00:07 - What Is TypeScript?

TypeScript Overview: TypeScript is a statically typed superset of JavaScript that compiles down to plain JavaScript. It adds static type definitions, which makes it easier to catch errors early during development. TypeScript improves code readability and maintainability by enforcing type annotations and providing enhanced tooling support.

Why Use TypeScript?

1. Type Safety: Ensures that variables and functions are used in a type-safe manner.
2. Tooling: Provides autocompletion, type inference, and refactoring tools.
3. ESNext Features: Supports modern JavaScript features like async/await, destructuring, etc.

Example:

typescript

CopyEdit

```
let age: number = 30;  // TypeScript checks that 'age' is a number
```

02:34 - Course Setup

In this section, we set up TypeScript for development by installing the required tools, initializing a project, and configuring the TypeScript compiler.

1. **Install Node.js and TypeScript:**

bash

CopyEdit

```
npm install -g typescript
```

2. **Create a TypeScript Project:**

bash

CopyEdit

```
mkdir typescript-project

cd typescript-project

tsc --init
```

3. **Create a tsconfig.json File:** This file tells TypeScript how to compile your code.

11:40 - Annotations In Depth

Type Annotations: Type annotations are a way to explicitly declare the type of a variable, function parameter, or return value in TypeScript.

Example:

typescript

CopyEdit

```
let name: string = "John";  // Annotation ensures that 'name' is a string

let isActive: boolean = true;  // Annotation ensures that 'isActive' is a boolean
```

In addition to basic types (number, boolean, string), TypeScript supports more complex types like arrays, objects, and even custom types.

18:35 - Type Inference

Type Inference: TypeScript is smart enough to infer the type of a variable based on its initial value. You don’t always have to specify the type explicitly if it can be inferred.

Example:

typescript

CopyEdit

```
let message = "Hello, World!";  // TypeScript infers 'message' to be a string
```

Here, TypeScript infers the type of message as string because the value "Hello, World!" is a string.

22:53 - Any Type

The any Type: The any type allows you to opt out of TypeScript’s static typing system. When you use any, TypeScript no longer performs type checking on that variable.

Example:

typescript

CopyEdit

```
let anything: any = "Hello";

anything = 42;  // No error, can assign any value

anything = true;  // No error
```

Warning: Use any sparingly, as it bypasses TypeScript’s safety checks.

26:11 - Function Parameter Annotations

You can provide type annotations for function parameters to enforce the types of arguments passed to the function.

Example:

typescript

CopyEdit

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

console.log(greet("John"));  // Correct usage
// console.log(greet(42));  // Error: Argument of type '42' is not assignable to parameter of type 'string'
```

Sub-topic: Parameter Type Declaration If a parameter is optional, you can use the ? symbol to mark it as optional.

Example:

typescript

CopyEdit

```
function greet(name: string, age?: number): string {
  return `Hello, ${name}. You are ${age ? age : "unknown"}`;
}

console.log(greet("John"));
console.log(greet("Alice", 25));
```

31:22 - Default Params Values

Default Parameter Values: You can assign default values to parameters, which are used when the argument is not passed.

Example:

typescript

CopyEdit

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

console.log(greet());      // "Hello, Guest"
console.log(greet("John")); // "Hello, John"
```

33:27 - Function Return Annotations

Function Return Type Annotations: You can specify the return type of a function. This is helpful for ensuring the function always returns the expected type.

Example:

typescript

CopyEdit

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

console.log(sum(2, 3));  // 5
```

If you return a type that doesn’t match the return type annotation, TypeScript will give an error.

36:19 - Void Functions

Void Return Type: A function that returns void means that it does not return anything. This is used for functions that perform side effects (like logging or modifying data) but don’t need to return any value.

Example:

typescript

CopyEdit

```
function logMessage(message: string): void {
  console.log(message);
}

logMessage("Hello, TypeScript!");  // Outputs: Hello, TypeScript!
```

38:05 - Never Keyword In Depth

The never Type: The never type represents values that never occur, typically used in functions that throw an error or have infinite loops.

Example:

typescript

CopyEdit

```
function throwError(message: string): never {
  throw new Error(message);
}

throwError("An error occurred!");  // Throws an error and never returns
```

Sub-topic: Infinite Loop: You can also use never in functions with infinite loops:

typescript

CopyEdit

```
function runForever(): never {
  while (true) {}
}
```

42:19 - Arrays Types In Depth

Array Types: You can specify the type of elements in an array using either type[] or Array<type>.

Example:

typescript

CopyEdit

```
let numbers: number[] = [1, 2, 3];
let names: Array<string> = ["Alice", "Bob", "Charlie"];
```

49:36 - Multi Dimensional Arrays

Multi-dimensional Arrays: A multi-dimensional array is an array of arrays. It is often used for tables or matrices.

Example:

typescript

CopyEdit

```
let matrix: number[][] = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

console.log(matrix[0][0]);  // 1
```

52:18 - Objects In Depth

Objects and Types: You can define the shape of an object by specifying types for its properties.

Example:

typescript

CopyEdit

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

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

58:58 - Type Aliases

Type Aliases: Type aliases allow you to give a name to a specific type, which can make complex types easier to read and reuse.

Example:

typescript

CopyEdit

```
type Car = {
  make: string;
  model: string;
  year: number;
};
```

```
let myCar: Car = {
  make: "Toyota",
  model: "Corolla",
  year: 2021
};
```

1:03:30 - Optional Properties

Optional Properties: You can make properties optional in an interface or type alias using the ? symbol.

Example:

typescript

CopyEdit

```
type Person = {
  name: string;
  age: number;
  email?: string;  // Optional property
};

let user1: Person = { name: "Alice", age: 25 };
let user2: Person = { name: "Bob", age: 30, email: "bob@example.com" };
```

1:07:13 - Readonly Property

Readonly Properties: The readonly modifier makes properties of an object immutable after initialization.

Example:

typescript

CopyEdit

```
type Person = {
  readonly id: number;
  name: string;
};
```

```
let person: Person = { id: 1, name: "John" };
// person.id = 2;  // Error: Cannot assign to 'id' because it is a read-only property
```

1:08:27 - Intersection Types

Intersection Types: An intersection type combines multiple types into one. An object of this type must satisfy all the types it intersects.

Example:

typescript

CopyEdit

```
type Person = { name: string };
type Employee = { id: number };

type EmployeeDetails = Person & Employee;

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

1:13:30 - Unions

Union Types: A union type allows a variable to hold one of several types. This is useful when a value can have multiple possible types.

Example:

typescript

CopyEdit

```
let value: string | number;
value = "Hello";  // Valid
value = 42;      // Valid
value = true;     // Error: Type 'boolean' is not assignable to type 'string | number'
```

1:18:35 - Literal Types

Literal Types: Literal types allow you to specify exact values a variable can hold, making it possible to restrict the possible values to specific ones.

Example:

typescript

CopyEdit

```
let direction: "left" | "right" = "left";  // Only "left" or "right" allowed
direction = "right";  // Valid
direction = "up";     // Error: Type '"up"' is not assignable to type '"left" | "right"'
```

1:22:44 - Tuples

Tuples: A tuple is an array with a fixed number of elements where each element can have a different type.

Example:

typescript

CopyEdit

```
let user: [string, number] = ["John", 30];
console.log(user[0]);  // John
console.log(user[1]);  // 30
```

1:28:31 - Enums

Enums: Enums allow you to define a set of named constants. You can access enum members by name or by value.

Example:

typescript

CopyEdit

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT"
}

let dir: Direction = Direction.Up;
console.log(dir);  // "UP"
```

1:33:11 - OOP (Object-Oriented Programming)

TypeScript supports object-oriented programming principles like classes, interfaces, and inheritance.

Example:

typescript

CopyEdit

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

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

let person = new Person("Alice");
person.greet();  // Hello, Alice
```

1:49:34 - Interfaces

Interfaces: Interfaces define the structure of an object. They can be used to enforce that a class implements a particular contract.

Example:

typescript

CopyEdit

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

let employee: Person = { name: "John", age: 30 };
```

2:13:44 - Generics

Generics: Generics allow you to create reusable functions, classes, and interfaces that can work with any type while maintaining type safety.

Example:

typescript

CopyEdit

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

let numberResult = identity(5);    // number
let stringResult = identity("Hi");  // string
```

2:47:00 - Type Narrowing

Type Narrowing: Type narrowing is the process of refining the type of a variable by using type guards or conditional checks.

Example:

typescript

CopyEdit

```
function isString(value: string | number): value is string {
  return typeof value === "string";
}

let result: string | number = "Hello";
if (isString(result)) {
  console.log(result.toUpperCase());  // No error, 'result' is narrowed to string
}
```

This breakdown provides a detailed understanding of each concept with examples and explanation of subtopics where applicable. TypeScript allows you to write more reliable code with better tooling support, especially when working on large applications.

You said:

narrow typing in depth

ChatGPT said:

Narrowing Types in TypeScript (In-depth Explanation)

Type Narrowing is the process of refining or reducing a type to a more specific subtype based on certain conditions. In TypeScript, narrowing helps you handle union types (e.g., string | number) by narrowing them down to a specific type (string or number) based on runtime checks. This allows you to work with more specific types after certain checks, enabling TypeScript to give you accurate type checking.

Why is Type Narrowing Important?

1. **Refining Union Types:** Type narrowing is particularly useful when you have union types, which combine multiple types into one. It helps ensure that once you've checked the type, you can safely use properties or methods specific to that type.
2. **Enabling More Specific Operations:** By narrowing a type, you can access properties or methods that are only available on one of the types in the union.

Basic Type Narrowing Examples

Let’s look at a basic scenario of narrowing a union type from string | number to a more specific type using **typeof** operator.

Example 1: Using typeof for Type Narrowing

typescript

CopyEdit

```
function printLength(value: string | number): void {
 if (typeof value === "string") {
  console.log(value.length);  // Safe, because value is now known to be a string
 } else {
  console.log(value.toFixed(2));  // Safe, because value is now known to be a number
 }
}

printLength("Hello, World!");  // 13
printLength(42);  // 42.00
```

In this example, typeof value === "string" narrows the type of value to string in that block of code. Similarly, the else block narrows value to number. The compiler knows that you’re either dealing with a string or a number, so you can safely access their respective properties or methods.

Narrowing with instanceof

The **instanceof** operator can be used to narrow down types for objects that are instances of classes or constructor functions. It works with object-oriented types like Date, Array, and custom classes.

Example 2: Using instanceof for Type Narrowing

typescript

CopyEdit

```
function printDateInfo(date: Date | string): void {
  if (date instanceof Date) {
    console.log(date.toDateString());  // Works only if 'date' is an instance of Date
  } else {
    console.log(new Date(date).toDateString());  // Converts string to Date object
  }
}

printDateInfo(new Date());  // Outputs today's date
printDateInfo("2025-01-01");  // Outputs "Thu Jan 01 2025"
```

Here, date instanceof Date narrows date to Date type in the first block, and else handles the case when date is a string.

Narrowing with Type Guards (Custom Functions)

Type guards are user-defined functions that return a **type predicate** to inform TypeScript of a variable's type after a runtime check.

Example 3: Using Type Guards

typescript

CopyEdit

```
function isString(value: string | number): value is string {
  return typeof value === "string";  // Return true if value is a string
}
```

```
function printLength(value: string | number): void {
  if (isString(value)) {
    console.log(value.length);  // Safe, as TypeScript now knows 'value' is a string
  } else {
    console.log(value.toFixed(2));  // Safe, as 'value' is narrowed to number
  }
}

printLength("Hello");
printLength(123);
```

In this example, the isString function is a custom **type guard**. It checks if value is a string, and TypeScript knows that within the block where isString(value) is true, value is a string.

What is a Type Predicate?

The value is string part in the isString function is a **type predicate**. It tells TypeScript that if isString returns true, then value must be of type string.

Narrowing with in Operator (For Objects)

The **in** operator allows you to check if a property exists in an object. This can be useful when narrowing types involving object shapes.

Example 4: Using in for Narrowing Object Types

typescript

CopyEdit

```
interface Car {
  type: "car";
  speed: number;
}

interface Bike {
  type: "bike";
  gearCount: number;
}
```

```
type Vehicle = Car | Bike;


function describeVehicle(vehicle: Vehicle): void {
  if ("speed" in vehicle) {
    console.log(`Car speed: ${vehicle.speed} km/h`);
  } else {
    console.log(`Bike gears: ${vehicle.gearCount}`);
  }
}


const myCar: Car = { type: "car", speed: 120 };
const myBike: Bike = { type: "bike", gearCount: 18 };


describeVehicle(myCar);  // "Car speed: 120 km/h"
describeVehicle(myBike); // "Bike gears: 18"
```

In this case, if ("speed" in vehicle) narrows the type of vehicle to Car (because only Car has the speed property), and the else block narrows it to Bike.

Narrowing with null and undefined

TypeScript has built-in support for checking null and undefined. You can use null and undefined checks to narrow types as well.

Example 5: Using null and undefined Checks for Narrowing

typescript

CopyEdit

```
function printLength(value: string | null): void {
  if (value === null) {
    console.log("No value provided");
  } else {
    console.log(value.length);  // Safe, as 'value' is narrowed to string
  }
}
```

```
printLength("Hello");  // Outputs 5

printLength(null);      // Outputs "No value provided"
```

In this case, value === null narrows the type to null, and in the else block, value is treated as a string.

Using as for Type Assertions

Type assertions allow you to tell TypeScript that you know the type of a variable better than it does. You can use the as keyword to assert a type, though it's generally better to use type guards for safer type narrowing.

Example 6: Using as for Type Narrowing

typescript

CopyEdit

```
function getLength(value: string | number): number {

  if (typeof value === "string") {

    return value.length;  // Narrowed to string

  }

  return (value as string).length;  // Type assertion, assumes 'value' is string

}


console.log(getLength("Hello")); // Outputs 5
```

Though as can be used, relying on proper type narrowing mechanisms (such as typeof, instanceof, or type guards) is safer and avoids potential runtime errors.

Combining Multiple Narrowing Techniques

You can combine narrowing techniques to handle more complex cases.

Example 7: Combining Narrowing Techniques

typescript

CopyEdit

```
type Animal = { type: "dog", bark: () => void } | { type: "cat", meow: () => void };


function makeSound(animal: Animal): void {

  if (animal.type === "dog") {
    animal.bark();  // Narrowed to 'dog' type, so 'bark' is safe
  } else {
    animal.meow();  // Narrowed to 'cat' type, so 'meow' is safe
  }
}

const dog: Animal = { type: "dog", bark: () => console.log("Woof!") };
const cat: Animal = { type: "cat", meow: () => console.log("Meow!") };

makeSound(dog);  // Outputs "Woof!"
makeSound(cat);  // Outputs "Meow!"
```

In this example, the type of animal is narrowed based on the type property, allowing access to the methods specific to each type (bark for dogs, meow for cats).

Summary of Type Narrowing Techniques:

1. **typeof operator** - Used for primitive types like string, number, boolean, etc.
2. **instanceof operator** - Used to narrow types for objects or class instances.
3. **Custom Type Guards** - Functions that return a **type predicate** to narrow types.
4. **in operator** - Used to check for the existence of properties in an object.
5. **null and undefined checks** - Used to narrow types based on the presence or absence of a value.
6. **as keyword (Type Assertion)** - Used to forcefully assert a type, but should be used with caution.

Type narrowing is a crucial feature in TypeScript that allows developers to write more accurate and safer code by reducing the ambiguity of union types and enabling more specific type checks.