Day 2: Functions, Objects, and Interfaces

Objective: To learn how to apply types to functions and structure data with objects and interfaces.

Typing Functions (20 minutes)

In TypeScript, we can add type annotations to the parameters and the return value of a function. This makes our functions more predictable and easier to use.

1. Adding Types to Function Parameters and Return Values

You define the type for a parameter right after its name, separated by a colon. The return type is specified after the parameter list.

```
// A function that takes two numbers and returns a number
function add(x: number, y: number): number {
  return x + y;
}

// A function that takes a string and doesn't return anything (void)
function logMessage(message: string): void {
  console.log(message);
}
```

If you don't specify a return type, TypeScript will try to infer it. However, it's good practice to be explicit.

2. Optional and Default Parameters

- **Optional Parameters:** You can make a parameter optional by adding a `?` after its name. Optional parameters must come after required parameters.
- **Default Parameters:** You can provide a default value for a parameter, which will be used if no argument is provided. A parameter with a default value is automatically considered optional.

```
// 'lastName' is an optional parameter
function greet(firstName: string, lastName?: string): string {
  if (lastName) {
    return `Hello, ${firstName} ${lastName}!`;
  }
  return `Hello, ${firstName}!`;
}

// 'separator' has a default value of " "
function joinStrings(a: string, b: string, separator: string = " "): string {
  return a + separator + b;
}
```

3. Function Overloading

Function overloading allows you to define multiple function signatures for a single function body. This is useful when a function can be called with different types or numbers of arguments. You provide the overload signatures, and then one final implementation signature that is compatible with all the overloads.

```
// Overload signatures
function makeDate(timestamp: number): Date;
function makeDate(year: number, month: number, day: number): Date;

// Implementation signature
function makeDate(mOrY: number, month?: number, day?: number): Date {
  if (month !== undefined && day !== undefined) {
    return new Date(mOrY, month, day);
  } else {
    return new Date(mOrY);
  }
}

const date1 = makeDate(1663000000000);
const date2 = makeDate(2025, 8, 13);
```

4. Hands-on: Creating and Typing Various Functions

Open your `index.ts` file and add the following functions. Try calling them with different arguments to see how TypeScript helps you avoid errors.

```
// Function to calculate the area of a rectangle
function calculateArea(width: number, height: number): number {
  return width * height;
}

// Function to create a user profile string. The age is optional.
function createUserProfile(name: string, age?: number): string {
  if (age) {
    return `User: ${name}, Age: ${age}`;
  }
  return `User: ${name}`;
}

console.log(calculateArea(10, 5));
console.log(createUserProfile("Alice"));
console.log(createUserProfile("Bob", 32));
```

Working with Objects (15 minutes)

TypeScript allows you to define the "shape" of an object, specifying what properties it should have and what types those properties should be.

1. Defining Object Types

You can annotate an object's type directly. This ensures that any object assigned to that variable conforms to the specified structure.

```
let user: { name: string; id: number; isActive: boolean };

user = {
  name: "John Doe",
  id: 101,
  isActive: true,
};

// This would cause an error:
// user.id = "101"; // Type 'string' is not assignable to type 'number'.
```

2. Readonly Properties

You can mark a property as `readonly`. This means it can only be set when the object is first created and cannot be changed afterward.

```
let point: { readonly x: number; readonly y: number };

point = { x: 10, y: 20 };

// This would cause an error:
// point.x = 15; // Cannot assign to 'x' because it is a read-only property.
```

3. Activity: Creating Typed Objects

Let's practice by defining a typed object that represents a product. Add this to your `index.ts`:

```
let product: {
  readonly id: number;
  name: string;
  price: number;
  inStock?: boolean; // Optional property
};

product = {
  id: 12345,
  name: "Laptop",
  price: 999.99,
  inStock: true,
};

// Change the price
product.price = 899.99;

console.log(product);
```

Interfaces and Type Aliases (25 minutes)

Defining object shapes directly is fine for simple cases, but for more complex applications, you'll want to create reusable, named types. You can do this with `interface` and `type`.

1. Defining Custom Types with `type` and `interface`

- **Type Alias:** A `type` alias is a name for any type. It can be used for primitive types, unions, tuples, or objects.
- **Interface:** An `interface` is another way to name an object type.

```
// Using a Type Alias
type Person = {
  name: string;
  age: number;
};

// Using an Interface
interface Animal {
  species: string;
  sound: string;
}

let person: Person = { name: "Jane", age: 28 };
let dog: Animal = { species: "Canine", sound: "Woof" };
```

2. Extending Interfaces

You can combine interfaces. An interface can `extend` another, inheriting its members. This helps you build more complex types from simpler, reusable pieces.

```
interface Shape {
  color: string;
}

interface Square extends Shape {
  sideLength: number;
}

let mySquare: Square = { color: "blue", sideLength: 10 };
```

A `type` alias can achieve similar results using intersections (`&`).

3. Difference Between `type` and `interface`

For defining object shapes, they are very similar. The key differences are:

- **Extending:** The syntax for extending is different (`extends` for interfaces, `&` for types).
- **Declaration Merging:** An interface can be defined multiple times and the definitions will be merged. This is not possible with a `type` alias. This makes interfaces more "extendable" by third-party code.

General Recommendation: Use `interface` when defining the shape of an object. Use `type` for defining union types, tuples, or other more complex type compositions.

4. Hands-on: Building a Simple Application Structure with Interfaces

Let's model a simple blog post structure using interfaces. Add this to your `index.ts`:

```typescript
interface Author {
  id: number;
  name: string;
}

interface Post {
  readonly id: number;
  title: string;
  content: string;
  author: Author;
  tags?: string[];
}

const myPost: Post = {
  id: 1,
  title: "Learning TypeScript",
  content: "Today we are learning about functions, objects, and interfaces...",
  author: {
    id: 101,
    name: "Admin",
  },
  tags: ["typescript", "learning", "code"],
};

function displayPost(post: Post): void {
  console.log(`Title: ${post.title}`);
  console.log(`By: ${post.author.name}`);
  console.log(post.content);
}

displayPost(myPost);
```

Q&A and Wrap-up (5 minutes)

- **Recap of Functions, Objects, and Interfaces:**
  - We can add explicit types to function parameters and return values.
  - Functions can have optional and default parameters.
  - We can define the "shape" of objects to ensure their structure.
  - `interface` and `type` allow us to create reusable, named types for our objects.
- **Preview of Day 3:**
  - Tomorrow we'll explore object-oriented programming in TypeScript with **classes**, learn about a useful data type called **enums**, and see how to combine types using **union** and **intersection** types.