<number> = [85, 90, 78];

// Tuple
let employee: [string, number, boolean] = ['John', 35, true];

// Enum
enum WeekDays { Monday, Tuesday, Wednesday, Thursday, Friday }
let today: WeekDays = WeekDays.Wednesday;

// Any
let dynamicData: any = 20;

// Void
function greet(): void {
  console.log("Hello!");
}

// Null and Undefined
let data: null = null;
let user: undefined = undefined;

// Never
function errorMessage(message: string): never {
  throw new Error(message);
}

// Object
let person: object = {
  name: 'John',
  age: 30
};

// Function
let calculate: Function;
calculate = function (x: number, y: number): number {
  return x + y;
};
```

4. How do you declare *variables* in *TypeScript*?

In TypeScript, **variable declarations** support different methodologies for declaring variables and their associated types.

Variable and Type Declaration Methods

1. var

```
var score: number = 100;
```

This declaration can lead to **variable hoisting** and has global scope or function-level scope.

2. let

Use `let` when you want to define variables within a **block scope**. This is the recommended default choice:

```
let playerName: string = "John";
```

3. const

`const` allows you to declare **constants** and is especially useful for maintaining data integrity:

```
const apiKey: string = "your-api-key-here";
```

4. Function Scope

All three methods (`var` , `let` , and `const`) are confined to their immediate function scope:

```
function doSomething() {
  let tempValue: number = 42;
  var result: boolean = true;
}
```

Rules for Variable Declaration and Initialization

- **Order Matters**: In TypeScript, a variable must be declared before being used. This is not a requirement in JavaScript, but good JavaScript practice is to declare a variable before using it.

  If you're dealing with complex or interconnected codes, it's a good practice to use the `let` and `const` declarations that ensure the block-level scoping, thus helping with potential hoisting issues.

- **Static Types**: TypeScript requires that you declare the data type of a variable (or let the system infer it) and then initialize it with a value of exactly the same type:

  ```
  let count: number;  // Declaration
  count = 42;  // Allowed
  count = "42";  // Error! Type 'string' is not assignable to type 'number'.
  ```

- **Type Inference**: TypeScript can often infer the variable's type based on its initialization value. This reduces the need to specify a type explicitly.

  ```
  let word = "hello!";  // TypeScript infers the type as 'string' because of the initialization.
  ```

Best Practices for Variable Declarations

- **Use `const` Where You Can**: This approach isn't always possible, especially when dealing with object properties. However, favor `const` for better code readability and to prevent accidental data mutations.
- **Prefer `let` over `var`** : `let` adheres better to **block-level scoping** and offers more predictability in the code.
- **Initialize at Declaration**: Although TypeScript allows initializations after declarations, it's best to declare and initialize variables simultaneously to improve code clarity and type safety.

- **Prefer Type Annotations**: Explicitly specifying variable types can improve code readability. However, when the variable type is obvious from the initialization, type inference should suffice.

5. What are *Interfaces* in *TypeScript* and how do they work?

In TypeScript, an **interface** defines the structure and types of its members. It acts as a contract for the required properties and methods, ensuring that implementing classes or objects match this structure.

Key Features of Interfaces

- **Type Consistency**: Objects that adhere to an interface's structure are considered compatible with it.
- **Optional and Readonly Members**: Interfaces allow for optional attributes and readonly members with the `?` and `readonly` keywords respectively.
- **Call Signatures**: Interfaces can define method types, specifying function parameter and return types.
- **Index Signatures**: Useful for specifying that an object can have any number of properties, all of a certain type.

Core Use-Cases

- **Standardizing Objects**: Ensuring that disparate objects share a common structure for increased cohesiveness and ease of use.
- **Contract Enforcement**: Enforcing property and method requirements on classes to reduce errors and improve maintainability.

Code Example: Basic Interface

Here is the TypeScript code:

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

function printPoint(p: Point) {
    console.log(`Point coordinates: (${p.x}, ${p.y})`);
}

let pointA = { x: 3, y: 7 }; // This object matches Point's structure
let pointB = { x: 8 }; // This object is missing the 'y' property

printPoint(pointA); // Output: Point coordinates: (3, 7)
printPoint(pointB); // Compile-time error due to incorrect structure
```

6. Describe the *Enum* type and when you might use it.

The **Enum** is a data type that simplifies the representation and management of discrete, named values. It's a foundational tool to ensure **type safety** in TypeScript and a number of vital use-cases:

- **Reducing 'Magic Values'**: When ensuring readability and preventing repetitive literal values, such as `1`, `2`, or `'red'`.
- **Configuring Behaviour**: influencing functionalities sets of associated values, such as HTTP methods, ordering or customer types.
- **Ensuring Type Safety and Efficiency**: The predefined set of valid members and a clear data type ensures that value assignments and operations are unequivocal and consistent.

Core Components

- **Key**: a unique identifier, typically a number or string.
- **Value**: Data associated with the key. If not provided, the key is used as the value.

Standard, String, and Heterogeneous Enums

- **Standard Enum**: Every key and value are of the same data type, typically numbers.

- **String Enum**: All keys and values must be strings, ensuring consistent data representation.
- **Heterogeneous Enum**: Defines keys with both number or string values. However, due to the mixed-type nature of these enums, it's best to steer clear of them in most cases.

Code Example: Standard Enum

Here is the TypeScript code:

```
enum HttpMethods {
  GET,
  POST,
  PUT,
  DELETE
}

const requestType: HttpMethods = HttpMethods.GET;

// ❌ This is not allowed due to type safety
// const requestType2: HttpMethods = 'GET';
```

In the example, the key `GET` is implicitly assigned the value `0`.

Code Example: String Enum

Here is the TypeScript code:

```
enum MediaTypes {
  Image = 'image',
  Video = 'video',
  Audio = 'audio'
}

const selectedType: MediaTypes = MediaTypes.Image;

// ❌ This is not allowed due to type safety
// const selectedType2: MediaTypes = MediaTypes.Video;

// ✅ Accessing the value
const associatedText: string = MediaTypes.Image;

// ❌ This is not allowed due to type safety
// const invalidType: MediaTypes = 'image';
```

the Enum helps ensure the proper data type and its values.

Pragmatic Use of Enums

While Enums are a powerful tool for maintaining type safety, simplify associating related sets of values.

However, a consideration is that an Enum value can be inferred or forced to be of any key and underlying value type.

Potential Downsides of Enums

- **Compilation Impact**: When used in a broader context, or in data structures like arrays or maps, TypeScript generates additional code to convert Enum keys to their associated values.
- **Memory Usage**: Every usage of an Enum requires memory allocation for its value.

When a simple constant would suffice or if there's a need for a more dynamic relationship between keys and values, detailed types would be a better alternative.

7. How do you define and use a *function* in *TypeScript*?

When defining a **function** in TypeScript, you have the following fundamental components to consider:

- **Function Signature**: Comprising the function's purpose, parameters, type, and return value.

- **Function Body**: Containing the actual operation or series of steps the function will execute.

Key Concepts

1) Function Declaration

To declare a function, you specify its name, its parameter list, and its return type. If the function doesn't return a value, you set the return type to `void`.

Here is a code example:

```
function greet(name: string): void {
    console.log(`Hello, ${name}!`);
}
```

2) Function Expression

You can also declare functions using expressions, which involve assigning functions to variables as values. This approach allows you to be more flexible, such as when you're using **callbacks**.

Here is an example:

```
let greet: (name: string) => void;
greet = function(name: string): void {
    console.log(`Hello, ${name}!`);
};
```

3) Optional and Default Parameters

TypeScript supports both **optional** and **default** function parameters, enhancing the flexibility of your functions.

Optional Parameters are denoted by a `?` symbol after the parameter name.

Here is the code example:

```
function greet(name: string, title?: string) {
    if (title) {
        console.log(`Hello, ${title} ${name}!`);
    } else {
        console.log(`Hello, ${name}!`);
    }
}
```

Default Parameters are when you assign a default value to a parameter:

Here is the code example:

```
function greet(name = "Stranger") {
    console.log(`Hello, ${name}!`);
}
```

4) Use Rest Parameters

You can define a parameter as a "rest" parameter, which means the function can accept any number of arguments for that parameter.

Here is the code example:

```
function introduce(greeting: string, ...names: string[]) {
    console.log(`${greeting}, ${names.join(", ")}!`);
}

introduce("Hello", "Alice", "Bob", "Carol");
```

5) Function Overloads

You can declare multiple function **overloads** to define a set of parameters and their return types for a single function. This feature is especially beneficial when the function's behavior logically varies based on different input types.

Here is the code example:

```
function specialGreet(name: string): void;
function specialGreet(title: string, name: string): void;

function specialGreet(a: any, b?: any): void {
    if (b) {
        console.log(`Hello, ${a}, ${b}`);
    } else {
        console.log(`Hello, ${a}`);
    }
}
```

6) Call Signature

When using objects in TypeScript, you have the **call signature** to define the expected function structure for a specific method within the object.

Here is a code example:

```
type Greeter = {
    (name: string): void
};

let welcome: Greeter;
welcome = function(name: string): void {
    console.log(`Welcome, ${name}!`);
};
```

8. What does "*type inference*" mean in the context of *TypeScript*?

In TypeScript, **type inference** is a core feature that allows the type of a variable to be automatically determined from its value. This provides the benefits of static typing without the need for explicit type annotations.

How It Works

TypeScript employs a **best common type** algorithm to infer a variable's type. When TypeScript encounters multiple types for a variable during assignment or an array literal, it computes the **union** of these types and selects the best common type for the variable.

Code Example: Type Inference

Consider the following code:

```
let value = 10; // Type 'number' inferred
let message = "Hello, TypeScript!"; // Type 'string' inferred

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

let sum = add(5, 7); // Type 'number' inferred
```

TypeScript can infer the most likely type from the context, such as:

- When a value is assigned immediately, TypeScript assigns the value's type to the variable.
- Type information from adjacent types is used to determine the best common type. If all values are of a compatible type, that type is used.

Benefits of Type Inference

- **Conciseness**: Eliminates the need for explicit type declarations, leading to more compact and readable code.
- **Adaptability**: Codebase types align naturally with values, enhancing maintainability when values change.
- **Error Reduction**: Reduces the risk of inconsistencies between the declared type and the actual value.

9. Explain the use of '*let*' and '*const*' in *TypeScript*.

TypeScript makes use of `const` and `let` for variable declaration. These two keywords offer explicitness, scoping, and immutability for efficient code maintenance.

Core Distinctions

- **const**: Designates constants that remain unchanged once declared. It's important to note that this makes the reference immutable but doesn't actively prevent alteration of the internal state for complex objects like arrays.
- **let**: Initiates variables with standard mutable behavior.

Code Example: `const`

Here is the TypeScript code:

```
const productId: number = 5;
let productName: string = 'Tesla';

const getProductDetails = (id: number): string => {
  return `Product ID: ${id}`;
};

// Attempting to modify will result in a compilation error
// productId = 6;

// Reference is still immutable
const anotherProductId: number = 10;
// This will throw a compilation error since it's a constant
// anotherProductId = 12;

// Modifying internal state of an object is allowed for a const
const myArray: number[] = [1, 2, 3];
myArray.push(4);
```

Code Example: `let`

Here is the TypeScript code:

```
let vehicleType: string = 'Car';

if (true) {
  let vehicleType: string = 'Motorcycle';
  console.log(vehicleType);  // Output: Motorcycle
}

console.log(vehicleType);  // Output: Car
```

10. How do you compile *TypeScript* files into *JavaScript*?

Compiling **TypeScript** (.ts) into **JavaScript** (.js) involves integrating a TypeScript compiler (`tsc`). You can customize the compilation process using `tsconfig.json` and even adopt more advanced methods to suit project needs:

Workflow Steps

1. **File Creation**: Write TypeScript files (.ts).

2. **Compiler Config**: Set up a `tsconfig.json` file with compilation options.
3. **Compile**: Execute the `tsc` command to initiate the compilation process.
4. **Output Verification**: Review the generated JavaScript files.

TypeScript Configuration (`tsconfig.json`)

Here is the `tsconfig.json` file. The full configuration guide is available here.

```
{
  "compilerOptions": {
    "target": "ES5",
    "module": "commonjs",
    "strict": true,
    "outDir": "dist",
    "rootDir": "src"
  },
  "include": [
    "src/**/*.ts"
  ],
  "exclude": [
    "node_modules",
    "**/*.spec.ts"
  ]
}
```

Practical Example: Vineyard Residential Task Management App

Here is a practical and comprehensive `tsconfig.json` file.

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "lib": ["dom", "es2015", "es5", "es6", "es7", "es2015.collection"],
    "allowJs": true,
    "checkJs": false,
    "jsx": "react",
    "declaration": false,
    "sourceMap": true,
    "outDir": "dist",
    "rootDir": "src",
    "strict": true,
    "noImplicitAny": true,
    "noImplicitThis": true,
    "moduleResolution": "node",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "removeComments": true,
    "suppressImplicitAnyIndexErrors": true,
    "typeRoots": ["node_modules/@types", "custom-typings"],
    "baseUrl": ".",
    "paths": {
      "components/*": ["src/components/*"],
      "utils/*": ["src/utils/*"],
    },
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "incremental": true,
    "diagnostics": true,
    "resolveJsonModule": true,
    "isolatedModules": true,
    "newLine": "LF",
    "watchOptions": {
      "watchFile": "useFsEvents",
      "fallbackPolling": "dynamicPriority",
      "polling": true,
      "esModuleInterop": true,
          "pollingInterval": 2500,
```

```
      "followSymlinks": true
    }
  },
  "include": [
    "src/**/*.ts",
    "src/**/*.tsx",
    "@types"
  ],
  "exclude": [
    "node_modules",
    "dist"
  ]
}
```

Advanced Configuration

- **Project Reference**: Useful for code splitting in large projects.
- **Custom Transformers**: Employ custom logic during the compilation process.
- **Programmatic API**: Provides flexibility in managing compiler settings and execution.

11. Explain *classes* in *TypeScript*. How are they different from *ES6 classes*?

While TypeScript and **ES6** classes share many similarities, TypeScript's classes offer additional features and strong typing to make your code more robust.

Key Shared Class Features

- **Inheritance**: Subclasses (children) can extend parent classes, inheriting their methods and properties.
- **Polymorphism**: Derived classes can define methods with the same name as their parent.
- **Encapsulation**: Data hiding is supported through access modifiers, such as `public`, `private`, and `protected`.
- **Constructor**: Instantiation starts with a constructor method, if defined.

Unique TypeScript Class Features

Field Declaration

In TypeScript, you can specify fields directly in the class without initializing them. Automatic initialization to `undefined` occurs during object creation. ES6 requires initializing fields in the constructor or within their declaration.

```
class Example {
    // Field is automatically initialized to undefined upon object creation
    someField: string;
}
```

Abstract Classes

TypeScript supports abstract classes to serve as a blueprint for other classes. They cannot be instantiated on their own but can provide some implementation that derived classes can override.

```
abstract class AbstractExample {
    abstract someMethod(): void;  // Method has no implementation (abstract)
}
```

Readonly Properties

You can mark class properties as `readonly`, ensuring they are only set upon declaration or within the class constructor.

```
class Example {
    readonly id: number;
```

```
    constructor(id: number) {
        this.id = id;  // Readonly can only be assigned in the constructor or declaration
    }
}
```

Static Members

Classes in TypeScript support static members, such as properties and methods that belong to the class itself, rather than to instances of the class.

```
class Example {
    static count = 0;  // Static property
    static incrementCount() {
        Example.count++;
    }
}
```

Accessor Functions

You can define `get` and `set` functions in TypeScript, known as accessor functions, to control how class properties are accessed and modified.

```
class Example {
    private _name: string;

    get name(): string {
        return this._name;
    }

    set name(newName: string) {
        this._name = newName.trim();
    }
}
```

Parameter Properties

TypeScript provides a shortcut to declare a property and initialize it from the constructor parameter. This method can make code more concise, especially when a constructor parameter corresponds directly to a class property.

```
class Example {
    constructor(private _name: string, public age: number) {
        // Private _name property created and initialized from constructor parameter
        // Public age property created and initialized from constructor parameter
    }
}
```

Intersection Types for Classes

In TypeScript, when you define a base class and then later extend it, you are creating an intersection type. This means the child class will inherit all the properties and methods from both its parent(s) and itself.

ES6 Additional Features not Present in TypeScript

Class Expressions

Both ES6 and TypeScript support class expressions, which allows you to define a class without a class name.

In ES6:

```
const Animal = class {
    // Class methods and properties defined here
};
```

Iterator Protocol

ES6 classes support the Iterator protocol, making it easier to iterate over objects.

```
class IterableExample implements Iterable<string> {
    // Implement iterator function for strings
    [Symbol.iterator]() {
        let index = 0;
        const data = ['one', 'two', 'three'];
        return {
            next: () => {
                if (index < data.length) {
                    return { value: data[index++], done: false };
                }
                return { value: undefined, done: true };
            }
        };
    }
}
```

12. How do you implement *Inheritance* in *TypeScript*?

Let's look at how you can use **inheritance** in TypeScript using both **ES6 classes** and **prototypal inheritance**.

1. Inheritance with ES6 Class Syntax

With the advent of ES6, a more familiar class-based inheritance method was introduced. This method is usually easier to read and understand.

Code Example: Inheritance using ES6 Classes

Here is the TypeScript code:

```
class Animal {
  private name: string;

  constructor(theName: string) {
    this.name = theName;
  }

  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Snake extends Animal {
  constructor(name: string) {
    super(name);
  }

  move(distanceInMeters = 5) {
    console.log("Slithering...");
    super.move(distanceInMeters);
  }
}

const mySnake = new Snake("Cobra");
mySnake.move();  // Output: Slithering... Cobra moved 5m.
```

2. Inheritance with Prototypal Methodology

Prior to ES6, TypeScript, like JavaScript, used a **prototypal** inheritance approach.

The prototypal mechanism can be useful when developing complex object structures, but it is important to be aware of the nuances in order to avoid unexpected behavior.

Code Example: Prototypal Inheritance in TypeScript

Here is the TypeScript code:

```
// Define the Parent Class
function Animal(this: Animal, name: string) {
  this.name = name;
}

Animal.prototype.move = function(distanceInMeters: number = 0) {
  console.log(`${this.name} moved ${distanceInMeters}m.`);
};

// Define the Child Class
function Snake(name: string) {
  Animal.call(this, name);
}

// Set up the Inheritance
Snake.prototype = Object.create(Animal.prototype);
Snake.prototype.constructor = Snake;

// Override the Base Type's Method
Snake.prototype.move = function(distanceInMeters = 5) {
  console.log("Slithering...");
  Animal.prototype.move.call(this, distanceInMeters);
};

const mySnake = new Snake("Cobra");
mySnake.move();  // Output: Slithering... Cobra moved 5m.
```

Simplified Methods

Using the prototypal construct can be perplexing at first, but its strength lies in its flexibility.

You can avoid the complexities of direct prototype assignments using ES5 derived constructions, as seen next:

```
function Animal(name: string) {
  this.name = name;
}

Animal.prototype.move = function(distanceInMeters: number = 0) {
  console.log(`${this.name} moved ${distanceInMeters}m.`);
};

function Snake(name: string) {
  Animal.call(this, name);
}

// Utilize `Object.create` for simplified prototype delegation
Snake.prototype = Object.create(Animal.prototype);
Snake.prototype.constructor = Snake;

Snake.prototype.move = function(distanceInMeters = 5) {
  console.log("Slithering...");
  Animal.prototype.move.call(this, distanceInMeters);
};
```

13. What are *access modifiers* and how do they work in *TypeScript*?

Access modifiers are TypeScript's way of controlling class member visibility and mutability. They enforce encapsulation and are especially useful for object-oriented design.

Key Modifiers

- **Public**: Default for class members. They are accessible from both inside and outside the class.
- **Protected**: Members can be accessed within the class and its subclasses. They help establish the "is-a" relationship.

- **Private**: Marks members as accessible only within the declaring class. This ensures they're not modified or accessed externally.

Code Example: Access Modifiers in Action

Here is the TypeScript code:

```
class Person {
    public name: string;
    private age: number;
    protected contact: string;

    constructor(name: string, age: number, contact: string) {
        this.name = name;
        this.age = age;
        this.contact = contact;
    }
}

class Employee extends Person {
    private employeeId: string;

    constructor(name: string, age: number, contact: string, employeeId: string) {
        super(name, age, contact);
        this.employeeId = employeeId;
    }

    public displayDetails(): void {
        console.log(`${this.name} - ${this.age} - ${this.contact} - ${this.employeeId}`);
    }
}

// Somewhere in your code
const person = new Person("John Doe", 30, "1234567");
console.log(person.name);  // Accessible
console.log(person.age);   // ERROR: 'age' is private

const employee = new Employee("Jane Doe", 25, "2345678", "E123");
console.log(employee.contact);  // ERROR: 'contact' is protected
employee.displayDetails();     // Correctly displays details

employee.age = 35;    // ERROR: 'age' is private
employee.contact = "3456789";  // ERROR: 'contact' is protected
```

14. Discuss *Abstract classes* and their purposes in *TypeScript*.

In TypeScript, **abstract classes serve as blueprints** that guide derived classes, essentially laying out the structure without necessarily providing complete implementations of methods.

Core Features of Abstract Classes

Method Signatures

Abstract classes define method signatures without specifying their functionality. This feature provides a comprehensive form for derived classes to work from.

README

In addition to method signatures, abstract classes can contain completely implemented methods. These methods either support the abstract methods or serve as independent functionalities.

Abstract and Non-Abstract Members Separation

Abstract classes clearly demarcate between methods that require implementation by derived classes and those that are either fully implemented or optional.

Common Use-Cases for Abstract Classes

- Facilitate Reusability: Abstract classes help in consolidating common or shared functionalities among several derived classes.
- Contract Enforcement: They ensure that derived classes conform to a shared structure, guaranteeing a defined set of methods that must be implemented.
- Partial Implementations: Abstract classes allow for a mix of fully implemented methods alongside those requiring concrete implementations in derived classes.

TypeScript Utility: Static Properties

Abstract classes in TypeScript can have **static members**, which belong to the class itself and not to any specific instance. This feature provides a convenient way to define properties or methods that are accessible without the need for class instantiation.

Code Example: Abstract Class

Here is the TypeScript code:

```
abstract class Shape {
    abstract getArea(): number;
    abstract getPerimeter(): number;
    color: string;

    constructor(color: string) {
        this.color = color;
    }

    static defaultColor: string = 'red';

    describe() {
        return `This shape is ${this.color}.`;
    }
}

// This will throw an error because the derived class does not provide concrete implementations for abstract methods.
class Circle extends Shape {
    constructor(public radius: number, color: string) {
        super(color);
    }

    // The 'Circle' class inherited the following properties from 'Shape', but neither implements nor specifies them in the der
    getArea(): number {
        return Math.PI * this.radius ** 2;
    }

    getPerimeter(): number {
        return 2 * Math.PI * this.radius;
    }
}

const myCircle = new Circle(5, 'blue');
console.log(myCircle.getArea()); // Outputs: 78.54
console.log(myCircle.describe()); // Outputs: This shape is blue.
console.log(Shape.defaultColor); // Outputs: red
```

15. Can you describe the use of *Constructors* within *TypeScript classes*?

TypeScript provides a convenient way to define **constructors** for classes using the `constructor` keyword. A constructor method allows you to initialize class members and can have access specifiers. They are useful for setting up an object's initial state.

Key Features

- **Automatic Invocation**: The constructor is automatically called when an object of the class is instantiated.

- **Single Unique Constructor**: A class can only have one constructor, providing a centralized place for initialization.
- **Overload Capabilities**: You can overload a constructor to define multiple ways of object initialization.

Example: Constructor in TypeScript

We use the `this` keyword to refer to the current instance, ensuring proper data assignment.

```
class Person {
  // Member variables
  name: string;
  age: number;

  // Constructor
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
}
```

Constructor Access Modifiers

TypeScript supports **access modifiers** on constructor parameters, enabling concise and safe class initialization.

- **Public**: Parameters without a modifier are public by default.
- **Private**: Adding the `private` keyword makes them accessible within the class only.
- **Read-Only**: Combining `readonly` with parameter and the private or public access modifier ensures the parameter is assigned a value just once, in the constructor.

Explore all 100 answers here 👉 **Devinterview.io - Typescript**

Releases

No releases published

Packages

No packages published