Objectives

Utility Types

Type Assertions and Type Guards

Use Cases of Type Assertion

Type Guards

Custom Type Guards

Working with any, unknown, and never

Modules and Namespaces

Error Handling and Debugging

Utility Types

1. Partial

Creates a type where all properties are optional.

Example:

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

function updateUser(user: User, updates: Partial<User>): User {
    return { ...user, ...updates };
}

const user: User = { id: 1, name: "John", email: "john@example.com" };
```

```
const updatedUser = updateUser(user, { name: "Johnny" });
console.log(updatedUser); // Output: { id: 1, name: "Johnny", email:
"john@example.com" }
```

When Should You Use `Partial`?

- Use `Partial` when you're working with objects where updates involve only a subset of the properties.
- It's particularly helpful in scenarios like:
    - Updating a record in a database.
    - Working with forms where only some fields might be modified.
    - Passing configuration objects with optional overrides.

2. Readonly

Creates a type where all properties are immutable.

Example:

```
const user: Readonly<User> = { id: 1, name: "John", email:
"john@example.com" };
// user.name = "Johnny"; // Error: Cannot assign to 'name' because it is a
read-only property
console.log(user);
```

3. Pick

Creates a type by picking a subset of properties from an existing type.

Example:

```
type UserPreview = Pick<User, "name" | "email">;

const userPreview: UserPreview = { name: "John", email: "john@example.com"
};
console.log(userPreview); // Output: { name: 'John', email:
'john@example.com' }
```

4. Omit

Creates a type by omitting specified properties from an existing type.

Example:

```
type UserWithoutEmail = Omit<User, "email">;

const userWithoutEmail: UserWithoutEmail = { id: 1, name: "John" };
console.log(userWithoutEmail); // Output: { id: 1, name: 'John' }
```

Type Assertions and Type Guards

Type Assertion in TypeScript

Type Assertion allows you to override TypeScript's inferred type or explicitly tell the compiler about the type of a value. It provides a way to inform TypeScript about the type of a variable when TypeScript cannot automatically infer it or when you know the type better than the compiler.

Syntax of Type Assertion

There are two ways to perform type assertions:

1. **Using the `as` keyword** (preferred syntax):

   ```
   let value: unknown = "Hello, TypeScript!";
   let strLength: number = (value as string).length;
   ```

2. **Using angle brackets `< >`** (not allowed in `.tsx` files):

   ```
   let value: unknown = "Hello, TypeScript!";
   let strLength: number = (<string>value).length;
   ```

When to Use Type Assertion

1. **When TypeScript cannot infer the type:** For example, when working with the `unknown` or `any` type, you may need to explicitly assert the type to access specific properties or methods.

   ```
   let someValue: any = "This is a string";
   let stringLength: number = (someValue as string).length;
   console.log(stringLength); // Output: 16
   ```

2. **When accessing DOM elements:** TypeScript cannot infer the specific type of a DOM element, so you might need to use type assertion.

   ```
   let inputElement = document.getElementById("myInput") as
   HTMLInputElement;
   inputElement.value = "New value";
   ```

3. **When dealing with third-party libraries:** Some libraries may not have complete TypeScript definitions, requiring you to assert types explicitly.

Use Cases of Type Assertion

1. Overriding Type Inference

TypeScript may infer a less specific type than required. You can use type assertions to tell the compiler the exact type.

```
let value: unknown = "TypeScript";
console.log((value as string).toUpperCase());
```

2. Handling `unknown` Type

The `unknown` type is safer than `any` but requires type assertions to narrow the type for usage.

```
function processValue(value: unknown) {
    if (typeof value === "string") {
        console.log((value as string).toUpperCase());
    }
}
processValue("hello");
```

3. Narrowing Union Types

When working with union types, TypeScript might not know the specific type of a value unless you assert it.

```
function getLength(input: string | string[]): number {
    if ((input as string[]).length !== undefined) {
        return (input as string[]).length; // Treat it as an array
    }
    return (input as string).length; // Treat it as a string
}
```

4. Type Assertion in Arrays

Useful when TypeScript infers the type of an array too broadly.

```
let mixedArray: any[] = [1, "two", 3];
let stringArray = mixedArray as string[]; // Treat all elements as strings
```

5. Working with Custom Types or Interfaces

When the type of a value aligns with a custom interface or type, you can assert it.

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

let userData: any = { id: 1, name: "John" };
let user = userData as User;
console.log(user.name); // Output: John
```

Type Assertion vs Type Casting

- **Type Assertion:** Only affects TypeScript during compilation. No changes are made to the actual object or runtime behavior.
- **Type Casting (in other languages):** May involve actual runtime changes (e.g., converting a string to a number).

Example:

```
let num: number = "123" as unknown as number; // Type assertion
```

Type Assertion and Runtime Errors

Type assertion doesn't perform any actual type-checking at runtime. If the type assertion is incorrect, it can lead to runtime errors.

Example:

```
let someValue: any = 42;
console.log((someValue as string).toUpperCase()); // Runtime Error:
someValue.toUpperCase is not a function
```

Custom Example

Scenario: Parsing JSON with Type Assertion

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

let jsonString = '{"id":1,"name":"John Doe","email":"john@example.com"}';
let user = JSON.parse(jsonString) as User;

console.log(user.name); // Output: John Doe
```

Caution: If the JSON string doesn't match the `User` interface, the code may compile but result in runtime errors.

When Not to Use Type Assertion

1. **Avoid Assertions When TypeScript Already Knows the Type:**

   ```
   let value: string = "hello";
   let length = (value as string).length; // Unnecessary assertion
   ```

2. **Be Cautious with `any`:** Overusing type assertions with `any` can negate TypeScript's type-safety advantages.

2. Type Guards

Type Guards in TypeScript

Type Guards are techniques or constructs that allow TypeScript to narrow down the type of a variable within a conditional block. They help ensure type safety by making sure that the code operates on values of the expected type.

Why Use Type Guards?

Type Guards are necessary because:

1. **Union Types:** When a variable can have multiple types, TypeScript cannot automatically determine the specific type in a particular block of code.
2. **Runtime Type Checking:** Type Guards allow you to verify the type of a variable dynamically during execution and inform TypeScript of the narrowed type.
3. **Error Prevention:** They reduce the risk of runtime errors by ensuring operations are performed on appropriate types.

Types of Type Guards

1. `typeof` Type Guard

The `typeof` operator is used to check the type of primitive values such as `string`, `number`, `boolean`, `bigint`, `symbol`, `undefined`, or `object`.

Example:

```
function printValue(value: string | number) {
    if (typeof value === "string") {
        console.log("String value:", value.toUpperCase());
    } else if (typeof value === "number") {
        console.log("Number value:", value.toFixed(2));
    }
}

printValue("TypeScript"); // Output: String value: TYPESCRIPT
printValue(123.45);       // Output: Number value: 123.45
```

---

2. `instanceof` Type Guard

The `instanceof` operator checks if an object is an instance of a specific class.

Example:

```
class Dog {
    bark() {
        console.log("Woof!");
    }
}

class Cat {
    meow() {
        console.log("Meow!");
    }
}

function handleAnimal(animal: Dog | Cat) {
    if (animal instanceof Dog) {
        animal.bark();
    } else if (animal instanceof Cat) {
        animal.meow();
    }
}

handleAnimal(new Dog()); // Output: Woof!
handleAnimal(new Cat()); // Output: Meow!
```

---

3. Custom Type Guards

Custom type guards are functions that return a boolean and use a type predicate to narrow the type.

Syntax of Type Predicate:

```
param is Type
```

Example:

```
interface Fish {
    swim(): void;
}

interface Bird {
    fly(): void;
}

function isFish(animal: Fish | Bird): animal is Fish {
    return (animal as Fish).swim !== undefined;
}

function handleAnimal(animal: Fish | Bird) {
    if (isFish(animal)) {
        animal.swim();
    } else {
        animal.fly();
    }
```

```
}

const fish: Fish = { swim: () => console.log("Swimming") };
const bird: Bird = { fly: () => console.log("Flying") };

handleAnimal(fish); // Output: Swimming
handleAnimal(bird); // Output: Flying
```

---

4. `in` Operator Type Guard

The `in` operator checks if a property exists in an object.

Example:

```
interface Car {
    drive(): void;
}

interface Boat {
    sail(): void;
}

function operateVehicle(vehicle: Car | Boat) {
    if ("drive" in vehicle) {
        vehicle.drive();
    } else if ("sail" in vehicle) {
        vehicle.sail();
    }
}

const car: Car = { drive: () => console.log("Driving a car") };
const boat: Boat = { sail: () => console.log("Sailing a boat") };

operateVehicle(car);  // Output: Driving a car
operateVehicle(boat); // Output: Sailing a boat
```

Working with `any`, `unknown`, and `never` in TypeScript

These three special types in TypeScript serve different purposes and provide flexibility while working with strict type systems. Let's explore them in detail.

---

1. `any` Type

- **Definition:** The `any` type disables all type checking for a variable. It allows the variable to hold values of any type, effectively opting out of TypeScript's static type system.
- **Key Points:**
    - No type checking is performed.
    - Allows any operation to be performed on the variable without errors.
    - Should be used sparingly, as it defeats the purpose of TypeScript.

- **Example:**

```
let value: any = 42;
value = "Hello";  // No error
value = true;     // No error

function log(value: any) {
    console.log(value.toUpperCase()); // No error, but runtime error
if `value` is not a string
}

log(123); // Runtime error: value.toUpperCase is not a function
```

- **Use Case:** Use `any` when migrating JavaScript code to TypeScript or working with data from dynamic sources (e.g., JSON).

2. `unknown` Type

- **Definition:** The `unknown` type is a safer alternative to `any`. It represents a value with an unknown type, but unlike `any`, it requires you to explicitly check or assert the type before performing operations.
- **Key Points:**
  - Type-safe compared to `any`.
  - You cannot directly perform operations on `unknown` values without narrowing the type.
  - Forces you to validate or assert the type before use.
- **Example:**

```
let value: unknown = "Hello, TypeScript!";

// Compilation error: Object is of type 'unknown'
// console.log(value.toUpperCase());

// Correct way: Narrow the type first
if (typeof value === "string") {
    console.log(value.toUpperCase()); // Output: HELLO, TYPESCRIPT!
}
```

- **Use Case:** Use `unknown` when dealing with values whose type isn't known at compile time (e.g., data from APIs, external libraries, or user input).

Comparison: `any` vs `unknown`

Feature	`any`	`unknown`
Type Checking	Disabled	Enforced
Safety	Unsafe	Safe
Usage	No restrictions	Requires type checking or assertion
Errors on Operations	No compile-time errors	Compile-time errors if type not checked

3. `never` Type

- **Definition:** The `never` type represents values that never occur. It is used in scenarios where a function never successfully returns, such as when throwing errors or infinite loops.
- **Key Points:**
  - Used for functions that always throw errors or never terminate.
  - Indicates unreachable code.
  - Can also appear in exhaustive checks for union types.
- **Example 1: Function That Throws an Error**

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

throwError("Something went wrong!"); // The function never returns
```

- **Example 2: Function with Infinite Loop**

```
function infiniteLoop(): never {
    while (true) {
        console.log("Running forever...");
    }
}
```

- **Use Case:**
  1. Functions that do not return a value (like those throwing exceptions or infinite loops).
  2. Exhaustive type checks.

Exhaustive Checks Using `never`

When dealing with discriminated unions, the `never` type is useful to ensure all cases are handled.

Example:

```
type Shape = { kind: "circle"; radius: number } | { kind: "square"; side: number };

function calculateArea(shape: Shape): number {
    switch (shape.kind) {
        case "circle":
            return Math.PI * shape.radius ** 2;
        case "square":
            return shape.side ** 2;
        default:
            const _exhaustiveCheck: never = shape; // Ensures no new cases are missed
            throw new Error(`Unhandled shape: ${_exhaustiveCheck}`);
```

```
    }
}
```

If a new kind (e.g., `triangle`) is added to the `Shape` type and not handled in the `switch` statement, TypeScript will throw a compile-time error for the `_exhaustiveCheck`.

Real-World Scenarios

`any` Use Case:

Dynamic data processing where type is unknown at the time of writing code.

```
function parseJSON(json: string): any {
    return JSON.parse(json); // Returns any type
}
```

`unknown` Use Case:

Handling API responses safely.

```
function processApiResponse(response: unknown) {
    if (typeof response === "object" && response !== null) {
        console.log("Response is an object:", response);
    } else {
        console.log("Invalid response format");
    }
}
```

`never` Use Case:

Handling impossible states or exhaustive checks.

```
function handleValue(value: string | number): string {
    if (typeof value === "string") {
        return `String value: ${value}`;
    } else if (typeof value === "number") {
        return `Number value: ${value}`;
    }
    // Compile-time error if new cases are added to the union
    const _exhaustive: never = value;
    throw new Error(`Unhandled value: ${_exhaustive}`);
}
```

Modules and Namespaces in TypeScript

Modules and namespaces are two mechanisms in TypeScript to organize and structure code.

1. Modules

Modules are a way to split code into separate files and reuse them across different parts of your application. TypeScript supports **ES6 modules** where each file can be a module and exports or imports parts of the code.

Key Concepts:

- **Export**: Makes variables, functions, or classes available outside the file.
- **Import**: Brings in functionality from another module.

Example: Using Modules

File: `mathUtils.ts`

```
// Exporting functions
export function add(a: number, b: number): number {
  return a + b;
}

export function subtract(a: number, b: number): number {
  return a - b;
}

// Exporting a constant
export const PI = 3.14;
```

File: `app.ts`

```
// Importing specific exports
import { add, subtract, PI } from './mathUtils';

console.log(`Add: ${add(5, 3)}`);          // Output: Add: 8
console.log(`Subtract: ${subtract(5, 3)}`); // Output: Subtract: 2
console.log(`PI: ${PI}`);                  // Output: PI: 3.14
```

Dynamic Import You can also dynamically import modules:

```
const mathUtils = await import('./mathUtils');
console.log(mathUtils.add(2, 3)); // Output: 5
```

---

2. Namespaces

Namespaces are an older mechanism for organizing code in TypeScript. They are used to group related code together under a single name. Unlike modules, namespaces do not create separate files.

Key Concepts:

- Use the `namespace` keyword.
- Use the `export` keyword to make components available outside the namespace.

Example: Using Namespaces

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

  export function subtract(a: number, b: number): number {
    return a - b;
  }

  export const PI = 3.14;
}

// Accessing members of the namespace
console.log(`Add: ${MathOperations.add(10, 5)}`);          // Output: Add:
15
console.log(`Subtract: ${MathOperations.subtract(10, 5)}`); // Output:
Subtract: 5
console.log(`PI: ${MathOperations.PI}`);                   // Output: PI:
3.14
```

Modules vs Namespaces

Feature	Modules	Namespaces
Use case	Split code across files	Group code in the same file
Usage	Preferred for modern projects	Legacy projects
Export mechanism	`export` and `import`	`namespace` and `export`
Scope	File-level scope	Encapsulated within the namespace

Complete Example: Modules and Namespaces Together

File: *`geometry.ts`*

```
export namespace Geometry {
  export function areaOfCircle(radius: number): number {
    return Math.PI * radius * radius;
  }

  export function perimeterOfCircle(radius: number): number {
    return 2 * Math.PI * radius;
  }
}
```

File: *`main.ts`*

```
import { Geometry } from './geometry';

console.log(`Area of Circle: ${Geometry.areaOfCircle(5)}`);          //
Output: Area of Circle: 78.53981633974483
console.log(`Perimeter of Circle: ${Geometry.perimeterOfCircle(5)}`); //
Output: Perimeter of Circle: 31.41592653589793
```

This provides the flexibility of modules while still grouping related functionalities within namespaces.

Error Handling and Debugging

1. Exceptions

```
try {
    throw new Error("Something went wrong!");
} catch (error) {
    console.log((error as Error).message); // Output: Something went wrong!
} finally {
    console.log("Cleaning up resources...");
}
```

---

2. Strict Null Checks

```
function getLength(value: string | null): number {
    if (value === null) {
        return 0;
    }
    return value.length;
}

console.log(getLength(null)); // Output: 0
console.log(getLength("hello")); // Output: 5
```

Assignment: Employee Management and Analytics System

Objective:

Build a simple **Employee Management and Analytics System** to manage employees, track their activities, and generate reports using advanced TypeScript concepts.

1. Employee Management

1. **Define an Employee Model:**
    - Create an `Employee` structure with properties like ID, name, email, role, and status (active/inactive).
    - Ensure roles include predefined options like `Manager`, `Developer`, and `Intern`.
2. **Utility Types:**
    - Use TypeScript's **utility types**:
        - Allow updates to employee records with specific fields.
        - Generate employee summaries using only essential fields (e.g., name and role).
        - Store employees in a structured map format (e.g., group employees by role).
3. **Tasks:**

- Add a list of employees.
- Update specific fields for an existing employee.
- Create and display a summary of employees.

2. Activity Tracking

1. **Activity Logger:**
   - Implement a feature to log activities for employees.
   - Restrict certain activities to specific roles (e.g., only managers can approve reports).
2. **Type Guards:**
   - Write type guards to:
     - Validate an employee's role (e.g., check if an employee is a `Manager`).
     - Ensure activities can only be logged by active employees.
3. **Tasks:**
   - Log activities for different employees.
   - Attempt to log activities for inactive employees and observe the error handling.

3. Analytics Dashboard

1. **Analyze Data:**
   - Create a function to:
     - Count total employees.
     - Calculate active employees.
     - Display the distribution of roles.
2. **Handle Unknown Data:**
   - Simulate receiving external data.
   - Validate and process this data to ensure it matches the expected employee format.
   - Handle invalid data scenarios with meaningful error messages.
3. **Tasks:**
   - Pass valid and invalid employee lists to analyze the data.
   - Display analytics results and handle errors gracefully.

4. Error Handling and Debugging

1. **Error Handling:**
   - Simulate common errors like:
     - Updating a non-existent employee.
     - Logging activities for an invalid employee.
2. **Debugging:**
   - Use debugging tools like grouped console logs and tables to inspect data.
   - Create meaningful logs for each step.

3. **Tasks:**
    - Simulate error scenarios and document how they are handled.
    - Display employee data and analytics results in a structured format for debugging.