TypeScript Interview Questions: Advanced Level Part 3

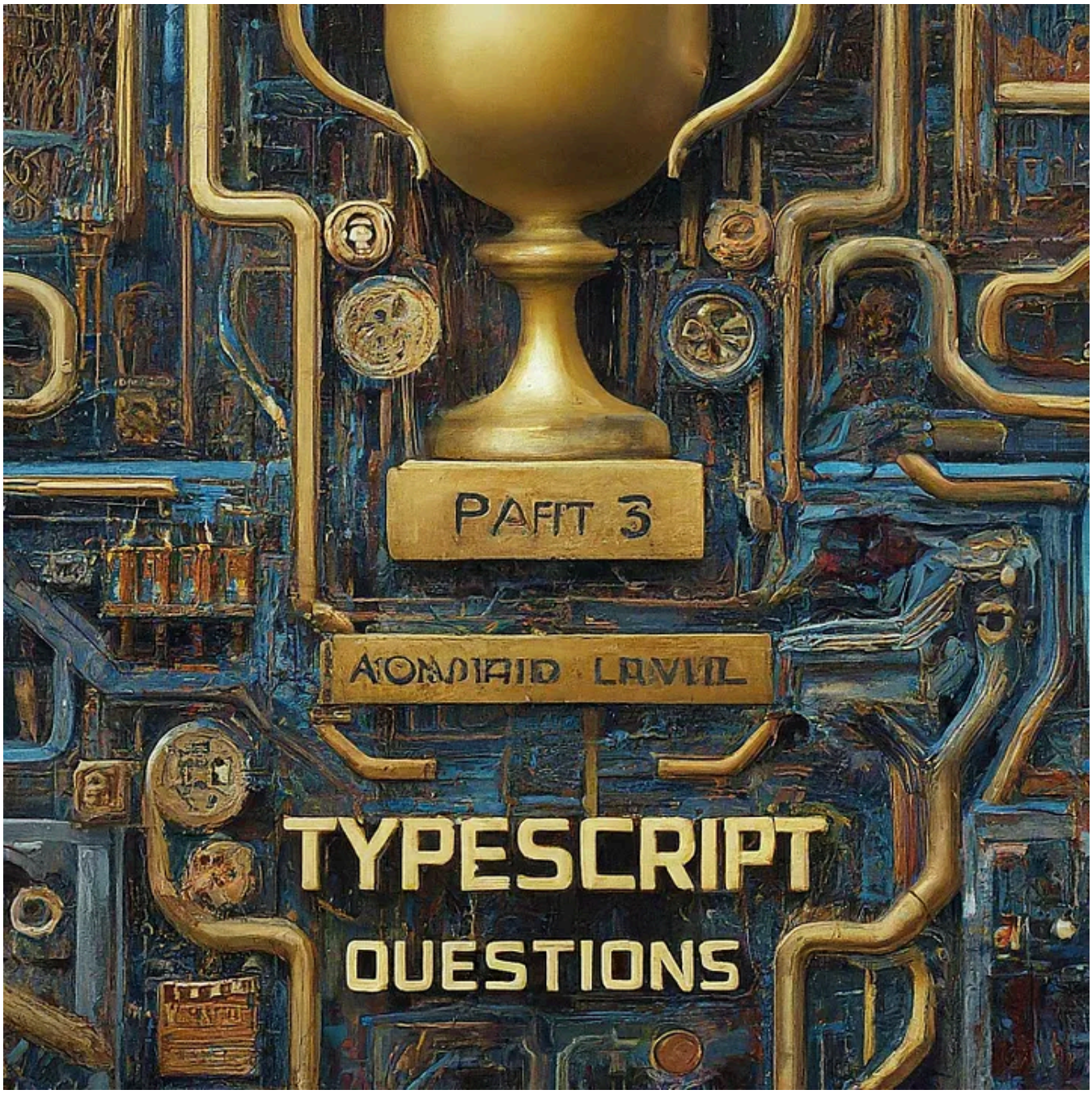
6 min read · May 19, 2024

Pawan Kumar

Follow

Listen

Share



As you dive deeper into TypeScript, you'll encounter more complex and nuanced features that can significantly enhance your coding skills and understanding of the language. This article covers advanced TypeScript interview questions, complete with answers and examples, to help you master these sophisticated concepts.

1. What is type narrowing in TypeScript and how does it work?

Answer: Type narrowing is the process of refining the type of a variable within a certain scope, typically through conditional checks. TypeScript uses control flow analysis to determine the type of a variable after these checks.

Example:

```
function printId(id: number | string) {
  if (typeof id === 'string') {
    console.log(`ID as string: ${id.toUpperCase()}`);
  } else {
    console.log(`ID as number: ${id}`);
  }
}

printId(101); // ID as number: 101
printId('202'); // ID as string: 202
```

2. How does TypeScript's type system improve code safety and readability?

Answer: TypeScript's type system catches errors at compile time, ensuring that variables and functions are used correctly. It improves code readability by providing explicit type annotations and interfaces, making it clear what types are expected and used.

Example:

```
interface User {
  id: number;
  name: string;
}

function getUser(user: User): string {
  return `User: ${user.name}, ID: ${user.id}`;
}
```

```
const newUser: User = { id: 1, name: 'Alice' };
console.log(getUser(newUser)); // User: Alice, ID: 1
```

3. How do you implement custom type guards in TypeScript?

Answer: Custom type guards are functions that determine whether a variable is of a specific type. They return a type predicate, which is a return type of `arg is Type`.

Example:

```
interface Cat {
  meow: () => void;
}

interface Dog {
  bark: () => void;
}

function isCat(pet: Cat | Dog): pet is Cat {
  return (pet as Cat).meow !== undefined;
}

function makeSound(pet: Cat | Dog) {
  if (isCat(pet)) {
    pet.meow();
  } else {
    pet.bark();
  }
}

const cat: Cat = { meow: () => console.log('Meow') };
const dog: Dog = { bark: () => console.log('Bark') };

makeSound(cat); // Meow
makeSound(dog); // Bark
```

4. What are discriminated unions in TypeScript?

Answer: Discriminated unions (or tagged unions) are a type-safe way to handle different types that share a common field. This common field (the discriminant) helps TypeScript determine the type of the union.

Example:

```
interface Circle {
  kind: 'circle';
  radius: number;
}

interface Square {
  kind: 'square';
  sideLength: number;
}

type Shape = Circle | Square;

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case 'circle':
      return Math.PI * shape.radius ** 2;
    case 'square':
      return shape.sideLength ** 2;
  }
}

const myCircle: Circle = { kind: 'circle', radius: 5 };
const mySquare: Square = { kind: 'square', sideLength: 4 };

console.log(getArea(myCircle)); // 78.53981633974483
console.log(getArea(mySquare)); // 16
```

5. How do you use the `keyof` and `typeof` operators in TypeScript?

Answer:

- `keyof` is used to obtain the keys of a type as a union of string literal types.
- `typeof` is used to obtain the type of a variable or expression.

Example:

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

type PersonKeys = keyof Person; // "name" | "age"

const person = {
  name: 'Alice',
```

```
  age: 30
};

type PersonType = typeof person; // { name: string; age: number }
```

6. What are index types and how are they used in TypeScript?

Answer: Index types allow you to get the type of an object's properties using the `keyof` and `[]` operators. They are useful for dynamic key access and type safety.

Example:

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

type PersonKeys = keyof Person; // "name" | "age"

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

const person: Person = { name: 'Alice', age: 30 };
const name = getProperty(person, 'name'); // string
const age = getProperty(person, 'age'); // number
```

7. How do you handle complex type relationships and constraints with generics in TypeScript?

Answer: Generics allow you to create flexible and reusable components. You can define constraints using the `extends` keyword to ensure that a type parameter meets specific criteria.

Example:

```
interface Lengthwise {
  length: number;
}

function logLength<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
```

```
  return arg;
}

logLength({ length: 10, value: 'Hello' }); // 10
// logLength(10); // Error: Argument of type 'number' is not assignable to paramete
```

8. What are the differences between `abstract` classes and interfaces in TypeScript?

Answer:

- `abstract` classes can provide implementation details and define methods that must be implemented by subclasses. They cannot be instantiated directly.
- Interfaces define a contract that a class must follow without providing implementation. They are purely structural.

Example:

```
abstract class Animal {
  abstract makeSound(): void;

  move(): void {
    console.log('Moving...');
  }
}

class Dog extends Animal {
  makeSound(): void {
    console.log('Bark');
  }
}

interface Shape {
  area(): number;
}

class Circle implements Shape {
  constructor(private radius: number) {}

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

9. How does TypeScript support mixins and how do you use them?

Answer: Mixins are a pattern to create classes that combine functionality from multiple sources. TypeScript supports mixins through class expressions and the `extends` keyword.

Example:

```
type Constructor<T = {}> = new (...args: any[]) => T;

function CanJump<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    jump() {
      console.log('Jumping');
    }
  };
}

function CanRun<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    run() {
      console.log('Running');
    }
  };
}

class Person {
  constructor(public name: string) {}
}

const Athlete = CanJump(CanRun(Person));

const athlete = new Athlete('John');
athlete.jump(); // Jumping
athlete.run(); // Running
```

10. What is the significance of declaration merging in TypeScript?

Answer: Declaration merging allows multiple declarations with the same name to be combined into a single definition. This is commonly used with interfaces and namespaces.

Example:

```
interface Person {
  name: string;
```

```
}

interface Person {
  age: number;
}

const person: Person = { name: 'Alice', age: 30 };

namespace MyNamespace {
  export class MyClass {}
}

namespace MyNamespace {
  export function myFunction() {}
}

MyNamespace.myFunction();
```

11. How do you extend built-in types and interfaces in TypeScript?

Answer: You can extend built-in types and interfaces by creating new interfaces that extend the original ones or by using module augmentation.

Example:

```
interface String {
  repeat(times: number): string;
}

String.prototype.repeat = function(times: number): string {
  return new Array(times + 1).join(this);
};

console.log('Hello'.repeat(3)); // HelloHelloHello
```

12. What are the best practices for structuring large TypeScript projects?

Answer: Best practices include:

- Using a consistent project structure
- Organizing code into modules and namespaces
- Keeping configuration files (`tsconfig.json`) up-to-date

- Using type annotations and interfaces for clarity
- Applying strict compiler options for type safety
- Writing comprehensive tests

Example Project Structure:

```
src/
  models/
    user.ts
  services/
    userService.ts
  controllers/
    userController.ts
  index.ts
tsconfig.json
package.json
```

13. How do you optimize TypeScript code for performance?

Answer: Optimization techniques include:

- Using the latest TypeScript version for improved features and performance
- Enabling `strict` mode for better type safety
- Minimizing use of `any` type
- Utilizing advanced types and type inference
- Compiling to a suitable ECMAScript target

Example:

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true
```

```
    }
}
```

14. What are the key differences between TypeScript 3.x and 4.x?

Answer: Key differences include:

- TypeScript 4.x introduced new features like variadic tuple types, labeled tuple elements, and more strictness in the type system.
- Improvements in performance and tooling
- Better support for editor features and integrations

Example:

```
// Variadic Tuple Types (TypeScript 4.x)
function concat<T extends any[], U extends any[]>(arr1: [...T], arr2: [...U]): [...
  return [...arr1, ...arr2];
}

const result = concat([1, 2], ['a', 'b']); // [1, 2, 'a', 'b']
```

15. How do you ensure compatibility and interoperability between TypeScript and JavaScript?

Answer: Ensure compatibility by:

- Using declaration files (`.d.ts`) to describe the shape of JavaScript code
- Leveraging JSDoc comments for type annotations in JavaScript
- Configuring `tsconfig.json` for mixed codebases
- Avoiding TypeScript-specific features when writing code meant to run in both environments

Example:

```
// JavaScript file (utils.js)
/**
 * @param {number} a
 * @param {number} b
 * @returns {number}
 */
function add(a, b) {
  return a + b;
}

// TypeScript file (index.ts)
/// <reference path="./utils.js" />
import { add } from './utils';

console.log(add(1, 2)); // 3
```

Conclusion

Mastering these advanced TypeScript concepts will significantly enhance your ability to write robust, efficient, and scalable applications. By understanding and applying these features, you'll be well-equipped to handle complex coding challenges and excel in your development career.

For further reading, be sure to check out the other parts of this series:

- **TypeScript Interview Questions: From Beginners to Advanced Part 1**
- **TypeScript Interview Questions: Intermediate Level Part 2**