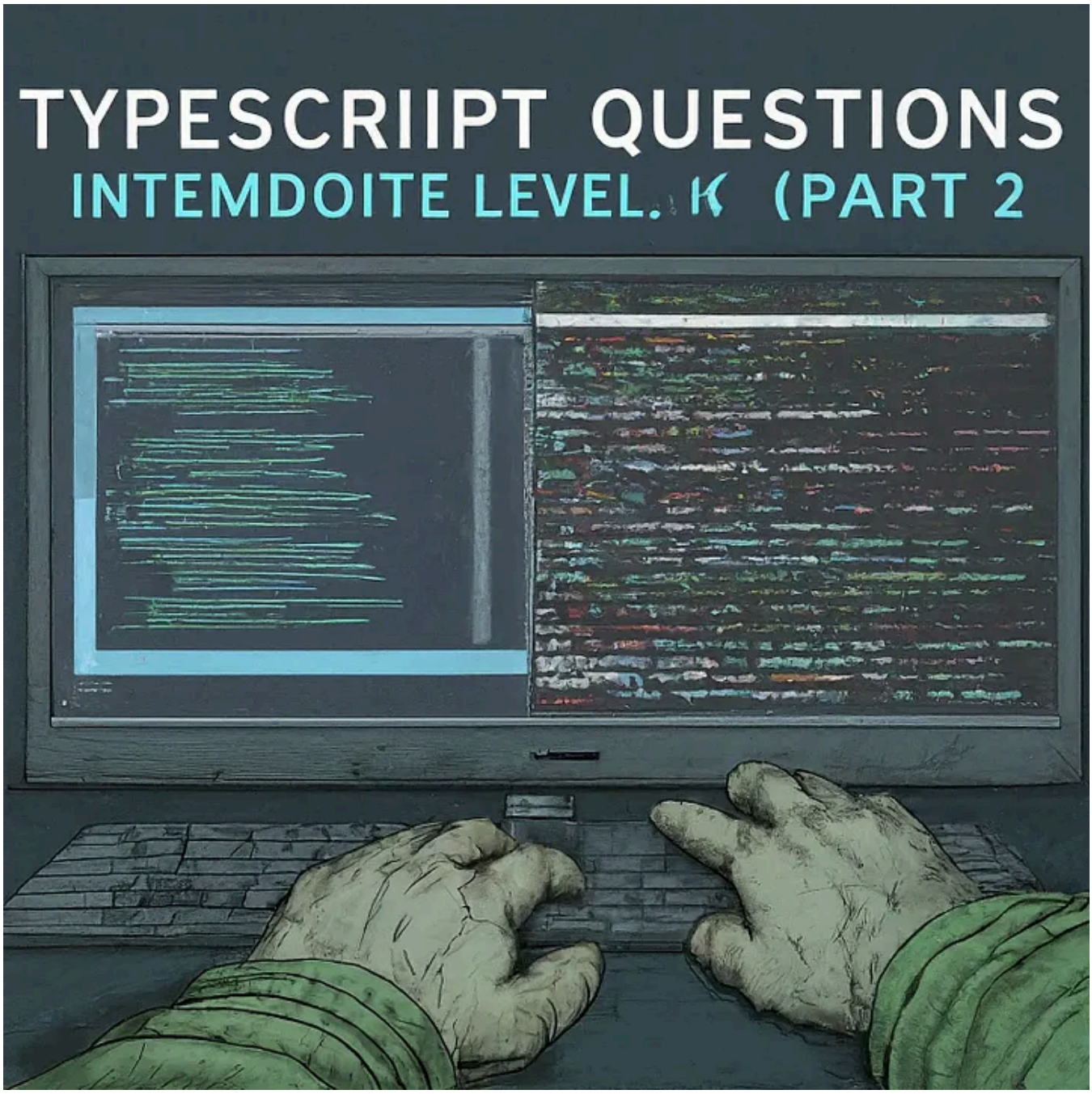
# TypeScript Interview Questions: Intermediate Level Part 2

7 min read · May 19, 2024

Pawan Kumar Follow

Listen Share



As you advance in your understanding of TypeScript, you'll encounter more complex features that can greatly enhance the robustness and maintainability of your code. In this article, we'll delve into intermediate-level TypeScript interview questions, complete with answers and examples, to help you prepare for more advanced discussions and practical applications of the language.

### 1. What are TypeScript decorators and how are they used?

Answer: Decorators are a special kind of declaration that can be attached to a class, method, accessor, property, or parameter. They are used to modify the behavior of the item they are attached to. Decorators are experimental features and require enabling the `experimentalDecorators` option in `tsconfig.json`.

### Example:

```
function log(target: any, key: string) {
  let value = target[key];

  const getter = () => {
    console.log(`Getting value: ${value}`);
    return value;
  };

  const setter = (newValue: any) => {
    console.log(`Setting value: ${newValue}`);
    value = newValue;
  };

  Object.defineProperty(target, key, {
    get: getter,
    set: setter,
    enumerable: true,
    configurable: true
  });
}

class Person {
  @log
  public name: string;

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

let person = new Person('John Doe');
```

```
person.name = 'Jane Doe'; // Setting value: Jane Doe
console.log(person.name); // Getting value: Jane Doe
```

### 2. How does TypeScript handle null and undefined?

Answer: TypeScript has `null` and `undefined` types. By default, `null` and `undefined` are subtypes of all other types, meaning you can assign `null` or `undefined` to any type. However, with the `strictNullChecks` flag enabled in `tsconfig.json`, `null` and `undefined` are only assignable to `any`, `unknown`, `void`, or their respective types.

**Example:**

```
let name: string | null = 'Alice';
name = null; // Allowed when strictNullChecks is false

function greet(name: string | null) {
  if (name === null) {
    console.log('Hello, stranger!');
  } else {
    console.log(`Hello, ${name}!`);
  }
}

greet(null); // Hello, stranger!
greet('Bob'); // Hello, Bob!
```

### 3. What is the difference between `interface` and `type` in TypeScript?

Answer: Both `interface` and `type` can be used to define the shape of an object. However, there are some differences:

- `interface` can be merged (declaration merging), but `type` cannot.
- `type` can define a union or intersection type, but `interface` cannot.
- `interface` is generally preferred for defining object shapes, while `type` is used for more complex type definitions.

**Example:**

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

// Type alias
type PersonType = {
  name: string;
  age: number;
};

// Using type for union
type StringOrNumber = string | number;

// Using type for intersection
type Name = { name: string };
type Age = { age: number };
type PersonIntersection = Name & Age;
```

### 4. How do you use generics in TypeScript?

Answer: Generics allow you to create reusable components that can work with a variety of types. You can define generics with a type parameter that can be used within the function, class, or interface.

### Example:

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

let output1 = identity<string>('Hello');
let output2 = identity<number>(123);

console.log(output1); // Hello
console.log(output2); // 123

class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
```

```
myGenericNumber.zeroValue = 0;
myGenericNumber.add = (x, y) => x + y;
```

### 5. What are mapped types in TypeScript?

Answer: Mapped types allow you to create new types by transforming properties of an existing type. They are often used with the `keyof` operator to iterate over the keys of a type.

### Example:

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

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

type ReadonlyPerson = Readonly<Person>;

let person: ReadonlyPerson = {
  name: 'Alice',
  age: 30
};

// person.name = 'Bob'; // Error: Cannot assign to 'name' because it is a read-only
```

### 6. How do you create and use namespaces in TypeScript?

Answer: Namespaces are a way to organize code into logical groups and prevent name collisions. They are defined using the `namespace` keyword and can be nested.

### Example:

```
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }

  export class LettersOnlyValidator implements StringValidator {
```

```
    isAcceptable(s: string): boolean {
      return /^[A-Za-z]+$/.test(s);
    }
  }

  export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string): boolean {
      return /^\d{5}$/.test(s);
    }
  }
}

let validators: { [s: string]: Validation.StringValidator } = {};
validators['Letters only'] = new Validation.LettersOnlyValidator();
validators['ZIP code'] = new Validation.ZipCodeValidator();

for (let s in validators) {
  console.log(`"${s}" - ${validators[s].isAcceptable('12345')}`);
}
```

## 7. What are modules in TypeScript and how do they work?

Answer: Modules in TypeScript are files that can export and import other modules using `export` and `import` keywords. Modules help in organizing code and managing dependencies.

## Example:

```
// mathUtils.ts
export function add(a: number, b: number): number {
  return a + b;
}

export function subtract(a: number, b: number): number {
  return a - b;
}

// main.ts
import { add, subtract } from './mathUtils';

console.log(add(5, 3)); // 8
console.log(subtract(5, 3)); // 2
```

## 8. How does TypeScript’s type inference work?

Answer: TypeScript uses type inference to automatically determine the type of a variable when there is no explicit type annotation. This can occur during variable initialization, function return types, and more.

### Example:

```
let x = 3; // TypeScript infers the type as number
x = 'hello'; // Error: Type 'string' is not assignable to type 'number'

function multiply(a: number, b: number) {
  return a * b; // TypeScript infers the return type as number
}
```

## 9. What is a type alias in TypeScript?

Answer: A type alias is a name for any type. It can be used to simplify complex type definitions and make code more readable.

### Example:

```
type StringOrNumber = string | number;

let value: StringOrNumber;
value = 'Hello';
value = 123;

type Point = {
  x: number;
  y: number;
};

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

## 10. How do you define and use union and intersection types in TypeScript?

Answer:

- Union types allow a variable to hold one of several types.

- Intersection types combine multiple types into one.

### Example:

```
// Union Type
type StringOrNumber = string | number;

let value: StringOrNumber;
value = 'Hello';
value = 123;

// Intersection Type
type Person = { name: string };
type Employee = { id: number };
type EmployeeDetails = Person & Employee;

let employee: EmployeeDetails = { name: 'Alice', id: 1 };
```

## 11. What are the advantages of using `readonly` in TypeScript?

Answer: The `readonly` modifier makes properties immutable, preventing them from being reassigned after initialization. This helps maintain the integrity of data and reduces bugs caused by unintended mutations.

### Example:

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

let point: Point = { x: 10, y: 20 };
// point.x = 5; // Error: Cannot assign to 'x' because it is a read-only property.
```

## 12. How do you work with advanced types like `Partial`, `Pick`, and `Omit` in TypeScript?

Answer:

- `Partial<T>` makes all properties in `T` optional.

- `Pick<T, K>` creates a new type by picking a set of properties `K` from `T`.
- `Omit<T, K>` creates a new type by omitting a set of properties `K` from `T`.

**Example:**

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

// Partial
type PartialPerson = Partial<Person>;

// Pick
type NameAndAge = Pick<Person, 'name' | 'age'>;

// Omit
type AddressOnly = Omit<Person, 'name' | 'age'>;

let partialPerson: PartialPerson = { name: 'Alice' };
let nameAndAge: NameAndAge = { name: 'Bob', age: 30 };
let addressOnly: AddressOnly = { address: '123 Main St' };
```

### 13. What are conditional types in TypeScript?

Answer: Conditional types enable type relationships to be expressed using conditional logic. The type `T extends U ? X : Y` means if `T` is assignable to `U`, then the type is `X`, otherwise it's `Y`.

**Example:**

```
type IsString<T> = T extends string ? 'Yes' : 'No';

type A = IsString<string>; // 'Yes'
type B = IsString<number>; // 'No'
```

### 14. How do you handle asynchronous operations in TypeScript?

Answer: TypeScript handles asynchronous operations using promises and the `async` / `await` syntax, similar to JavaScript. TypeScript provides type definitions for

promises and async functions.

### Example:

```
function fetchData(): Promise<string> {
  return new Promise((resolve) => {
    setTimeout(() => resolve('Data received'), 1000);
  });
}

async function getData() {
  const data = await fetchData();
  console.log(data);
}

getData(); // Data received
```

## 15. How do you define and use utility types in TypeScript?

Answer: TypeScript provides several utility types to facilitate common type transformations:

- `Partial<T>`
- `Required<T>`
- `Readonly<T>`
- `Record<K, T>`
- `Pick<T, K>`
- `Omit<T, K>`

### Example:

```
interface Todo {
  title: string;
  description: string;
}

// Partial
type PartialTodo = Partial<Todo>;
```

```
// Required
type RequiredTodo = Required<Todo>;

// Readonly
type ReadonlyTodo = Readonly<Todo>;

// Record
type StringRecord = Record<string, number>;

let partialTodo: PartialTodo = { title: 'Learn TypeScript' };
let requiredTodo: RequiredTodo = { title: 'Learn TypeScript', description: 'Underst
let readonlyTodo: ReadonlyTodo = { title: 'Learn TypeScript', description: 'Underst
// readonlyTodo.title = 'New Title'; // Error: Cannot assign to 'title' because it
let stringRecord: StringRecord = { key1: 1, key2: 2 };
```

## Conclusion

By mastering these intermediate-level concepts in TypeScript, you can write more efficient, type-safe, and maintainable code. These skills are crucial for building robust applications and will be valuable in your career as a developer. Stay tuned for the next part of this series where we will delve into advanced TypeScript interview questions.



For further reading, don't miss the other parts of this series:

- **TypeScript Interview Questions: From Beginners to Advanced Part 1**
- **TypeScript Interview Questions: Advanced Level Part 3**