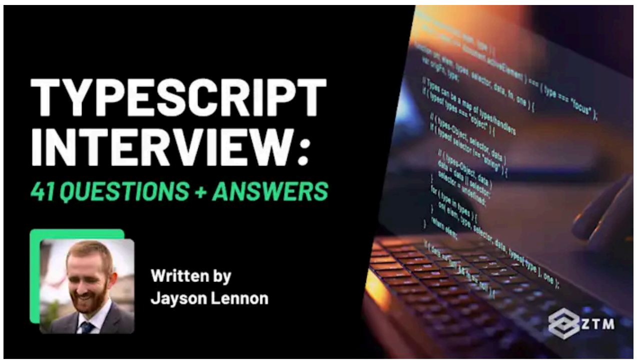
TypeScript Interview Questions + Answers (With Code Examples)

Jayson Lennon

June 21st, 2024 32 min read



In This Guide:

- Beginner Typescript Interview Questions
- Intermediate Typescript Interview Questions
- Advanced Typescript Interview Questions
- How did you do?
- BONUS: More TypeScript tutorials, guides & resources

So you've:

- Learned to code with TypeScript ✅
- Built a project portfolio ✅, and
- Prepped for the tech interview ✅

You should be ready to land that job right... ❓

Maybe!

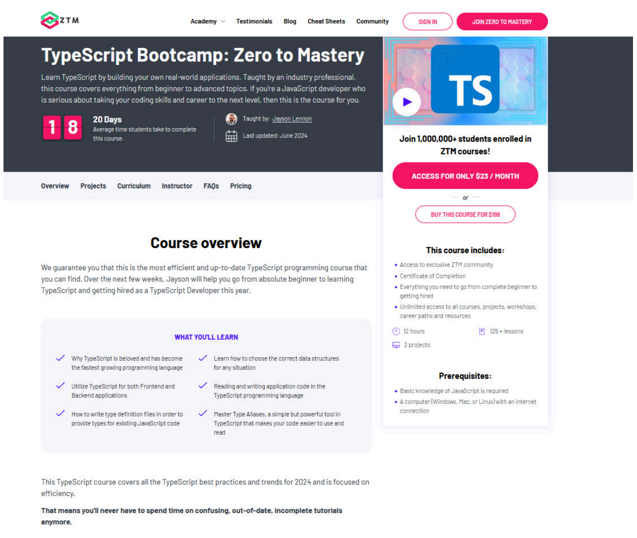
But here's the thing. Some interviewers may ask you some more specific questions about TypeScript before they get into the actual coding and technical problem-solving.

In theory, you should know the answers to all these questions, but it never hurts to be prepared.

This is why in this guide, I'm going to walk you through 42 TypeScript focused questions, along with their answers and code examples.

So grab a coffee, read along, and see how many of these you can answer correctly.

> ***Sidenote:*** *If you find that you're struggling with the questions in this guide, or just want to dive deeper into TypeScript and build some more impressive projects for your portfolio, then come and check out my complete TypeScript Developer course.*



TypeScript Bootcamp: Zero to Mastery

Learn TypeScript by building your own real-world applications. Taught by an industry professional, this course covers everything from beginner to advanced topics. If you're a JavaScript developer who is serious about taking your coding skills and career to the next level, then this is the course for you.

18 20 Days Average time students take to complete this course.

Taught by: Jayson Lennon

Last updated: June 2024

Join 1,000,000+ students enrolled in ZTM courses!

Course overview

We guarantee you that this is the most efficient and up-to-date TypeScript programming course that you can find. Over the next few weeks, Jayson will help you go from absolute beginner to learning TypeScript and getting hired as a TypeScript Developer this year.

WHAT YOU'LL LEARN

- ✓ Why TypeScript is beloved and has become the fastest growing programming language
- ✓ Learn how to choose the correct data structures for any situation
- ✓ Utilize TypeScript for both Frontend and Backend applications
- ✓ Reading and writing application code in the TypeScript programming language
- ✓ How to write type definition files in order to provide types for existing JavaScript code
- ✓ Master Type Aliases, a simple but powerful tool in TypeScript that makes your code easier to use and read

This TypeScript course covers all the TypeScript best practices and trends for 2024 and is focused on efficiency.

That means you'll never have to spend time on confusing, out-of-date, incomplete tutorials anymore.

This course includes:

- Access to exclusive ZTM community
- Certificate of Completion
- Everything you need to go from complete beginner to getting hired
- Unlimited access to all courses, projects, workshops, career paths and resources

12 hours
125 + lessons
2 projects

Prerequisites:

- Basic knowledge of JavaScript is required
- A computer (Windows, Mac, or Linux) with an internet connection

With that out of the way, let's get into the questions.

Beginner Typescript Interview Questions

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

TypeScript is a **statically typed superset of JavaScript** that compiles down to plain JavaScript.

It was developed by Microsoft to help deal with the complexities of large-scale applications and achieves this by adding static types to JavaScript, providing enhanced tooling like autocompletion, navigation, and refactoring capabilities, which makes it easier to read and debug code.

Key differences between TypeScript and JavaScript include:

- **Static Typing**: While JavaScript is dynamically typed, TypeScript uses static typing, which means that types are evaluated at compile-time instead of run-time. This can help catch potential bugs before the code is run
- **Interfaces and Classes**: TypeScript supports the latest JavaScript features, such as classes and interfaces, which are not available in some older JavaScript versions
- **Compile-time Errors**: The TypeScript compiler provides errors at compile-time, whereas JavaScript gives you an error at runtime. Early error-catching helps reduce the chance of running faulty code and speeds up bug-fixing
- **Tooling**: TypeScript has powerful tools for navigating code and automatic refactoring, which improves developer experience

#2. What are the main components of TypeScript?

There are three main components in TypeScript:

1. **Language**: The TypeScript language includes types, annotations, classes, interfaces, tuples, and more. These additions enhance JavaScript while keeping the same syntax and semantics
2. **The TypeScript Compiler**: The TypeScript compiler (`tsc`) is responsible for transpiling TypeScript code (`.ts` files) into JavaScript (`.js` files). It catches errors at compile time and

Server' provides information that helps editors and other tools provide better assistance, like autocompletion, type checking, and documentation pops

#3. What are the main benefits and disadvantages of using TypeScript?

We've hinted at a few already, but let's break them down.

Benefits of TypeScript

- **Static Typing:** TypeScript adds a strong static typing layer on top of the dynamic JavaScript, allowing for better tooling (autocompletion, type checking, etc.), leading to more robust code and a better developer experience
- **Improved Readability and Maintainability:** TypeScript's static typing and OOP features, like interfaces and classes, can make code more self-explanatory. This makes the code easier to read, understand, maintain, and refactor
- **Easier Navigation:** TypeScript makes navigating through the code easier. You can quickly go to definitions of functions, variables, and classes, making the codebase more accessible and manageable
- **Early Error Detection:** TypeScript can catch errors at compile-time before the code is run. This

can prevent many common errors that might be overlooked in JavaScript until they cause a problem at runtime

- **Better Collaboration:** TypeScript's static typing allows teams to work more efficiently. When codebases become large, and multiple developers or teams work on them, having a static type system can prevent many potential bugs and improve overall collaboration
- **Familiar Syntax for OOP Developers:** TypeScript's syntax is closer to other C-style languages (C++, C#, Java, etc.), making it easier for developers coming from those languages to learn and use TypeScript

Disadvantages of TypeScript

- **Learning Curve:** TypeScript introduces several new concepts not present in JavaScript, like static typing, interfaces, and generics. It can be slightly overwhelming for beginners or developers coming from dynamically typed languages
- **More Complexity:** TypeScript adds another layer of complexity to the development process, in that it requires a build/compile step to transpile TypeScript to JavaScript, which could complicate the build and deployment processes
- **Potentially Outdated Type Definitions:** If you're using a library that doesn't provide TypeScript support, you are reliant on type definitions provided by the community, which may be outdated or incomplete
- **Not Necessarily Required for Small Projects:** While TypeScript proves its worth in large codebases and teams, the overhead might not be necessary for small projects or projects where type safety is not a huge concern

Often, the benefits of error checking, autocompletion, and better navigation make TypeScript worth the overhead for larger projects, but it may not be worth it for small, rapid prototypes.

#5. What are variables in TypeScript, and what are the different ways of declaring them?

In TypeScript, just like in JavaScript, a variable is a symbolic name for a value. (Variables are fundamental in programming because they allow us to store and manipulate data in our programs).

You can declare a variable in TypeScript using the `let` or `const` keywords, just like in modern JavaScript. However, TypeScript introduces type annotations that enable you to explicitly specify the type of data a variable can hold:

```
let isDone: boolean = false;
let lines: number = 42;
let greetings: string = "Hello,
```

In this example, `isDone` is a boolean variable, `lines` is a number variable, and `greetings` is a string variable.

This means that if you try to assign a value of a different type to these variables, TypeScript will throw an error at compile time, like so:

```
isDone = 1;  // Error: Type 'num
lines = "forty-two";  // Error:
greetings = true;  // Error: Typ
```

This is actually a good thing.

By using type annotations like this, TypeScript can help to catch errors early, making your code more robust and maintainable.

#6. What are interfaces in TypeScript?

Interfaces provide a way to specify the structure and behaviors of any given type in TypeScript. They act as a contract in your code by defining the properties and methods that an object must implement.

This ensures that the object adheres to a specific structure, making your code more predictable and easier to debug.

For example

```
interface Greeting {
  message: string;
}

function greet(g: Greeting) {
  console.log(g.message);
}

greet({ message: 'Hello, TypeScr
```

In this example:

- `Greeting` is an interface that defines a contract for objects, specifying that any object of type `Greeting` must have a property `message` of type `string`
- The function `greet` takes an argument `g` of type `Greeting`. Since `g` must adhere to the `Greeting` interface, it guarantees that `g` will have a `message` property that is a string
- When calling `greet` with an object `{ message: 'Hello, TypeScript!' }`, TypeScript ensures that this object meets the `Greeting` interface's contract

Because the `message` property is guaranteed to exist by the TypeScript compiler, there's no need to check for `null` or `undefined` before accessing it. This makes your code safer and reduces the likelihood of runtime errors.

#7. What are classes in TypeScript? List out some

features of classes.

Classes are a fundamental component of object-oriented programming (OOP), and TypeScript includes them as a part of the language. They provide a great deal of flexibility in structuring your code. Classes in TypeScript support features such as inheritance, interfaces, access modifiers, and more.

Here's an example of a simple class:

```
class Greeter {
  greeting: string;

  constructor(message: string) {
    this.greeting = message;
  }

  greet() {
    console.log(`Hello, ${this.g
  }
}

let greeter = new Greeter('TypeS
greeter.greet(); // Outputs: Hel
```

In this example, `Greeter` is a class with three members: a property called `greeting`, a constructor, and a method `greet`. You use the `new` keyword to create instances of a class. Classes can also implement interfaces, which allow you to enforce that a class adheres to a specific contract. For example:

```
interface Animal {
  name: string;
  speak(): void;
}

class Dog implements Animal {
  name: string;

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

```
    speak() {
      console.log(`${this.name} ba
    }
  }

  const myDog = new Dog('Rex');
  myDog.speak();  // Outputs: Rex
```

In this example:

- `Animal` is an interface with a property `name` of type `string` and a method `speak`
- The class `Dog` implements the `Animal` interface, providing a concrete implementation of the `speak` method
- TypeScript ensures that `Dog` adheres to the `Animal` interface, guaranteeing that all instances of `Dog` have a `name` property and a `speak` method

By using classes and interfaces together, you can create flexible, reusable, and well-structured code. Classes in TypeScript also support features like inheritance and access modifiers (`public` , `private` , `protected`), which provide further control over how your code is organized and accessed.

#8. Are types in TypeScript optional?

Yes. TypeScript is a superset of JavaScript which means you can use static types as well as dynamic types. This flexibility means you can have strongly typed variables, function parameters, and objects.

However, you can also use the `any` type to opt out of compile-time type checking if you want to leverage JavaScript's dynamic typing.

Here's an example in TypeScript:

```
let foo: any = 42; // OK
foo = 'Hello';   // OK
foo = true;      // OK
```

In the above example, `foo` is declared with the type `any`, allowing it to hold any type of value.

But when the type is specified:

```
let foo: number = 42;   // OK
foo = 'Hello';       // Error: Ty
foo = true;          // Error: Ty
```

#9. Can you explain the concept of modules in TypeScript?

In TypeScript, just as in ECMAScript 2015, a module is a file containing variables, functions, classes, etc. The key difference is that these variables, functions, classes, etc., are private by default in TypeScript, and you decide what to expose by exporting them.

You can import functionality from other modules using the `import` keyword.

For example:

math.ts:

```
export function add(x: number, y
  return x + y;
}
```

app.ts:

```
import { add } from './math';

console.log(add(4, 5)); // Outpu
```

In the example above, the `math` module exports a function named `add`. The `app` module can then import this function and use it.

These concepts are key to building scalable applications and fostering code reuse.

#10. What is a TypeScript decorator and what is its purpose?

Decorators are a design pattern in JavaScript, and [they're available in TypeScript as well](). They're a way to modify classes or properties at design time.

TypeScript Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members. They can be used to modify classes, methods, and properties.

Here's an example of a class decorator:

```
function log(target: Function) {
  console.log(`${target.name} -
}

@log
class MyClass {}
```

In this example, `log` is a decorator that logs the name of the class or function. The `@log` syntax is then used to apply the decorator to a class.

#11. How does TypeScript handle `null` and `undefined`?

In JavaScript, `null` and `undefined` are common sources of bugs and confusion.

However, TypeScript addresses this with features like strict null checks and nullable types to help catch these issues before they cause problems.

Strict Null Checks (`--strictNullChecks`)

When strict null checks are enabled, TypeScript ensures that you check whether an object is defined before accessing its properties. This helps prevent runtime errors caused by `null` or `undefined` values.

For example:

Without strict null checks, assigning `null` or `undefined` to a variable that expects a number does not produce an error:

```
let x: number;
x = null;    // No error without
x = undefined;   // No error with
```

But, with strict null checks enabled, the above code would produce errors:

```
let x: number;
x = null;    // Error: Type 'null
x = undefined;   // Error: Type '
```

So, if you want a variable to accept `null` or `undefined`, you can use a union type.

Using Union Types

If you want a variable to accept `null` or `undefined`, you can use a union type to explicitly allow these values.

For example:

```
let x: number | null | undefined
x = null;  // Ok
x = undefined;   // Ok
```

In this example, `x` can be a `number`, `null`, or `undefined`. This explicit typing makes it clear that `x` can hold any of these values, reducing the chance of unexpected errors in your code.

#12. What are generics in TypeScript and why are they useful?

Generics are a tool for creating reusable code. They allow a type to be specified dynamically, providing a way to create reusable and general-purpose functions, classes, and interfaces without sacrificing type safety.

Here's an example of a generic function that returns whatever is passed into it:

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

console.log(identity("myString")
```

In this example, `T` acts as a placeholder for any type. The type is typically inferred automatically by TypeScript, but the can specify the exact `T` when calling the function.

Generics are useful because they enable you to write flexible and reusable code while maintaining type safety.

#13. How do you create and use enums in TypeScript?

In TypeScript, enums are a way of creating named constants, making code more readable and less error-prone.

Here's how you can define and use an enum:

```
enum Weekday {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday,
}

let day: Weekday;
day = Weekday.Monday;
```

In the example above, `Weekday` is an enum type that can contain one of the listed seven days. Enums can improve your code by enforcing that variables take one of a few predefined constants.

#14. Can you describe access modifiers in TypeScript?

In TypeScript, each class member is public by default, which means they can be accessed anywhere without any restrictions.

However, TypeScript also supports access modifiers similar to many object-oriented languages such as Java and C#. Access modifiers control the visibility and accessibility of the class members (properties, methods, etc.).

There are three types of access modifiers in TypeScript: `public`, `private`, and `protected`.

Public access modifier

Public members are visible everywhere, and it's the default access level for class members.

```
class Car {
  public color: string;

  public constructor(color: stri
  this.color = color;
  }
}
```

In this example, both the `color` property and the `constructor` method are public and can be accessed outside of the `Car` class.

Private access modifier

Private members are only accessible within the class that declares them.

```
class Car {
  private color: string;

  public constructor(color: stri
  this.color = color;
  }
}
```

Here, the `color` property is private and cannot be accessed outside of the `Car` class.

Protected access modifier

Protected members are like private members,but can also be accessed within subclasses.

```
class Vehicle {
  protected color: string;

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

class Car extends Vehicle {
  constructor(color: string) {
  super(color);
  console.log(this.color);  // A
```

```
    }
  }
```

In this example, `color` is a protected property. It's not accessible outside of the `Vehicle` class, but it is accessible within the `Car` class because `Car` extends `Vehicle`.

#15. What is a namespace in TypeScript and how is it useful?

Namespaces in TypeScript are used to group related code together, which helps in organizing and structuring your code. They also help prevent naming collisions by creating a scope for identifiers such as variables, functions, classes, and interfaces. For example:

```
namespace MyNamespace {
  export class MyClass {
    public static doSomething()
    console.log("Doing something
    }
  }
}

MyNamespace.MyClass.doSomething(
```

In this example:

- `MyNamespace` is a namespace that groups the `MyClass` class.
- The `doSomething` method is defined within `MyClass`.
- By using `MyNamespace.MyClass`, we can access the `doSomething` method, avoiding potential naming conflicts with other parts of the code.

> ***Important:*** *While namespaces are useful, ES modules have become the preferred way to organize code in modern JavaScript and TypeScript projects.*

ES Modules explained

ES modules offer better support for code splitting, lazy loading, and tree shaking, making them a more powerful and flexible solution.

For example:

Here's how you would use ES modules to achieve similar functionality:

myModule.ts:

```
export class MyClass {
  public static doSomething() {
    console.log("Doing something
  }
}
```

main.ts:

```
import { MyClass } from './myMod

MyClass.doSomething();  // Outpu
```

In this example:

- `MyClass` is exported from `myModule.ts`
- `MyClass` is imported and used in `main.ts`

Using ES modules is recommended for new projects as they provide better interoperability with modern JavaScript tools and libraries.

#16. How do you annotate functions in TypeScript?

In TypeScript, you can annotate a function by specifying the types of its parameters and its return type. This helps catch errors at compile time and makes your code more predictable and easier to understand.

For example:

```
let add: (x: number, y: number)

add = function(x, y) {
  return x + y;
};

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

In this example:

- `add` is a function that takes two parameters, `x` and `y`, both of type `number`, and returns a `number`.
- The function implementation adheres to this type annotation, so TypeScript will not throw any errors.

However, if you try to assign a function that doesn't match the type annotation, TypeScript will throw an error like so:

```
let add: (x: number, y: number)

add = function(x: number, y: str
  return x + y;
};
```

In this invalid example:

- The second parameter `y` is of type `string`, but the function type annotation specifies that it should be a `number`.
- TypeScript throws an error because the assigned function does not match the declared type annotation.

By using function annotations in TypeScript, you can write more robust and maintainable code, catching potential issues early in the development process.

#17. How does TypeScript support asynchronous programming and optional chaining?

Asynchronous Programming

TypeScript supports asynchronous programming out of the box, using promises and `async/await` syntax.

For example:

Here’s how you can use `async/await` in TypeScript.

```
async function fetchUsers() {
  const response = await fetch('
  const users = await response.j
  console.log(users);
}

fetchUsers().catch(error => cons
```

In this example:

- The `fetchUsers` function is declared as an `async` function, which means it returns a promise.
- Inside the function, the `await` keyword pauses the function execution until the `fetch` promise resolves.
- The `await` keyword can only be used inside an `async` function.
- If an error occurs, it is caught by the `catch` block attached to the `fetchUsers` call.

Optional Chaining

Optional chaining is another feature supported by TypeScript. It allows you to access deeply nested properties without worrying if an intermediate property is `null` or `undefined`.

For example:

```
type User = {
  name: string;
  address?: {
    street?: string;
    city?: string;
  }
}

const user: User = {
  name: 'John Doe',
};

console.log(user.address?.city);
```

In this example:

- `User` is a type with an optional `address` property.
- The `address` property, if it exists, may contain `street` and `city` properties.
- The optional chaining operator (`?.`) is used to access `user.address.city` .
- If `user.address` is `undefined` , the expression safely returns `undefined` instead of throwing an error.

#18. Can you briefly explain how to use TypeScript with popular libraries such as jQuery and lodash?

The process is fairly simple, thanks to DefinitelyTyped, which is a large repository of high-quality TypeScript type definitions.

Before using a JavaScript library with TypeScript, you need to install the corresponding type definition package. These packages are available on npm and usually named `@types/<library-name>` .

For example:

To use lodash with TypeScript, you would first install lodash and its type definition:

```
npm install lodash
npm install @types/lodash
```

Then you can use lodash in your TypeScript code with full type checking:

```
import _ from 'lodash';

const numbers: number[] = [1, 2,
const reversed = _.reverse(numbe
console.log(reversed);  // Outpu
```

Similarly, to use jQuery with TypeScript, you first need to install jQuery and its type definition:

```
npm install jquery
npm install @types/jquery
```

Then you can use jQuery in your TypeScript code:

```
import $ from 'jquery';

$(document).ready(() => {
  $('body').text('Hello, world!'
});
```

In both examples above:

- TypeScript knows the types of lodash and jQuery functions and objects
- This allows TypeScript to provide full autocompletion and error-checking in your editor

This is one of the key advantages of using TypeScript with popular JavaScript libraries, as it enhances the development experience by making the code more robust and easier to maintain.

Intermediate Typescript Interview Questions

#19. What is ‘type inference’ in TypeScript?

Type inference is one of TypeScript's key features, and refers to the automatic detection of the data type of an expression in a programming language. This means that you don't always have to annotate types explicitly.

TypeScript is good at inferring types in many situations. For example, in the declaration and initialization of variables, the return values of functions, and the setting of default parameters.

```
let x = 10; // `x` is inferred t
```

In the above example, TypeScript automatically infers the type of `x` is `number` from the assigned value `10`.

However, TypeScript can't always infer types in all situations.

You'll still want to provide explicit types in many cases, especially when writing function signatures, to avoid accidental incorrect typings and benefit from autocompletion and other type-checking functions.

#20. What are type guards in TypeScript?

Type guards are a way to provide additional information about the type of a variable inside a specific block of code. TypeScript supports type guards that let us determine the kind of a variable in conditional branches.

Type guards can be simple `typeof` or `instanceof` checks, or more advanced user-defined type guards.

For example:

```
function padLeft(value: string,
  if (typeof padding === "number
    return Array(padding + 1).jo
  }
  if (typeof padding === "string
    return padding + value;
  }
  throw new Error(`Expected stri
}

console.log(padLeft("Hello, worl
```

In this example, `typeof padding === "number"` and `typeof padding === "string"` are type guards. They provide extra information about the type of `padding` within their respective blocks. Inside the if-block where `typeof padding === "number"` is true, TypeScript knows that `padding` is a number.

Similarly, inside the if-block where `typeof padding === "string"` is true, TypeScript knows that `padding` is a string.

#21. Can you explain the concept of type compatibility in TypeScript?

Type compatibility is based on structural subtyping, which is a form of type checking, and is used to determine the compatibility between types based on their members.

If a type Y has at least the same members as type X, then type Y is said to be compatible with type X.

For example:

```
interface Named {
  name: string;
}
```

```
class Person {
  name: string;
}

let p: Named;
p = new Person();  // OK, becaus
```

In this example:

- `Named` is an interface with one member `name`
- `Person` is a class with one member `name`

TypeScript considers `Person` to be compatible with `Named` because `Person` has at least the same members as `Named`. This concept allows for flexible and safe type checking, enabling you to create interchangeable objects that maintain the correct constraints.

#22. How do you use and create type aliases in TypeScript?

Type aliases allow you to create new names for types. They're kind of similar to interfaces, but they can also name primitives, unions, tuples, and any other types that you'd otherwise have to write by hand.

You can declare a type alias using the `type` keyword:

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

Here, we've created a type alias `Point` for an object type that has `x` and `y` properties, both of type `number`.

Once you've defined a type alias, you can use it in places where you would use any other type:

```
function drawPoint(p: Point) {
  console.log(`The point is at p
}
```

```
drawPoint({ x: 10, y: 20 }); //
```

In this function `drawPoint`, we've used the `Point` type alias as the type for the parameter `p`.

#23. Can you explain the difference between type and interface in TypeScript?

In TypeScript, both `type` and `interface` can be used to define custom types, and they have a lot of overlap.

You can define both simple and complex types with both `type` and `interface`. For example, you can use both to represent a function type or an object type.

The key differences between `type` and `interface` are as follows:

Declaration merging differences

TypeScript allows interface declarations with the same name to be merged. It's a key advantage of interfaces; but, you can't do this with type aliases.

```
interface User {
  name: string;
}

interface User {
  age: number;
}

let user: User = {
  name: 'John Doe',
  age: 30,
};
```

Syntax and feature differences

`type` alias has a few more features than `interface`. It supports union (`|`), intersection (`&`), `typeof`, etc, which allows you to create more complex types.

```
type User = {
  name: string;
} & {
  age: number;
};

let user: User = {
  name: 'John Doe',
  age: 30,
};
```

TL;DR

If you want to create complex type combinations, you'd use `type`. If you're describing object shapes, `interface` is usually the better choice.

#24. What is a discriminated union in TypeScript and how is it implemented?

Discriminated Unions, also known as tagged unions or algebraic data types, are types that can represent a value that could be one of several different types. However, each type in the union is marked with a distinct literal type, hence the term 'discriminated'.

Discriminated Unions typically involve a `type` or `kind` field that acts as the discriminant. Here's how to implement a Discriminated Union:

```
type Cat = {
  type: 'cat';
  breeds: string;
};

type Dog = {
  type: 'dog';
  breeds: string;
```

```
};

type Pet = Cat | Dog;

function printPet(pet: Pet) {
  switch (pet.type) {
  case 'cat':
    console.log(`Cat breeds: ${p
    break;
  case 'dog':
    console.log(`Dog breeds: ${p
    break;
  }
}

printPet({ type: 'cat', breeds:
```

In this example, `Pet` can be either a `Cat` or a `Dog`. The `printPet` function uses the `type` field (the discriminant) to determine the specific type of `Pet` and access the corresponding `breeds` property.

Discriminated Unions are an effective way of handling different types of objects, and they showcase TypeScript's strengths in static type checking. They also enable you to work with only the data that's necessary for a given section of your code, increasing functionality and efficiency.

#25. How do you implement method overloading in TypeScript?

Method overloading allows for multiple methods with the same name but different parameters or types. This then provides the ability to manipulate data in different ways based on the input type.

For example

In TypeScript, method overloading is a bit different than in other languages. Here's how it's done:

```
class Logger {
    // Method overload signature
```

```
        log(message: string): void;
        log(message: object): void;

        // Implementation of the met
        log(message: string | object
            if (typeof message === '
                console.log(`String
            } else if (typeof messag
                console.log(`Object
            }
        }
    }

    const logger = new Logger();

    logger.log("This is a string mes
    // Output: String message: This

    logger.log({ key: "value", anoth
    // Output: Object message: {"key
```

In the `Logger` class, we're declaring two overloaded methods `log(message: string)` and `log(message: object)`. We then have an implementation of the `logger` method that accepts either a `string` or `object` type.

This implementation will be used when the `logger` method is called inside the implementation, and we use type guards to check the type of `message` and handle each case separately.

While method overloading in TypeScript allows you to create cleaner and more explicit APIs, it's essential to note that TypeScript does this syntactic checking at compile time and then compiles the overloads into a single JavaScript function.

#26. What are mapped types in TypeScript and how are they used?

In TypeScript, a mapped type is a generic type which uses a union created from another type (usually an interface) to compute a set of properties for a new type.

Here's an example of a mapped type that turns all properties in `T` into readonly properties. We can use it to create a read-only `User`:

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

type User = {
  name: string;
  age: number;
};

let user: ReadOnly<User> = {
  name: "John",
  age: 30
};

user.name = "Jim";  // Error: Ca
```

Mapped types allow you to create new types based on old ones by transforming properties. They're a great way to keep your code DRY (Don't Repeat Yourself), making your code more maintainable and easier to read.

#27. What are conditional types in TypeScript?

Conditional types in TypeScript allow you to introduce type variables and type inference within types, making them incredibly powerful for building complex type logic.

A conditional type selects one of two possible types based on a condition expressed as a type relationship test. They are written in the form `T extends U ? X : Y`. So if `T extends U` is true, then the type `X` is selected. But if `T extends U` is false, then `Y` is selected.

Here's an example:

```
type TypeName<T> =
  T extends string ? 'string' :
  T extends number ? 'number' :
  T extends boolean ? 'boolean'
  T extends undefined ? 'undefin
  'object';

type T0 = TypeName<string>;  //
type T1 = TypeName<'a'>;  // "st
type T2 = TypeName<true>;  // "b
type T3 = TypeName<() => void>;
```

Here, `TypeName<T>` is a conditional type that checks if `T` extends certain types and resolves to a string literal of the type name.

Conditional types can be chained with more conditional types to create more complex type relations. Also, TypeScript includes an `infer` keyword, that allows you to infer a type inside your conditional type.

#28. How does TypeScript handle rest parameters and spread operators?

TypeScript supports rest parameters and spread operators, similar to ES6.

Rest parameters

A rest parameter is a function parameter that allows you to pass an arbitrary number of arguments to the function. In TypeScript, you can include type annotations for rest parameters.

For example:

```
function sum(...numbers: number[
  return numbers.reduce((a, b) =
}
```

```
    sum(1, 2, 3, 4, 5);  // Outputs:
```

In this example:

- `...numbers: number[]` is a rest parameter
- It takes any number of arguments and packs them into an array of `numbers`

Spread operators

The spread operator is used to spread elements of an array or object properties of an object. In TypeScript, you can spread arrays of any type.

For example:

```
    let arr1 = [1, 2, 3];
    let arr2 = [...arr1, 4, 5, 6];
```

In this example:

- `...arr1` is a spread operator
- It takes an array `arr1` and spreads its elements into a new array `arr2`

TL;DR

By utilizing rest parameters and spread operators, TypeScript enables developers to write more expressive and flexible code, especially when dealing with arrays and function arguments.

Rest parameters allow functions to handle an indefinite number of arguments, while spread operators enable more expressive and concise code when working with arrays and objects

These features enhance code reusability and readability, making it easier to work with collections of data

#29. What is the `keyof` keyword and how is it used in TypeScript?

In TypeScript, the `keyof` keyword is an index type query operator. It yields a union type of all the known, public property names of its operand type.

Let's take a look at an example:

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

type UserProps = keyof User;  //
```

In this example, `UserProps` will be a type representing all property names of `User`, i.e., the union type `"name" | "age"`.

The `keyof` keyword is handy here when you want to write functions that can handle any property of an object, regardless of its name.

For example:

```
function getProperty<T, K extend
  return obj[key];
}

let user: User = {
  name: 'John Doe',
  age: 30,
};

let userName: string = getProper
let userAge: number = getPropert
```

In this `getProperty` function, the `keyof` keyword is used to ensure that the `key` argument is a valid property name of `obj`. The function then returns the value of that property.

#30. How do you use TypeScript's utility types like Partial, Readonly, and Record?

TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally. Some of the most commonly used utility types are `Partial`, `Readonly`, and `Record`.

Partial

The `Partial` type takes in another type as its parameter, and makes all of its properties optional.

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

let partialUser: Partial<User> =
```

Here, `partialUser` is of type `Partial<User>`, so both `name` and `age` properties are optional.

Readonly

The `Readonly` type makes all properties of a type readonly, meaning they can't be reassigned after creation.

```
let readonlyUser: Readonly<User>
readonlyUser.age = 31;  // Error
```

Record

The `Record` utility type helps create a map where all entries are keyed on a specified type. This is useful when you want to associate some key (like a database ID) with a value (record):

```
interface User {
│ name: string;
│ age: number;
}

type UserRecord = Record<number,
let users: UserRecord = {
  1: {
    name: 'John Doe',
    age: 30
  },
  2: {
    name: 'Jane Doe',
    age: 29
  }
};
```

In this example, `UserRecord` is an object type containing values of type `User` keyed on a number.

#31. How do you handle errors and exceptions in TypeScript?

Exceptions

Exception handling in TypeScript is similar to JavaScript. You can use `try` / `catch` / `finally` blocks to handle errors and exceptions.

Here's a simple example:

```
try {
  throw new Error('This is an er
}
catch (error) {
  console.error(error.message);
}
finally {
  console.log('This always runs!
}
```

In this example:

- The `try` block throws an error
- The `catch` block catches the error and logs its message to the console
- The `finally` block always runs, regardless of whether an error occurred or not

Static types in error handling

TypeScript also brings static types to JavaScript, and you can use them in error handling. For instance, you can create custom error classes that extend the built-in `Error` class and add custom properties to them.

For example:

```
class CustomError extends Error
  constructor(public code: numbe
  super(message);
  }
}

try {
  throw new CustomError(404, 'Re
}
catch (error) {
  if (error instanceof CustomErr
  console.error(`Error ${error.c
  }
}
```

In this example:

- The `CustomError` class extends `Error` and adds a `code` property to it
- The `catch` block checks if the error is an instance of `CustomError`
- If the error is an instance of `CustomError`, it logs the error code and message to the console

Using TypeScript's static typing and custom error classes enhances type safety, provides clearer error messages, and improves debugging by offering more context-specific information about the error.

This approach allows you to create more robust and maintainable error-handling mechanisms in your code.

#32. Can you explain how to implement a Generic Class in TypeScript?

In TypeScript, a generic class refers to a class that can work with a variety of data types, rather than a single one. This allows users of the class to specify the type they want to use.

Here's an example of a generic class:

```
class DataStore<T> {
  private data: T[] = [];

  add(item: T): void {
    this.data.push(item);
  }

  remove(item: T): void {
    const index = this.data.inde
    if (index > -1) {
        this.data.splice(index,
    }
  }

  getAll(): T[] {
    return this.data;
  }
}
```

In this example:

- `DataStore` is a generic class, denoted by the `<T>` next to its name.
- `T` represents any type.
- Inside the class, `T` is used as if it were a regular type.

When we create an instance of `DataStore`, we can specify what `T` should be.

For example:

Here’s how we can create a `DataStore` that works with numbers:

```
const numberStore = new DataStor
numberStore.add(5);
console.log(numberStore.getAll()

numberStore.remove(5);
console.log(numberStore.getAll()
```

#33. How does TypeScript support Dynamic Module Loading?

Dynamic module loading in TypeScript is supported through the ES6 dynamic import expression. The dynamic `import()` expression allows us to load modules on-demand, which can result in significant performance benefits by minimizing the startup cost of your application.

The dynamic import expression returns a promise that resolves to the module. Here is a basic example:

```
// Loads the module "./utils" dy
import('./utils').then((utils) =
  // Use the module
  console.log(utils.divide(10, 2
}).catch((error) => {
  console.log(`An error occurred
});
```

In this example, the `utils` module is loaded dynamically. Once it's loaded, the promise is resolved, and the `then` callback is executed, where we can use the module. If an error occurs while loading the module, the `catch` callback is triggered.

Dynamic module loading provides a way to load modules on the fly, which can be useful for code splitting or lazy loading modules in an application.

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

In TypeScript, index types allow you to create complex types that are based on the properties of another type.

The `keyof` operator is used to create a type representing the property names of a given type:

```
type Person = {
  name: string;
  age: number;
};

type PersonKeys = keyof Person;
```

The `indexed access operator` is used to access the property type of a specific type:

```
type PersonNameType = Person['na
```

Putting these together, you could create a function that takes an object, a property of that object, and a new value for the property, and types it correctly:

```
function setProperty<T, K extend
  obj[key] = value;
}

let person: Person = {
  name: 'John',
  age: 30
};

setProperty(person, 'name', 'Jan
setProperty(person, 'age', '35')
```

This function ensures type safety, meaning that it only allows setting properties that exist on the object, and the value must be of the right type.

#35. How do you work with read-only arrays and objects in TypeScript?

TypeScript provides built-in utility types `ReadonlyArray<T>` and `Readonly<T>` for representing read-only array and object types, respectively.

Read-Only Arrays

You can define a read-only array using the `ReadonlyArray<T>` generic type where `T` is the type of array elements:

```
const arr: ReadonlyArray<number>
```

However, once defined as `ReadonlyArray`, you can't alter the array. This means that any attempt to mutate the array such as push, pop, or assignment will result in a compile-time error:

```
arr.push(6);  // Error: Property
arr[0] = 10;  // Error: Index si
```

Read-Only Objects

Similarly, you can make the properties of an object read-only using the `Readonly<T>` generic utility type where `T` is the type of the object:

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

const point: Readonly<Point> = {
  x: 10,
  y: 20
};
```

Any attempt to change the properties of `point` will result in a compile-time error:

```
point.x = 30; // Error: Cannot a
```

The `Readonly` and `ReadonlyArray` types ensure immutability, which is particularly useful in functional programming and can be beneficial for state management in large applications like those made with React and Redux.

#36. What is the significance of the 'never' type in TypeScript?

The `never` type in TypeScript represents a value that never occurs. It is the return type for functions that never return or for functions that always throw an exception.

For instance, if you have a function that always throws an error and never returns a value, you can type it like this:

```
function throwError(message: str
  throw new Error(message);
}
```

Similarly, for a function that contains an infinite loop and never reaches its end:

```
function infiniteLoop(): never {
  while(true) {
    console.log('Never ending lo
        }
    }
```

The `never` type is a subtype of all types and can be assigned to any other types; however, no types can be assigned to `never` (except `never` itself).

This can be useful for exhaustiveness checks in switch-case constructs where TypeScript can ensure that every case is handled.

Advanced Typescript Interview Questions

#37. How do you use TypeScript with external libraries that don't have type definitions?

When working with external JavaScript libraries that do not have type definitions, TypeScript might display errors since it cannot understand the types from these libraries.

To get around this, you can create a declaration file with a `.d.ts` extension in your project to tell TypeScript how the module is shaped. For instance, if you are using a JavaScript library called `jsLibrary`, you would create a `jsLibrary.d.ts` file:

```
declare module 'jsLibrary';
```

By declaring the module, you tell TypeScript to treat this module as `any` type and thus bypass the type checking.

For more complex libraries, TypeScript provides an option to install type definitions from DefinitelyTyped using npm also:

```
npm install --save @types/librar
```

Here `library-name` is the name of the JavaScript library, and the `@types` organization on npm is managed by the DefinitelyTyped project, which provides high-quality TypeScript definitions for a multitude of JavaScript libraries.

However, not all libraries have types available on DefinitelyTyped, and in such cases, creating your own declaration file is the way to go.

To create a more detailed declaration file, you can define the library's types and functions:

```
declare module 'jsLibrary' {
  export function libFunction(ar
}
```

This declares a function `libFunction` in `'jsLibrary'` that takes a `string` and returns a `number`. You can add more functions and types as required by the library you're using.

#38. Can you explain ambient declarations in TypeScript?

Ambient declarations are a way of telling the TypeScript compiler that some other code not written in TypeScript will be running in your program.

With the help of ambient declarations, you can opt out of the type checking for certain values, or inform the compiler about types it can't know about. This is commonly done in a `.d.ts` file, or an ambient context file, where you can declare types, variables, classes or even modules.

Here's an example of an ambient variable declaration, that might be found in a `.d.ts` file:

```
declare let process: {
  env: {
    NODE_ENV: 'development' | 'p
    [key: string]: string | unde
  }
};
```

In this example, we're declaring a global `process` object that has a property `env`, which is another object. This `env` object has a known property `NODE_ENV` that can either be 'development' or 'production', and can have any number of string properties.

You can also use ambient declarations to describe types of values that come from a third-party JavaScript library, which doesn't provide its own type definitions.

Here's an example of how to declare a class in a third-party library:

```
declare class SomeLibrary {
  constructor(options: { verbose
  doSomething(input: string): nu
}
```

Here we're telling TypeScript that there will be a `SomeLibrary` class, it has a constructor that takes an options object, and has a `doSomething` method.

This helps ensure type safety when using the `SomeLibrary` class in your TypeScript code.

#39. What are TypeScript declaration files and how do you use them?

TypeScript declaration files (`.d.ts` files) are a means to provide a type layer on top of existing JavaScript libraries, or for defining types for environmental variables that exist in runtime environments outside of our own TypeScript code.

In simple terms, a declaration file is for telling TypeScript that we are using some external module that isn't written in TypeScript.

Here is an example of a declaration file (`index.d.ts`) for a module named `someModule`:

```
declare module 'someModule' {
  export function someFunction(i
}
```

This `index.d.ts` file tells TypeScript about the shape and the types used in the `someModule` module.

When you use the `someModule` in your TypeScript code like this:

```
import { someFunction } from 'so

let result: string = someFunctio
```

TypeScript will use your declaration file to check that your usage of `someModule` is correct - that you're importing a function that does exist in the module, and that you're using the result as a `string`.

A TypeScript declaration file can be complex and can contain declarations for modules, classes, functions, variables, etc. They serve a crucial role when using JavaScript libraries in TypeScript, or when you need to use global variables that exist in the runtime environment.

#40. How does TypeScript handle function overloads with different signatures?

In TypeScript, function overloading or method overloading is the ability to create multiple methods with the same name but with different parameter types and return type.

For example:

```
function add(a: number, b: numbe
function add(a: string, b: strin

function add(a: any, b: any): an
    if (typeof a === 'string' &&
        return a.concat(b);
    }
    if (typeof a === 'number' &&
        return a + b;
    }
}

let result1 = add(10, 20);  // r
let result2 = add('Hello', ' Wor
```

Here, the `add` function is overloaded with different parameter types.

When two numbers are passed, it returns the sum of numbers When two strings are passed, it concatenates and returns the string

The compiler uses the number of parameters and their types to determine the correct function to call.

Function overloading allows you to call the same function in different ways, which can make your code easier to read and use. However, you should take care not to overload functions too often, as it can make your code more complex and harder to maintain.

Always make sure to document your function's behavior well, so other developers know what to expect when they use your overloaded function.

#41. Can you explain what type intersection is in TypeScript, and provide an example?

Type intersection in TypeScript is a way to combine multiple types into one. The resulting type has all the properties and methods of the intersected types. This is particularly useful when you want to create a new type that combines the features of multiple existing types.

The syntax for creating an intersection type is to use the `&` operator between the types you want to combine:

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

interface Admin {
  adminLevel: number;
}

type AdminUser = User & Admin;

const admin: AdminUser = {
  name: "Alice",
  age: 30,
  adminLevel: 1
};
```

In this example, the `AdminUser` type combines the properties of both `User` and `Admin`.

How did you do?

Did you get all 41 correct?

It’s ok if you couldn’t answer all of these. Not even the most senior programmers have all the answers, and need to Google things!

You're interviewer won't expect you to know everything.

Whats most important is to answer interview questions with confidence, and if you don't know the answer, **use your experience to talk through how you're thinking about the problem and ask follow-up questions if you need some help**.

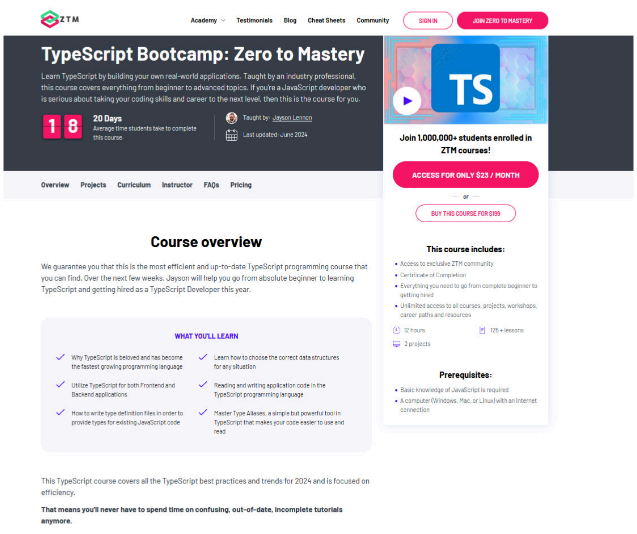
This shows the interviewer that you can think through situations instead of giving up and saying "I don't know". They don't expect you to know 100%, but they do want people who can adapt and figure things out.

Someone who puts in the effort to answer questions outside their scope of knowledge is much more likely to get hired than someone who gives up at the first sign of trouble.

So good luck, and remember to stay calm. You've got this!

P.S.

If absolutely none of these questions made sense, or if you simply want to dive deeper into TypeScript and build some more impressive projects for your portfolio, then come and check out my complete TypeScript Developer course.



TypeScript Bootcamp: Zero to Mastery

Learn TypeScript by building your own real-world applications. Taught by an industry professional, this course covers everything from beginner to advanced topics. If you're a JavaScript developer who is serious about taking your coding skills and career to the next level, then this is the course for you.

1 8 20 Days
Average time students take to complete this course.

Taught by: Jayson Lennon

Last updated: June 2024

Join 1,000,000+ students enrolled in ZTM courses!

Course overview

We guarantee you that this is the most efficient and up-to-date TypeScript programming course that you can find. Over the next few weeks, Jayson will help you go from absolute beginner to learning TypeScript and getting hired as a TypeScript Developer this year.

WHAT YOU'LL LEARN

- ✓ Why TypeScript is beloved and has become the fastest growing programming language
- ✓ Learn how to choose the correct data structures for any situation
- ✓ Utilize TypeScript for both Frontend and Backend applications
- ✓ Reading and writing application code in the TypeScript programming language
- ✓ How to write type definition files in order to provide types for existing JavaScript code
- ✓ Master Type Aliases, a simple but powerful tool in TypeScript that makes your code easier to use and read

This TypeScript course covers all the TypeScript best practices and trends for 2024 and is focused on efficiency.

That means you'll never have to spend time on confusing, out-of-date, incomplete tutorials anymore.

This course includes:

- Access to exclusive ZTM community
- Certificate of Completion
- Everything you need to go from complete beginner to getting hired
- Unlimited access to all courses, projects, workshops, career paths and resources

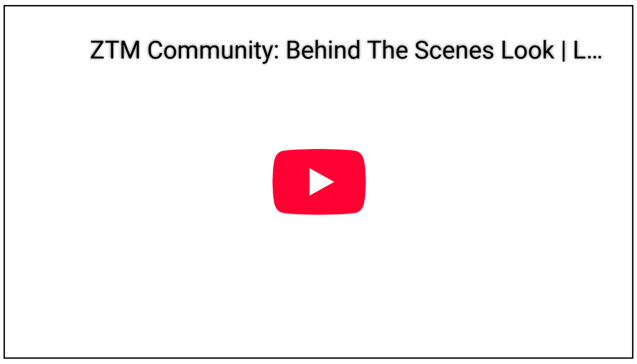
12 hours
125 + lessons
2 projects

Prerequisites:

- Basic knowledge of JavaScript is required
- A computer (Windows, Mac, or Linux) with an internet connection

Taught by an industry professional (me!), this course covers everything from beginner to advanced topics. So if you're a JavaScript developer who is serious about taking your coding skills and career to the next level, then this is the course for you.

You'll not only get access to step-by-step tutorials, but you can ask questions from me and other instructors, as well as other students inside our private Discord.



Either way, if you decide to join or not, good luck with your interview and go get that job!

BONUS: More TypeScript tutorials, guides & resources

If you've made it this far, you're clearly interested in TypeScript so definitely check out all of my TypeScript posts and content:

- TypeScript Cheat Sheet
- TypeScript Generics Explained: Beginner's Guide With Code Examples
- TypeScript Interview Questions + Answers (With Code Examples)
- TypeScript vs. JavaScript... Which Is Better and Why?
- TypeScript Arrays: Beginners Guide With Code Examples
- TypeScript Utility Types: A Beginners Guide (With Code Examples)
- TypeScript Union Types
- Type Checking In TypeScript

over 300,000 ZTM email subscribers, you'll receive exclusive ZTM posts, opportunities, and offers.

Email SUBSCRIBE

No spam ever, unsubscribe anytime

TypeScript Bootcamp: Zero to Mastery

Learn TypeScript from scratch by building your own real-world apps. This course is for developers serious about taking their coding skills and career to the next level.

Build a ChatBot with Nuxt, TypeScript, and the OpenAI Assistants API

Take your skills to the next level and build an AI-powered customer support ChatBot! You'll learn to utilize the OpenAI Assistants API to power a chatbot that's fine-tuned for your specific product or service.

Build an Instagram

More from Zero To Mastery

TypeScript Arrays: Beginners Guide With Code Examples

10 min read

Looking to improve your TypeScript skills? Learn how to store and access multiple data types, with this beginner's guide to using arrays in TypeScript.

Jayson Lennon

Tutorials Web Development

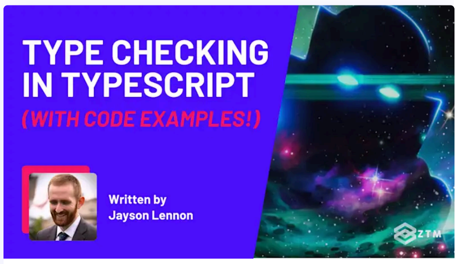
TypeScript vs. JavaScript... Which Is Better and Why?

16 min read

You should 100% learn both. But these are the key pros and cons of each (with code examples) so you know when best to use each of them for your next project!

Jayson Lennon

Web Development



Type Checking In TypeScript: A Beginners Guide

20 min read

What if you could catch errors at both compile and runtime? Thanks to TypeScript's type checking feature, you can! Learn how in this guide, with code examples.

Jayson Lennon

Tutorials Web Development

Lifetime

Challenge

Learning

Passport

Excellent

4.9 out of 5