TYPE-SCRIPT

TypeScript Course Outline

1. What is TypeScript?
2. TypeScript vs JavaScript
3. Setup VS Code Editor
4. How TypeScript Works & Why?
5. Pros & Cons of TypeScript
6. Setup TypeScript Compiler
7. Debugging TypeScript Applications
8. Basic Types
9. any type
10. Arrays
11. Tuples
12. Enums
13. unknown type
14. never type
15. void type
16. Type Inference & Type Assertions
17. Union types
18. Type Narrowing
19. Interfaces & Type Aliases
20. Optional fields

TypeScript Course Outline

21. Functions
22. Practice problems (4)
23. Classes & Access Modifiers
24. Practice problems (2)
25. Generics
26. Literal types
27. Type guards
28. keyof operator
29. index signature
30. Utility types
31. Modules & Namespaces
32. Decorators
33. Async Programming
34. Non-null Assertion
35. Interview Based Questions

What is TypeScript?

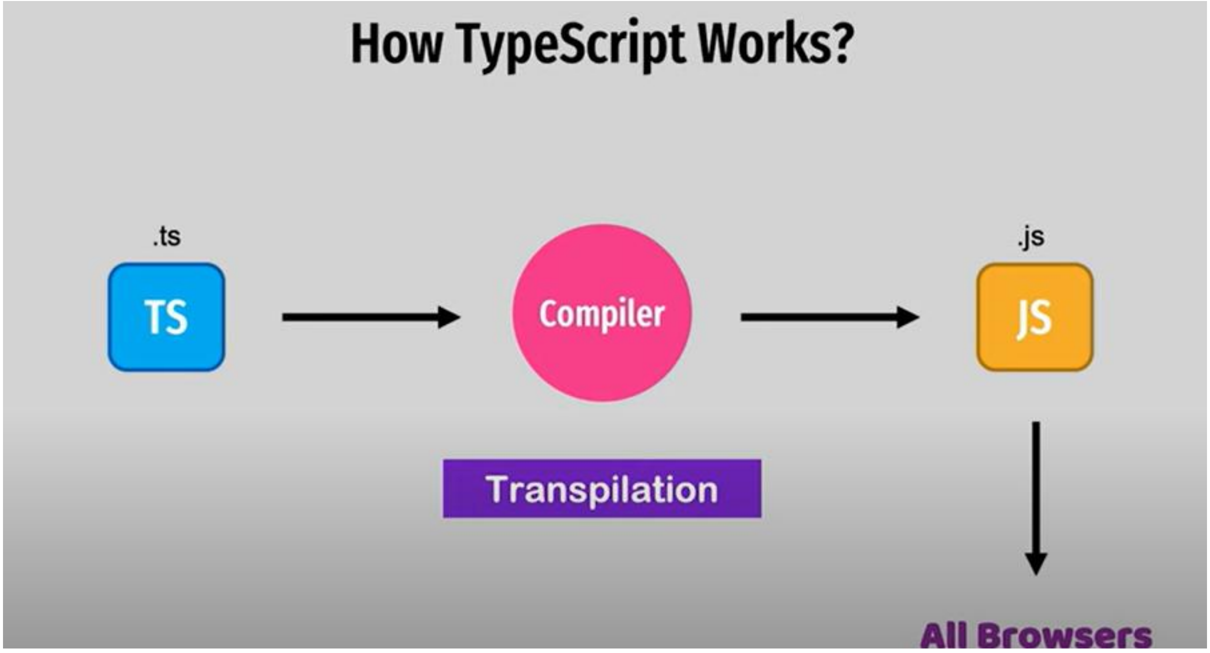
- A strongly-typed superset of JavaScript
- Adds Static Typing to the language & other features too
- Provides better error checking, enhanced tools, improved code readability

TypeScript vs JavaScript

	TYPESCRIPT	JAVASCRIPT
Type System	Statically typed (supports type annotations)	Dynamically typed
Compilation	Needs to be compiled to JavaScript	Directly executed by browsers and Node.js
Error checking	Compile-time error checking	Errors occur at runtime
Support for OOP	Better support with interfaces, classes, generics	Basic support with prototype-based OOP
Learning Curve	Higher due to static types and additional features	Easier to learn as it's simpler
Code Scalability	Easier to scale with static types, interfaces, and strong typing	More challenging to scale due to dynamic typing

TypeScript vs JavaScript

	TYPESCRIPT	JAVASCRIPT
Use Case	Suitable for large, complex applications	Ideal for smaller projects and fast prototyping
Interoperability	Can use JavaScript libraries and code	Fully interoperable with TypeScript
Development Speed	Slower due to type checks and compiling	Faster as no compilation is required
Community	Growing, particularly for large-scale applications	Large and well-established
Tooling	Rich tooling and editor support with autocompletion and refactoring	Basic tooling support
ES6+ features	Includes all ES6+ features and future proposals	Includes ES6+ features, but depends on browser/Node.js version



Why TypeScript Compiles to JavaScript ?

- **JavaScript** is the language that browsers and Node.js understand natively
- Browsers cannot directly execute TypeScript code, so it needs to be converted into JavaScript. This process is called **Transpilation**.
- Key Reasons:
    - JavaScript Compatibility
    - Cross-Platform Execution
    - Leveraging JavaScript Libraries
    - Backward Compatibility
    - Type Safety without Runtime Changes

JavaScript Compatibility

- **Why?**: TypeScript is a **superset** of JavaScript, meaning it adds extra features (like static typing) on top of JavaScript's syntax. However, under the hood, it's still JavaScript.

- Example:

TypeScript

```
let message: string = "Hello, world!";
console.log(message);
```

JavaScript

```
var message = "Hello, world!";
console.log(message);
```

Here, the **: string** type annotation in TypeScript is removed during compilation, leaving regular JavaScript.

Cross-Platform Execution

- **Why?**: JavaScript runs in every browser and on many platforms like Node.js. So, converting **TypeScript** into **JavaScript** ensures the code will run on all these platforms without any extra dependencies.

- **Example**: A web application developed with TypeScript will be converted to JavaScript so that it can be run on all browsers, no matter if they support TypeScript or not.

Leveraging JavaScript Libraries

- **Why?**: Many libraries, like React, jQuery, or D3.js, are written in JavaScript. By compiling to JavaScript, TypeScript can easily integrate with these libraries without any special changes.
- **Example**:

TypeScript

```
// TypeScript function using jQuery
$('#myButton').on('click', function() {
    console.log('Button clicked!');
});
```

This will compile into JavaScript and work seamlessly with jQuery.

Backward Compatibility

- **Why?**: TypeScript supports modern JavaScript features (ES6+) even when older browsers might not. During compilation, TypeScript converts these modern features into a form that older browsers can understand.
- **Example**: TypeScript with ES6 arrow function:

```
const greet = () => console.log("Hello, world!");
```

Compiled JavaScript for older browsers (using a regular function):

```
var greet = function() {
console.log("Hello, world!");
};
```

Type Safety without Runtime Changes

- **Why?**: TypeScript's static types help developers catch errors during development but don't affect the performance of the running code. The types are stripped away in the final JavaScript, so there's no overhead when the program runs.
- **Example**:

TypeScript checking types during development:

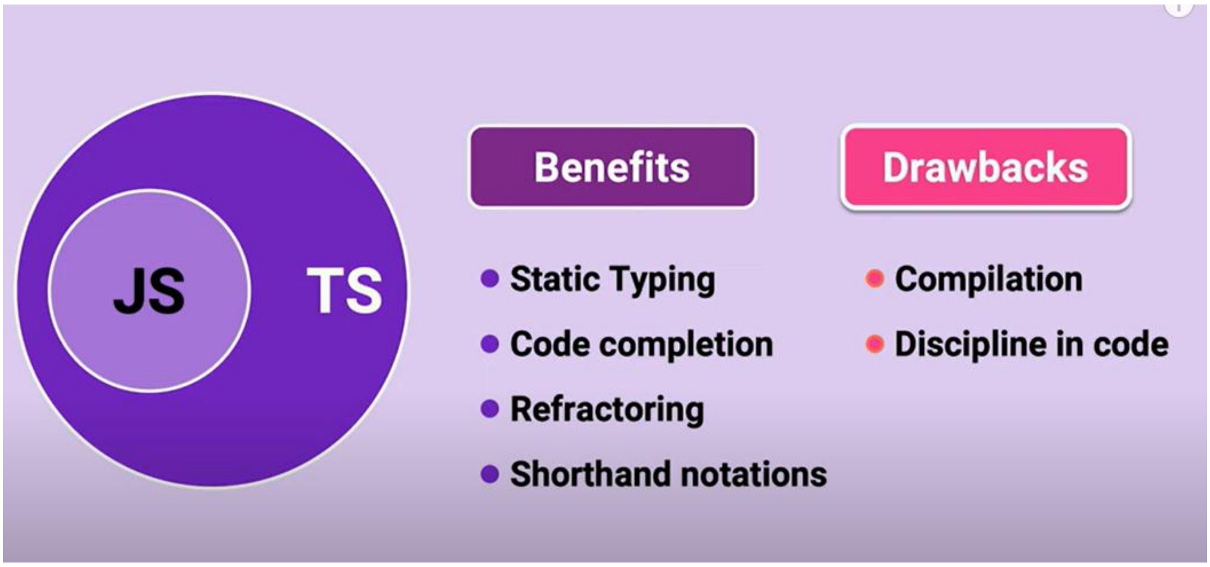
```
let age: number = "25"; // Error: "25" is not a number
```

Compiled JavaScript for older browsers (using a regular function):

```
var age = "25";
```

Conclusion

- **TypeScript** compiles to **JavaScript** to ensure the code can run on any environment that supports JavaScript, maintain compatibility with existing JavaScript libraries, and leverage static typing and modern features without sacrificing runtime performance.



Module Systems in TypeScript

Module System	Usage	Syntax	Support	Pros	Cons	Recommended For
commonjs	Default for Node.js	`require()`, `module.exports`	All Node.js versions, many npm packages	Simple, widely used in server-side JS	Synchronous loading, not suitable for modern browsers	Server-side Node.js apps, older projects
esnext	Latest ECMAScript standard	`import`, `export`	Modern browsers, Node.js 12+	Async module loading, modern tooling support	May need polyfills for older browsers	Modern web apps, Node.js 12+ projects
es2022	ECMAScript 2022 module system	`import`, `export`	Modern browsers, Node.js 18+	Latest ECMAScript features, async loading	Compatibility issues with older environments	Cutting-edge projects targeting modern environments
amd	Browser projects using RequireJS	`define()`, `require()`	Browsers with RequireJS	Asynchronous module loading for browsers	Outdated in modern web dev	Legacy browser apps using RequireJS

Module Systems in TypeScript

Module System	Usage	Syntax	Support	Pros	Cons	Recommended For
system	SystemJS for ES6 modules	`System.import()`	Browsers with SystemJS	Useful for older browsers or environments	Outdated, adds complexity	Legacy projects with SystemJS
umd	Universal (browser & Node.js)	Supports both `require()` and `define()`	Node.js and browsers	Cross-environment compatibility	Adds boilerplate code for both environments	Libraries targeting both Node.js and browser environments
none	No module system	No imports/exports	Custom environments	Used in custom setups	Rarely used in modern projects	Special use cases with custom tooling

Built-in Types

JavaScript

- number
- string
- boolean
- null
- undefined
- object

TypeScript

- any
- unknown
- never
- enum
- tupple

Feature	Interface	Type Alias
Can define union types	No	Yes
Can define intersection types	No (can extend multiple interfaces)	Yes
Extending other types	Via "extends" keyword	Via intersections "&"
Merging declarations	Yes	No
Used for primitives & other types	No (object structure only)	Yes
Flexibility	More restrictive (specific to objects & classes)	More flexible (unions, intersections)

Feature	Module	Namespace
Scope	File-based	Global (in-memory)
Syntax	import / export	namespace
Use Case	External dependencies	Internal, large projects
Benefits	Better for large-scale applications	Organizes internal code

Question 1

- What is **TypeScript**?
- TypeScript is a superset of JavaScript that
- adds static types, allowing for improved code quality and error checking before runtime.
- It supports features like interfaces, enums, generics, and more.

Question 2

What is the key difference between `interface` and `type` aliases in TypeScript?

- `interface` is best for object shapes and is extendable.
- `type` aliases can represent any type (including primitives), and intersections and unions can be defined with them.
- Interfaces can't represent unions.

Question 3 (MCQ)

Which of the following is the correct way to define a function in TypeScript?

A. function myFunc(name: string): void { console.log(name); }
B. function myFunc(name: string): string { console.log(name); }
C. function myFunc(name): void { console.log(name); }
D. function myFunc(name): any { console.log(name); }

Answer:

A. function myFunc(name: string): void { console.log(name); }
Why?: Option A specifies both the parameter type and return typ

Question 4

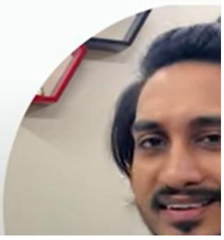
What is the purpose of the '***readonly***' modifier in TypeScript?

The '***readonly***' modifier ensures that
a property cannot be modified after it has been initialized,
providing immutability.

Question 5 (MCQ)

Which TypeScript utility type can be used to make all properties of an interface optional?

A. Partial<T>
B. Required<T>
C. Readonly<T>
D. Omit<T, K>



Answer:

A. Partial<T>

Question 6

How does TypeScript's *'**any**'* differ from *'**unknown**'*?

Both *'**any**'* and *'**unknown**'* can represent any type,

but *'**unknown**'* is safer since you cannot perform operations on an 'unknown' value without narrowing its type first,

while *'**any**'* disables type checking.

Question 7

What is **type inference** in TypeScript?

Type inference allows TypeScript to automatically determine a variable's type based on its value,

reducing the need for explicit type annotations.

Question 8 (MCQ)

What is the output of the following TypeScript code?

```
let x: unknown = 'hello';
console.log(x.toUpperCase());
```

A. 'undefined'
B. 'HELLO'
C. Error: Object is of type 'unknown'
D. 'hello'

Answer:

C. Error: Object is of type 'unknown'

Question 9

What are **generics** in TypeScript?

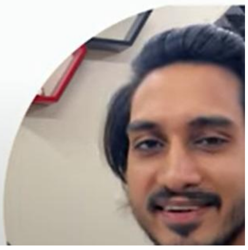
Generics allow creating reusable components that work with a variety of types, enabling flexibility and type safety.

They can be used with functions, classes, and interfaces.

Question 10 (MCQ)

Which keyword is used to assert that a value has a specific type in TypeScript?

A. typeof
B. instanceOf
C. declare
D. as



Answer:

D. as
Why?: 'as' is used for type assertions in TypeScript.

Question 11 (MCQ)

Which of the following is a correct way to use conditional types in TypeScript?

(Hint: Conditional types allow you to apply a logic to types based on whether one type ***extends*** *another.)*

A. type Check<T> = T extends string ? boolean : number;
B. type Check<T> = T is string ? boolean : number;
C. type Check<T> = T matches string ? boolean : number;
D. type Check<T> = typeof T === 'string' ? boolean : number;

Answer:
A. type Check<T> = T extends string ? boolean : number;

Why?: Conditional types in TypeScript use the syntax T extends U ? X : Y to dynamically infer types.

Question 12

How can you prevent a class property from being modified in TypeScript?

You can use the `***readonly***` modifier to prevent a class property from being modified after initialization.

Question 13 (MCQ)

Which of the following types allows creating a union of all possible string literal values from an object type in TypeScript?
(Hint: In TypeScript, an object type is a collection of properties (or keys) with values)

A. keyof typeof T
B. keyof T
C. keyUnion<T>
D. keyType<T>

Answer:
A. keyof typeof T

Why?: `keyof typeof T` returns a union of all string literal keys of an object type.
typeof gets the type of a value or object.
keyof gets the union of all keys (property names) of that type.

Eg: const myObject = { name: "John", age: 30 };
// keyof typeof myObject will give: "name" | "age"

Question 14 (MCQ)

Which of the following is a valid example of tuple types in TypeScript?

A. let tuple: Array<number, string> = [1, 'hello'];
B. let tuple: [number, string] = [1, 'hello'];
C. let tuple: (number, string) = (1, 'hello');
D. let tuple: [number | string] = [1, 'hello'];

Answer:

B. let tuple: [number, string] = [1, 'hello'];

Why?: Tuples define fixed-length arrays with specific types for each element. In this case, the tuple consists of a number and a string.

Question 15

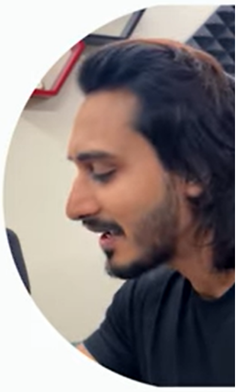
What are discriminated unions in TypeScript?

Discriminated unions use a common property (discriminator) across multiple types to distinguish between different types within a union, allowing for more precise type narrowing.

```
// Define the discriminated union
type Vehicle =
| { type: 'car'; wheels: number }
| { type: 'bicycle'; pedals: number };

// Function to describe the vehicle
function describeVehicle(vehicle: Vehicle) {
// TypeScript will narrow the type based on the `type` property
  if (vehicle.type === 'car') {
    console.log(`This vehicle has ${vehicle.wheels} wheels.`);
  } else if (vehicle.type === 'bicycle') {
    console.log(`This vehicle has ${vehicle.pedals} pedals.`);
  }
}

// Example usage
const myCar: Vehicle = { type: 'car', wheels: 4 };
const myBike: Vehicle = { type: 'bicycle', pedals: 2 };
describeVehicle(myCar); // Output: This vehicle has 4 wheels.
describeVehicle(myBike); // Output: This vehicle has 2 pedals.
```



Question 16 (MCQ)

Which of the following would allow narrowing a type using the `in` operator in TypeScript?

*Hint: The **in operator** checks whether a specific property exists in an object or one of its prototypes. Syntax -* 'propertyName' in object

A. if (object.hasOwnProperty('property')) { ... }
B. if ('property' in object) { ... }
C. if (object.property !== undefined) { ... }
D. if (typeof object.property !== 'undefined') { ... }

Answer:
B. if ('property' in object) { ... }

Why?: The `in` operator checks if a property exists in an object and type accordingly.

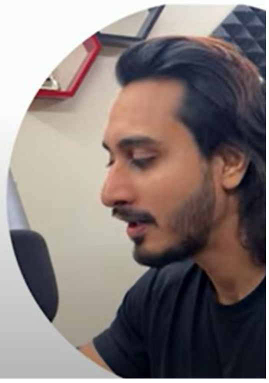
Question 16 (MCQ)

if ('property' in object) is the correct way to narrow a type using the in operator because it checks whether 'property' exists in object. If the check passes, TypeScript narrows the type of object inside the if block, allowing access to that property with confidence.

Eg.

```
type Dog = { bark: () => void };
type Cat = { meow: () => void };

function animalSound(animal: Dog | Cat) {
    if ('bark' in animal) {
        animal.bark(); // TypeScript knows animal is a Dog here
    } else {
        animal.meow(); // TypeScript knows animal is a Cat here
    }
}
```



The in operator checks whether bark exists in animal.If true,
TypeScript narrows the type to Dog, allowing safe access to bark().
If false, TypeScript narrows the type to Cat, allowing access to meow().

Question 17 (MCQ)

Which of the following is an example of using the `infer` keyword in TypeScript?

A. type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;
B. type ReturnType<T> = infer T extends (...args: any[]) => R ? R : never;
C. type ReturnType<T> = T extends infer (...args: any[]) => R ? R : never;
D. type ReturnType<T> = T matches (...args: any[]) => infer R ? R : never;

Answer:

A. type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;

Question 17 (MCQ)

Which of the following is an example of using the `infer` keyword in TypeScript?

The correct form of using infer in a conditional type is: T extends U ? infer R : never

T is a type.
U is a type pattern.
If T matches U, we **infer** some type R from U.
If the condition is met, the type R can be used in the "true" branch of the conditional type.

Question 17 (MCQ)

Which of the following is an example of using the `infer` keyword in TypeScript?

<u>Analyzing the Options:</u>

1. type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;
This is the **correct answer** because it uses the infer keyword correctly to infer the return type R from a function type.

This checks if T is a function type. If it is, TypeScript **infers** R as the return type of the function, and returns R. Otherwise, it returns never.

Eg:

```
type MyFunc = (x: number) => string;
type Result = ReturnType<MyFunc>; // Result is string
```

In this example, T is the function (x: number) => string. The infer R part captures the return type R, which is string.

// Infer Keyword with condition another Example functions

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

function greet(name: string): string {
  return `Hello, ${name}`;
}

// Using the ReturnType utility type to infer the return type of these functions
type AddReturnType = ReturnType<typeof add>;   // number
type GreetReturnType = ReturnType<typeof greet>; // string

// Checking the inferred types
const numberResult: AddReturnType = 42;
// This works, because the inferred return type is 'number'

const stringResult: GreetReturnType = "Hello, John";
// This works, because the inferred return type is 'string'
```

Done