TypeScript Topics and Questions

TypeScript Basics

Theory

- **What is TypeScript?**
  TypeScript is a superset of JavaScript that adds static typing, enabling type checking at compile time to catch errors early and improve code maintainability.
- **What is the difference between TypeScript and JavaScript?**
  TypeScript extends JavaScript with static types, interfaces, and advanced features like enums and generics, while JavaScript is dynamically typed and lacks these compile-time checks.
- **What is static typing?**
  Static typing means types are checked at compile time, ensuring variables adhere to defined types before execution.
- **What is dynamic typing?**
  Dynamic typing, as in JavaScript, means types are determined at runtime, offering flexibility but increasing the risk of type-related errors.
- **What are the disadvantages of TypeScript?**
  Disadvantages include a steeper learning curve, additional compilation step, and potential over-complexity for small projects.
- **What is the `tsc` compiler?**
  The TypeScript compiler (`tsc`) converts TypeScript code to JavaScript, performing type checking and applying configuration from `tsconfig.json`.
- **What is transpilation?**
  Transpilation is the process of converting TypeScript code to JavaScript, enabling it to run in browsers or Node.js environments.
- **What is `tsconfig.json`?**
  `tsconfig.json` is a configuration file for TypeScript projects, specifying compiler options, file inclusions, and output settings.
- **What is the `--target` compiler flag?**
  The `--target` flag specifies the JavaScript version (e.g., ES5, ES6) to which TypeScript code is compiled.
- **What is the `--noEmitOnError` flag?**
  When set, it prevents TypeScript from generating output JavaScript files if there are compilation errors.
- **What is the `noImplicitAny` flag?**
  When enabled, it disallows implicit `any` types, forcing explicit type annotations for untyped variables.

- **What is the `strict` flag?**

  The `strict` flag enables a set of strict type-checking options, like `strictNullChecks` and `noImplicitAny`, for safer code.
- **What is `strictNullChecks`?**

  When enabled, `null` and `undefined` are not implicitly assignable to other types, reducing null reference errors.
- **What is the `outFile` compiler option?**

  The `outFile` option concatenates all TypeScript files into a single JavaScript output file, useful for specific build scenarios.
- **What is tree shaking in TypeScript?**

  Tree shaking eliminates unused code during bundling, supported in TypeScript with compatible module systems like ES Modules.
- **What is the `--watch` flag?**

  The `--watch` flag enables automatic recompilation of TypeScript files on changes, streamlining development.
- **How does TypeScript handle null and undefined?**

  TypeScript treats `null` and `undefined` as distinct types, with `strictNullChecks` enforcing explicit handling to avoid errors.
- **What is `null` vs `undefined`?**

  `null` represents an intentional absence of value, while `undefined` indicates a variable has not been assigned a value.
- **What is `any` vs `unknown`?**

  `any` disables type checking, allowing any operation, while `unknown` is safer, requiring type checks before operations.
- **What is `never` vs `void`?**

  `void` represents a function that returns no value, while `never` indicates a function that never returns (e.g., throws an error or loops infinitely).
- **What is `never` vs `unknown`?**

  `never` represents values that never occur (e.g., a function that always throws), while `unknown` is a type-safe alternative to `any`.
- **What is the `unknown` type?**

  The `unknown` type is a type-safe alternative to `any`, requiring type narrowing before performing operations.
- **What is TypeScript's static type checker?**

  TypeScript's static type checker verifies types at compile time, catching type-related errors before runtime.
- **What is downleveling?**

  Downleveling is the process of transpiling modern JavaScript (or TypeScript) features to an older version (e.g., ES5) for compatibility.
- **What are TypeScript's best practices?**

  Best practices include using strict mode, avoiding `any`, leveraging interfaces/types, and maintaining consistent `tsconfig.json` settings.
- **What is duck typing/structural typing?**

  TypeScript uses structural typing, where compatibility is based on the structure (properties/methods) of objects, not their explicit type names.
- **What are erased types?**

  TypeScript types are removed during compilation to JavaScript, leaving no type information at runtime.

- **What is contextual typing?**
  Contextual typing infers types based on the context in which a value is used, reducing the need for explicit annotations.
- **What is type inference?**
  Type inference automatically determines variable types based on their initial values or usage, minimizing explicit type annotations.
- **What is module augmentation?**
  Module augmentation extends existing modules by adding new declarations, often used to enhance third-party libraries.
- **What is declaration merging?**
  Declaration merging allows multiple declarations (e.g., interfaces) with the same name to be combined into a single definition.
- **What are ambient declarations?**
  Ambient declarations define types for external JavaScript libraries without implementing their logic, often in `.d.ts` files.

Practical

- Install and configure TypeScript in a project.
- Set up a `tsconfig.json` file.
- Compile TypeScript to JavaScript using `tsc`.
- Debug a TypeScript file.
- Use the `--watch` flag to compile `.ts` files automatically with real-time changes.

Data Types

Theory

- **What are data types in TypeScript?**
  TypeScript includes primitive types (`string`, `number`, `boolean`, `bigint`, `symbol`, `null`, `undefined`, `object`), as well as complex types like arrays, tuples, and custom types.
- **What are arrays and objects in TypeScript?**
  Arrays are typed collections (e.g., `number[]`), and objects are typed with specific properties using interfaces or types.
- **What is a tuple?**
  A tuple is a fixed-length array with specific types for each element (e.g., `[string, number]`).
- **What is a literal type?**
  Literal types restrict a variable to a specific value (e.g., `type Direction = "up" | "down"`).
- **What is a union type?**
  A union type allows a variable to be one of multiple types (e.g., `string | number`).
- **What is an intersection type?**
  An intersection type combines multiple types into one, requiring all properties (e.g., `TypeA & TypeB`).

- **What is a type alias?**

  A type alias defines a reusable type using the `type` keyword (e.g., `type Point = { x: number; y: number }`).
- **What are optional object types?**

  Optional object properties, marked with `?`, allow properties to be undefined (e.g., `{ name?: string }`).
- **What are optional tuple elements?**

  Tuple elements can be marked optional with `?`, allowing fewer elements than the tuple's length.
- **What is a `ReadonlyArray`?**

  A `ReadonlyArray<T>` is an immutable array type that prevents modifications after creation.
- **What is an immutable array?**

  An immutable array cannot be modified after creation, often achieved with `ReadonlyArray` or `readonly` modifier.

Practical

- Define a variable with a union type.
- Create a tuple.
- Work with array details (e.g., typed arrays, `ReadonlyArray`).
- Create a function that accepts a user-defined type using an interface or type.

Interfaces and Types

Theory

- **What is an interface?**

  An interface defines the structure of an object, specifying properties and methods, and is extendable.
- **What is the difference between `type` and `interface`?**

  `type` can represent any type (unions, intersections, primitives), while `interface` is limited to object shapes but supports declaration merging.
- **Why use an interface instead of an abstract class?**

  Interfaces are lightweight, support multiple inheritance, and are purely for type checking, while abstract classes can include implementation logic.
- **What is extending an interface?**

  Extending an interface allows a new interface to inherit properties from another using `extends` (e.g., `interface B extends A`).
- **What is the `implements` clause?**

  The `implements` clause ensures a class adheres to an interface's structure (e.g., `class MyClass implements MyInterface`).
- **What is the difference between `extends` and `implements`?**

  `extends` is used for inheritance (classes or interfaces), while `implements` enforces a class to follow an interface's structure.

- **What are index signatures?**

  Index signatures allow objects to have dynamic keys with a specific type (e.g., `[key: string]: number`).
- **What is a class from an interface?**

  A class can implement an interface to ensure it has the required properties and methods.

Practical

- Create an interface and use it in a class.
- Extend an interface.
- Implement abstraction using an interface.
- Convert JavaScript code to TypeScript with interfaces.
- Use index signatures in an object type.
- Create a class from an interface.

Classes and OOP

Theory

- **What is a class in TypeScript?**

  A class is a blueprint for creating objects with properties and methods, enhanced with TypeScript's type system and access modifiers.
- **What is an abstract class?**

  An abstract class cannot be instantiated and may contain abstract methods that must be implemented by subclasses.
- **What is the difference between an abstract class and an interface?**

  Abstract classes can include implementation details and state, while interfaces are purely structural and cannot contain logic.
- **What is a constructor?**

  A constructor is a special method (`constructor`) used to initialize a class instance.
- **What is the `super` keyword?**

  The `super` keyword calls the parent class's constructor or methods from a child class.
- **What is a static keyword?**

  The `static` keyword defines properties or methods that belong to the class itself, not its instances.
- **What are access modifiers (`public`, `private`, `protected`)?**

  `public` allows access everywhere, `private` restricts to the class, and `protected` allows access in the class and subclasses.
- **What is the difference between `private` and `protected`?**

  `private` restricts access to the class only, while `protected` allows access in subclasses.
- **What is encapsulation?**

  Encapsulation hides internal details of a class, exposing only necessary parts via public methods or properties.
- **What is polymorphism?**

  Polymorphism allows objects of different classes to be treated as instances of a common type, often through inheritance or interfaces.

- **What are the types of polymorphism?**
  Types include compile-time (method overloading) and runtime (method overriding via inheritance or interfaces).
- **What is method overloading?**
  Method overloading allows multiple method signatures with different parameters but the same name in a class.
- **What is method overriding?**
  Method overriding allows a subclass to provide a specific implementation of a method defined in its parent class.
- **What is a singleton class?**
  A singleton class ensures only one instance exists, often implemented with a static method to access the instance.
- **What are getters and setters?**
  Getters and setters are methods to access (`get`) and modify (`set`) private properties in a controlled way.
- **What is constructor chaining?**
  Constructor chaining involves calling one constructor from another within the same class or parent class using `super`.
- **What is a static class?**
  TypeScript doesn't have true static classes, but `static` members can simulate class-level functionality.
- **What are mixins?**
  Mixins are a pattern to share functionality between classes using composition, often implemented via functions or interfaces.
- **What is dependency injection?**
  Dependency injection passes dependencies (e.g., services) to a class, promoting loose coupling and testability.
- **What are SOLID principles?**
  SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) guide object-oriented design for maintainable code.

Practical

- Create a class with a constructor.
- Implement abstraction using an abstract class.
- Implement abstraction using an interface.
- Practice class and inheritance problems.
- Implement multiple inheritance using interfaces.
- Create a singleton class.
- Call a parent class constructor from a child class.
- Call a parent class method from a child class.
- Create a mixin.
- Implement method overriding.
- Implement method overloading (e.g., find the area of a square and rectangle).
- Practice encapsulation.
- Practice polymorphism with interfaces or inheritance.
- Include dependency injection between classes.
- Create proper request and response models.
- Convert the array `[{a:3},{a:3},{a:3},{a:3},{a:3}]` to TypeScript and calculate the sum of `a` values.
- Implement a function that accepts either a Square or Rectangle and returns its area (abstraction).

Functions

Theory

- **What are functions in TypeScript?**

  Functions in TypeScript are like JavaScript functions but with type annotations for parameters and return types.
- **What is a generic function?**

  A generic function uses type parameters to work with multiple types (e.g., `function identity<T>(arg: T): T`).
- **What is a lambda function (anonymous function)?**

  A lambda function is an anonymous function, often used as an expression (e.g., `x => x * 2`).
- **What is a function overload?**

  Function overloads define multiple signatures for a function to handle different parameter types or counts.
- **What is a rest parameter?**

  A rest parameter (`...args: T[]`) collects multiple arguments into an array.
- **What is parameter destructuring?**

  Parameter destructuring extracts properties from objects or arrays passed as arguments (e.g., `({x, y}) => x + y`).
- **What is a variadic function?**

  A variadic function accepts a variable number of arguments, typically using rest parameters.
- **What is a generic identity function?**

  A generic identity function returns its argument unchanged, using a generic type (e.g., `function identity<T>(arg: T): T`).
- **What is a type argument inference?**

  TypeScript infers generic types from arguments if not explicitly provided (e.g., `identity(42)` infers `T` as `number`).
- **What is a generic constraint?**

  A generic constraint restricts a generic type to specific types using `extends` (e.g., `T extends Lengthwise`).
- **What is a function type?**

  A function type defines the shape of a function, including parameter and return types (e.g., `(x: number) => string`).
- **What is a call signature?**

  A call signature defines how a function or object with callable properties can be invoked.
- **What is declaring `this` in a function?**

  Declaring `this` in a function's signature specifies its type in methods (e.g., `function(this: Type)`).
- **What is a Promise type?**

  The `Promise<T>` type represents an asynchronous operation that resolves to a value of type `T`.

Practical

- Create a generic function to reverse an array.
- Create a generic function that takes two different data types as arguments and returns both in an array.

- Create a variadic function to filter string arguments and return them.
- Write a function that returns a Promise resolving with a string or rejecting with a boolean, with proper annotations.
- Create a function to find the sum of two numbers using generics.
- Implement parameter destructuring in a function.
- Use rest parameters in a function.
- Create a generic identity function.

Asynchronous Programming

Theory

- **What is asynchronous programming in TypeScript?**
  Asynchronous programming uses `Promise` or `async/await` to handle operations like API calls, with TypeScript adding type safety.
- **What is `async/await`?**
  `async` marks a function as asynchronous, returning a `Promise`, while `await` pauses execution until the `Promise` resolves.
- **What is a Promise in TypeScript?**
  A `Promise<T>` represents a value that may be available later, with `T` as the resolved value's type.

Practical

- Write an async function to fetch data from an API.
- Implement asynchronous operations using Promises or `async/await` with proper error handling.
- Create a Promise that resolves to a string.
- Create an async function with proper return type annotations.

Type Guards and Narrowing

Theory

- **What is a type guard?**
  A type guard is a runtime check that narrows a variable's type (e.g., `typeof`, `instanceof`, or custom predicates).
- **What is type narrowing?**
  Type narrowing refines a variable's type within a conditional block based on checks (e.g., `if (typeof x === "string")`).
- **What is a type predicate?**
  A type predicate is a function returning `arg is Type`, used to narrow types (e.g., `function isString(arg: any): arg is string`).
- **What is truthiness narrowing?**
  Truthiness narrowing uses boolean checks to narrow types (e.g., `if (x)` narrows `x` from `string | null` to

`string`).

- **What is equality narrowing?**

  Equality narrowing refines types based on equality checks (e.g., `if (x === "value")`).
- **What is `in` operator narrowing?**

  The `in` operator checks for properties to narrow types (e.g., `if ("prop" in obj)`).
- **What is `instanceof` narrowing?**

  `instanceof` narrows types based on class instances (e.g., `if (x instanceof Date)`).
- **What is a discriminated union?**

  A discriminated union uses a common property (discriminator) to distinguish between union types (e.g., `{ kind: "circle" } | { kind: "square" }`).
- **What is exhaustive checking?**

  Exhaustive checking ensures all cases in a discriminated union are handled, often using `never` for unhandled cases.

Practical

- Create a custom type guard.
- Implement a discriminated union.
- Use type narrowing with `typeof`, `instanceof`, or `in` operator.
- Create a function with a type predicate.

Utility Types

Theory

- **What are utility types?**

  Utility types are built-in TypeScript types that transform or manipulate other types (e.g., `Partial`, `Omit`, `Pick`).
- **What is the `Partial` type?**

  `Partial<T>` makes all properties of `T` optional.
- **What is the `Omit` type?**

  `Omit<T, K>` excludes specified keys `K` from type `T`.
- **What is the `Pick` type?**

  `Pick<T, K>` includes only specified keys `K` from type `T`.
- **What is the `Exclude<T, U>` type?**

  `Exclude<T, U>` removes types from `T` that are assignable to `U`.
- **What is the `Extract<T, U>` type?**

  `Extract<T, U>` extracts types from `T` that are assignable to `U`.
- **What is the `Record` type?**

  `Record<K, T>` creates an object type with keys `K` and values of type `T`.
- **What is the `Required` type?**

  `Required<T>` makes all optional properties of `T` required.

Practical

- Use the `Omit` utility type.
- Use the `Partial` utility type.
- Use the `Pick` utility type.
- Use the `Exclude` utility type.
- Use the `Record` utility type.
- Use the `Required` utility type.

Decorators

Theory

- **What are decorators in TypeScript?**
  Decorators are functions that modify classes, methods, properties, or parameters at design time, often used for metadata or behavior injection.
- **What are the types of decorators?**
  Types include class, method, accessor, property, and parameter decorators.
- **What is a decorator factory?**
  A decorator factory is a function that returns a decorator, allowing configuration (e.g., `@factory(arg)`).
- **What is decorator composition?**
  Decorator composition applies multiple decorators to a single target, executed in a specific order.

Practical

- Create a custom decorator.
- Create a decorator factory.
- Apply decorators to a class, method, or property.

Generics

Theory

- **What are generics in TypeScript?**
  Generics allow reusable components that work with multiple types, defined with type parameters (e.g., `<T>`).
- **What is a generic constraint?**
  A generic constraint restricts the types a generic can accept using `extends` (e.g., `T extends { length: number }`).
- **What are generic parameter defaults?**
  Generic parameter defaults provide a fallback type if none is specified (e.g., `<T = string>`).
- **What are the applications of generics?**
  Generics are used for type-safe collections, reusable functions, and classes that work with multiple types.

- **What are guidelines for writing good generic functions?**

  Guidelines include using meaningful type names, minimizing type constraints, and ensuring type safety.

Practical

- Implement a generic function (e.g., reverse an array).
- Create a generic function that takes two different data types and returns them in an array.
- Use a generic constraint (e.g., restrict to types with a `length` property).
- Create a generic identity function.
- Find the area of a square and rectangle using function overloading with generics.

Modules

Theory

- **What are modules in TypeScript?**

  Modules organize code into reusable units, using `import` and `export` for dependency management.
- **What is a namespace?**

  Namespaces group related code under a single name, an older alternative to modules (e.g., `namespace MyNamespace`).
- **What is the difference between modules and namespaces?**

  Modules use file-based `import/export`, while namespaces are internal to a file and less commonly used.
- **What are module types?**

  Module types (e.g., `commonjs`, `esnext`) define how TypeScript handles module resolution and output.

Practical

- Create a custom module with `export` and `import`.
- Use a namespace in a TypeScript file.

Enums

Theory

- **What are enums in TypeScript?**

  Enums define a set of named constants, either numeric or string-based (e.g., `enum Direction { Up, Down }`).
- **What are the advantages of enums?**

  Enums improve code readability, type safety, and maintainability by grouping related constants.

Practical

- Create an enum and use it in a function.

Type Assertion and Casting

Theory

- **What is type assertion?**

  Type assertion tells the compiler to treat a value as a specific type using `as` or `<Type>` syntax (e.g., `x as string`).
- **What is type casting vs type assertion?**

  Type casting is a broader term; in TypeScript, type assertion is used since no runtime conversion occurs.
- **What is force casting?**

  Force casting uses type assertion to override TypeScript's type checking, potentially unsafe (e.g., `x as any`).
- **What is the non-null assertion operator?**

  The non-null assertion operator (`!`) asserts that a value is not `null` or `undefined` (e.g., `x!.property`).

Practical

- Use type assertion with `as` or `<Type>`.
- Create an example for type assertion.
- Use the non-null assertion operator.

Utility Types and Advanced Types

Theory

- **What are mapped types?**

  Mapped types create new types by transforming properties of an existing type (e.g., `{ [P in keyof T]: Type }`).
- **What is the `keyof` type operator?**

  The `keyof` operator returns a union of an object's property names (e.g., `keyof { a: number; b: string }` is `"a" | "b"`).
- **What is the `typeof` type operator?**

  The `typeof` operator extracts the type of a value (e.g., `typeof x` for a variable `x`).
- **What is an indexed access type?**

  Indexed access types extract a type from an object's property (e.g., `T["property"]`).
- **What is a conditional type?**

  Conditional types select a type based on a condition (e.g., `T extends U ? X : Y`).
- **What is the `infer` keyword?**

  The `infer` keyword infers a type within a conditional type (e.g., `T extends Array<infer U> ? U : never`).

- **What is a distributive conditional type?**

  Distributive conditional types apply a conditional type to each member of a union type.
- **What is an anonymous type?**

  An anonymous type is an inline type without a name (e.g., `{ x: number; y: number }`).

Practical

- Create a mapped type.
- Use the `keyof` operator.
- Use a conditional type.
- Use the `infer` keyword in a conditional type.
- Create an example with `Exclude<T, U>` and `Extract<T, U>`.
- Work with deeply nested objects.

Optional Chaining

Theory

- **What is optional chaining?**

  Optional chaining (`?.`) accesses properties or methods safely, returning `undefined` if the chain is `null` or `undefined`.

Practical

- Use optional chaining in an object or function call.

Design Patterns

Theory

- **What is dependency injection?**

  Dependency injection passes dependencies to a class, improving modularity and testability.
- **What is the factory pattern?**

  The factory pattern creates objects without specifying the exact class, using a factory function or class.
- **What is the singleton pattern?**

  The singleton pattern ensures a class has only one instance, accessible globally.
- **What are TypeScript design patterns?**

  Common patterns include factory, singleton, decorator, and repository patterns, adapted with TypeScript's type system.

Practical

- Implement dependency injection.
- Implement the factory pattern.
- Implement the singleton pattern.
- Explore the repository pattern in TypeScript (e.g., as described in LogRocket blog (https://blog.logrocket.com/exploring-repository-pattern-typescript-node/)).

API Calls

Practical

- Write an async function to fetch data from an API.
- Handle API calls with proper type annotations for request and response models.

String Interpolation

Practical

- Use string interpolation (template literals) with TypeScript types.

Iterators and Generators

Theory

- **What are iterators and generators in TypeScript?**
  Iterators provide a way to traverse a collection, while generators (`function*`) yield values incrementally, with TypeScript typing their return values.

Practical

- Create a generator function in TypeScript.

Miscellaneous

Theory

- **What is the `readonly` modifier?**
  The `readonly` modifier prevents properties from being modified after initialization.
- **What is the `const` context?**
  The `const` context ensures literal types for object literals, preventing widening (e.g., `const x = { a: 1 } as const`).

- **What is a symbol in TypeScript?**

  A `symbol` is a unique, immutable primitive used as object keys or for unique identifiers.
- **How to optimize TypeScript performance?**

  Optimize by enabling `strict` mode, minimizing `any` usage, using specific types, and leveraging `noEmitOnError`.

Practical

- Use the `readonly` modifier in a class or interface.
- Use `const` assertions for literal types.
- Use the `symbol` type in an object.