Blueprint for a 4-Day Intensive TypeScript Course

Course Title: TypeScript Fundamentals in 4 Days

Instructor: Joseph Opio

Course Description

This intensive four-day workshop provides a comprehensive introduction to TypeScript, a powerful typed superset of JavaScript. Over four one-hour lessons, participants will transition from foundational concepts to more advanced features, enabling them to write more robust, scalable, and maintainable code. The course is designed for developers with a working knowledge of JavaScript who want to leverage the benefits of a statically typed language. Each session will include theoretical explanations and practical examples to solidify understanding.

Prerequisites

- Solid understanding of JavaScript fundamentals (variables, functions, objects, arrays, etc.).
- A code editor (e.g., Visual Studio Code) installed.
- Node.js and npm (or yarn) installed to set up a TypeScript environment.

Course Objectives

By the end of this four-day course, participants will be able to:

- Understand the core concepts and benefits of TypeScript.
- Set up a TypeScript development environment.
- Utilize basic and complex types to create strongly typed variables and functions.
- Work with interfaces and classes to build structured and object-oriented code.
- Implement advanced TypeScript features like generics and utility types.

Daily Lesson Plan

Day 1: Introduction to TypeScript and Basic Types

Objective: To understand the fundamentals of TypeScript, set up the development environment, and learn how to use basic types.

Introduction to TypeScript (15 minutes)

- What is TypeScript and why use it?
- The relationship between TypeScript and JavaScript.
- Overview of the TypeScript compiler (tsc).
- **Activity:** Setting up a basic TypeScript project.

Basic Types and Type Annotations (30 minutes)

- Number, String, Boolean, Null, Undefined.
- The any and unknown types.
- Type Inference: Letting TypeScript figure out the types.
- Working with Arrays and Tuples.
- **Hands-on:** Declaring variables with different basic types and creating typed arrays.

Q&A and Wrap-up (15 minutes)

- Review of key concepts.
- Preview of Day 2.

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)

- Adding types to function parameters and return values.
- Optional and default parameters.
- Function overloading.
- **Hands-on:** Creating and typing various functions.

Working with Objects (15 minutes)

- Defining object types.
- Readonly properties.
- **Activity:** Creating typed objects.

Interfaces and Type Aliases (25 minutes)

- Defining custom types with `type` and `interface`.
- Extending interfaces.
- Difference between `type` and `interface`.
- **Hands-on:** Building a simple application structure with interfaces.

*Q&A and Wrap-up (5 minutes)***

- Recap of functions, objects, and interfaces.
- Preview of Day 3.

Day 3: Classes, Enums, and Union/Intersection Types

Objective: To explore object-oriented programming with classes, learn about enums, and combine types using union and intersection.

Classes in TypeScript (25 minutes)

- Defining classes and their properties.
- Constructors and methods.
- Access modifiers: `public`, `private`, and `protected`.
- Implementing interfaces in classes.
- **Hands-on:** Creating a class with properties, methods, and access modifiers.

Enums (10 minutes)

- What are enums and when to use them?
- Numeric and string enums.
- **Activity:** Defining and using an enum.

Union and Intersection Types (20 minutes)

- Allowing a variable to be one of several types (Union).
- Combining multiple types into one (Intersection).
- Type narrowing with `typeof` and `instanceof`.
- **Hands-on:** Writing a function that accepts a union type and performs different actions based on the type.

Q&A and Wrap-up (5 minutes)

- Review of classes, enums, and advanced types.
- Preview of Day 4.

Day 4: Generics, Utility Types, and Final Review

Objective: To understand how to write reusable, type-safe code with generics and leverage built-in utility types.

Generics (25 minutes)

- Introduction to generics for creating reusable components.
- Generic functions and interfaces.
- Generic constraints to limit the types that can be used.
- **Hands-on:** Creating a generic function that can work with different data types while maintaining type safety.

Utility Types (20 minutes)

- Overview of commonly used utility types: `Partial<T>`, `Readonly<T>`, `Pick<T>`, `Omit<T>`.
- Practical examples of how utility types can simplify your code.
- **Activity:** Using utility types to create new types from existing ones.

Course Review and Next Steps (15 minutes)

- Recap of all topics covered.
- Best practices and further learning resources.
- Final Q&A session.

Day 1: Introduction to TypeScript and Basic Types

Objective: To understand the fundamentals of TypeScript, set up the development environment, and learn how to use basic types.

Introduction to TypeScript (15 minutes)

1. What is TypeScript and Why Use It?

TypeScript is a programming language developed by Microsoft that builds on top of JavaScript. It's a "superset" of JavaScript, meaning any valid JavaScript code is also valid TypeScript code. The primary feature TypeScript adds to JavaScript is **static typing**.

Why is this important?

- **Error Detection:** In regular JavaScript, you might accidentally try to perform an operation on the wrong type of data, which you would only discover when you run the code (a runtime error). TypeScript checks for these types of errors as you code, catching them before you even run the program.
- **Improved Readability and Maintainability:** Explicitly defining the types of data your code expects makes it easier for you and other developers to understand what the code is doing, especially in large and complex applications.
- **Better Tooling:** TypeScript's static typing allows for more intelligent code completion, refactoring, and error checking in your code editor.

2. The Relationship Between TypeScript and JavaScript

Think of TypeScript as a more advanced version of JavaScript. Browsers and other JavaScript environments cannot execute TypeScript directly. Therefore, you need a special program called the TypeScript compiler to "transpile" or convert your TypeScript code into standard JavaScript code that can run anywhere.

3. Overview of the TypeScript Compiler (tsc)

The TypeScript compiler, known as `tsc`, is a command-line tool that reads your TypeScript files (with a `.ts` extension) and generates corresponding JavaScript files (with a `.js` extension). During this process, it checks your code for any type-related errors.

The compiler is highly configurable through a file called `tsconfig.json`, which allows you to specify various options for how your code should be compiled.

4. Activity: Setting Up a Basic TypeScript Project

Let's get our hands dirty and set up a simple TypeScript project.

1. **Install Node.js:** If you don't have it already, download and install Node.js from the official website. This will also install the Node Package Manager (npm).

2. **Install TypeScript:** Open your terminal or command prompt and run the following command to install TypeScript globally on your system:

   ```
   npm install -g typescript
   ```

3. **Create a Project Folder:** Make a new folder for your project and navigate into it in your terminal.

   ```
   mkdir typescript-day1
   cd typescript-day1
   ```

4. **Initialize a TypeScript Configuration File:** Run the following command to create a `tsconfig.json` file. This file tells the TypeScript compiler how to handle your project.

   ```
   tsc --init
   ```

5. **Create Your First TypeScript File:** Create a new file named `index.ts` and add the following code:

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

6. **Compile Your TypeScript Code:** In your terminal, run the compiler:

   ```
   tsc
   ```

   This will generate a new `index.js` file.

7. **Run Your JavaScript Code:** You can now run the compiled JavaScript file using Node.js:

   ```
   node index.js
   ```

   You should see “Hello, TypeScript!” printed in your console.

Basic Types and Type Annotations (30 minutes)

In TypeScript, you can explicitly define the type of a variable. This is called **type annotation**.

1. Number, String, Boolean, Null, Undefined

These are the fundamental data types in JavaScript, and they work the same way in TypeScript.

- **`number`**: For all numeric values, including integers and floating-point numbers.
- **`string`**: For textual data.
- **`boolean`**: For `true` or `false` values.
- **`null`**: Represents the intentional absence of any object value.
- **`undefined`**: Represents a variable that has been declared but not yet assigned a value.

2. The `any` and `unknown` Types

- **any**: The any type is a powerful way to opt-out of type-checking for a variable. You can assign any type of value to a variable of type any. Use this sparingly as it undermines the benefits of TypeScript.
- **`unknown`**: The unknown type is a safer alternative to any. You can assign any value to a variable of type unknown, but you cannot perform any operations on it until you have performed a type check.

3. Type Inference: Letting TypeScript Figure Out the Types

TypeScript is smart! If you declare and initialize a variable in the same statement, TypeScript will infer the type for you.

4. Working with Arrays and Tuples

- **Arrays**: You can define an array of a specific type.
- **Tuples**: Tuples are a special kind of array with a fixed number of elements of known types. The order of the types matters.

5. Hands-on: Declaring Variables with Different Basic Types and Creating Typed Arrays

Let’s practice what we’ve learned. In your `index.ts` file, try the following:

```
// Basic Types
let age: number = 30;
let firstName: string = "John";
let isStudent: boolean = false;

// Type Inference
let lastName = "Doe"; // TypeScript infers this as a string

// any and unknown
let anything: any = "This could be anything";
anything = 42;

let maybe: unknown = "This is a string";
// The following line would cause an error until we check the type
// console.log(maybe.length);

if (typeof maybe === "string") {
  console.log(maybe.length);
}

// Arrays
let numbers: number[] = [1, 2, 3, 4, 5];
let names: string[] = ["Alice", "Bob", "Charlie"];

// Tuples
let person: [string, number] = ["David", 25];

// Let's log these to the console
console.log(age, firstName, isStudent);
console.log(lastName);
console.log(numbers);
console.log(person);
```

After adding this code, re-compile your `index.ts` file with `tsc` and run the resulting `index.js` file with `node index.js`.

Q&A and Wrap-up (15 minutes)

- **Review of Key Concepts:**
  - TypeScript is a superset of JavaScript that adds static typing.
  - The TypeScript compiler (`tsc`) converts TypeScript code to JavaScript.
  - We've learned about basic types: `number`, `string`, `boolean`, `null`, `undefined`, `any`, and `unknown`.
  - TypeScript can infer types.
  - We can create typed arrays and tuples.
- **Preview of Day 2:**
  - Tomorrow, we will dive into how to apply types to functions and how to structure our data more effectively using objects and interfaces.

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`:

```
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.

Day 3: Classes, Enums, and Union/Intersection Types

Objective: To explore object-oriented programming with classes, learn about enums, and combine types using union and intersection.

Classes in TypeScript (25 minutes)

Classes are a fundamental part of object-oriented programming (OOP). They act as blueprints for creating objects that have their own properties (data) and methods (functions).

1. Defining Classes, Properties, Constructors, and Methods

A class definition includes:

- **Properties:** Variables that belong to the class.
- **Constructor:** A special method for creating and initializing an object instance of the class. It's called when you use the new keyword.
- **Methods:** Functions that define the behavior of the objects created from the class.

```
class Vehicle {
  // Properties
  make: string;
  model: string;
  year: number;

  // Constructor
  constructor(make: string, model: string, year: number) {
    this.make = make;
    this.model = model;
    this.year = year;
  }

  // Method
  getDetails(): string {
    return `${this.year} ${this.make} ${this.model}`;
  }
}
```

2. Access Modifiers: `public`, `private`, and `protected`

TypeScript provides keywords to control the visibility of class members:

- **`public` (default):** The member can be accessed from anywhere.
- **`private`:** The member can only be accessed from within the class itself.
- **`protected`:** The member can be accessed from within the class and by any subclasses that extend it.

```
class Employee {
  public name: string;
  private salary: number;

  constructor(name: string, salary: number) {
    this.name = name;
    this.salary = salary;
  }

  public getAnnualSalary(): number {
    return this.salary * 12;
  }
}

const emp = new Employee("John", 50000);
console.log(emp.name); // OK
// console.log(emp.salary); // Error: Property 'salary' is private.
```

3. Implementing Interfaces in Classes

You can use an interface to ensure a class has a specific structure. The `implements` keyword is used to check if the class meets the contract of the interface.

```
interface ILoggable {
  log(): void;
}

class Product implements ILoggable {
  name: string;

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

  log(): void {
    console.log(`Product: ${this.name}`);
  }
}
```

4. Hands-on: Creating a Class with Properties, Methods, and Access Modifiers

Let’s create a `Student` class. Add this to your `index.ts` file:

```
class Student {
  public readonly studentId: number;
  private grades: number[] = [];

  constructor(public name: string) {
    // A parameter property like 'public name' is a shorthand
    // for declaring and initializing a property.
    this.studentId = Math.floor(Math.random() * 1000);
  }
  addGrade(grade: number): void {
    if (grade >= 0 && grade <= 100) {
      this.grades.push(grade);
    }
  }

  getAverageGrade(): number {
    if (this.grades.length === 0) return 0;
    const sum = this.grades.reduce((total, grade) => total + grade, 0);
    return sum / this.grades.length;
  }
}

const student1 = new Student("Alice");
student1.addGrade(95);
student1.addGrade(88);
console.log(
  `${student1.name}'s average grade is ${student1.getAverageGrade()}`
);
```

Enums (10 minutes)

Enums (enumerations) allow you to define a set of named constants. They make your code more readable and less prone to errors from using magic numbers or strings.

1. What Are Enums and When to Use Them?

Use enums when you have a value that can only be one of a small set of possible values. Examples include user roles (Admin, Editor, Viewer), status codes (Pending, Approved, Rejected), or directions (North, South, East, West).

2. Numeric and String Enums

- **Numeric Enums:** By default, enums are number-based. The first value is `0`, and the rest auto-increment. You can also set the starting value.
- **String Enums:** You can also use strings for your enum values. This can be more descriptive.

```
// Numeric Enum
enum OrderStatus {
  Pending, // 0
  Shipped, // 1
  Delivered, // 2
  Cancelled, // 3
}

// String Enum
enum UserRole {
  Admin = "ADMIN",
  Editor = "EDITOR",
  Viewer = "VIEWER",
}
```

3. Activity: Defining and Using an Enum

Let’s use the `OrderStatus` enum. Add this to your `index.ts`:

```
let myOrderStatus: OrderStatus = OrderStatus.Pending;
console.log(`My order status is: ${myOrderStatus}`); // Outputs: 0

myOrderStatus = OrderStatus.Shipped;
console.log(`My order status is now: ${myOrderStatus}`); // Outputs: 1

function processOrder(status: OrderStatus) {
  if (status === OrderStatus.Pending) {
    console.log("Processing a pending order.");
  }
}

processOrder(myOrderStatus);
```

Union and Intersection Types (20 minutes)

These advanced types provide more flexibility in how you define your data structures.

1. Union Types (|)

A union type allows a variable to be one of several types. The vertical bar | acts as an “OR”.

```
let id: string | number;

id = 101; // OK
id = "abc-123"; // OK
// id = false; // Error: Type 'boolean' is not assignable to type 'string | number'.
```

2. Intersection Types (`&`)

An intersection type combines multiple types into a single type with all the properties of the combined types. The ampersand & acts as an “AND”.

```
interface Draggable {
  drag(): void;
}

interface Resizable {
  resize(): void;
}

type UIWidget = Draggable & Resizable;

let widget: UIWidget = {
  drag: () => console.log("Dragging..."),
  resize: () => console.log("Resizing..."),
};
```

3. Type Narrowing with `typeof` and `instanceof`

When you have a union type, you often need to check what the type currently is before you can perform certain operations. This is called type narrowing.

- Use `typeof` for primitive types (`string`, `number`, `boolean`).
- Use `instanceof` for checking class instances.

4. Hands-on: Writing a Function That Accepts a Union Type

Let's write a function that can accept either a single string or an array of strings and prints them. This demonstrates a union type and type narrowing. Add this to your `index.ts`:

```
function printItems(items: string | string[]): void {
  if (typeof items === "string") {
    // Here, TypeScript knows 'items' is a string
    console.log(items);
  } else {
    // Here, TypeScript knows 'items' is a string[]
    items.forEach((item) => console.log(item));
  }
}

printItems("Hello"); // Works with a single string
printItems(["Apple", "Banana", "Cherry"]); // Works with an array of strings
```

Q&A and Wrap-up (5 minutes)

- **Review of Key Concepts:**
  - **Classes** provide a blueprint for creating objects with properties and methods, supporting OOP principles.
  - **Enums** give friendly names to a set of related constants.
  - **Union (|)** and **Intersection (&)** types allow for more flexible and composite type definitions.
- **Preview of Day 4:**
  - Tomorrow, in our final session, we will cover two powerful features: **Generics**, for writing reusable, type-safe components, and built-in **Utility Types** that help you transform existing types.

Day 4: Generics, Utility Types, and Final Review

Objective: To understand how to write reusable, type-safe code with generics and leverage built-in utility types.

Generics (25 minutes)

Generics are one of the most powerful features in TypeScript. They allow you to write flexible, reusable code that can work over a variety of types rather than a single one, while still maintaining type safety. Think of them as a placeholder for a type that will be specified later.

1. Introduction to Generics for Creating Reusable Components

Imagine you need a function that takes an argument and returns it. Without generics, you might write it using any, but you would lose type information.

```
// Without generics (loses type information)
function identity(arg: any): any {
  return arg;
}

let output = identity("myString"); // output is of type 'any'
```

Generics solve this by capturing the type of the argument. We use a *type variable*, a special kind of variable that works on types rather than values. A common name for a type variable is `T`.

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

let output = identity<string>("myString"); // output is of type 'string'
let numOutput = identity(123); // Type is inferred as 'number'
```

2. Generic Functions and Interfaces

You can use generics with functions, interfaces, and classes.

- **Generic Functions:** As seen above, they can operate on any type passed in.
- **Generic Interfaces:** You can create interfaces that are flexible in the types of their properties.

```
interface Box<T> {
  contents: T;
}

let stringBox: Box<string> = { contents: "hello" };
let numberBox: Box<number> = { contents: 100 };
```

3. Generic Constraints

Sometimes you want to write a generic function, but you need to ensure the type being passed in has certain properties. You can use the `extends` keyword to create a constraint.

For example, let's write a function that logs the length of its argument. Not all types have a `length` property, so we need to constrain our generic type.

```typescript
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length); // Now we know it has a .length property
  return arg;
}

loggingIdentity("hello"); // Works because strings have a length
loggingIdentity([1, 2, 3]); // Works because arrays have a length
// loggingIdentity(123); // Error: number does not have a .length property
```

4. Hands-on: Creating a Generic Function

Let's create a generic function that takes an array of any type and returns the first element. Add this to your `index.ts`:

```typescript
function getFirstElement<T>(arr: T[]): T | undefined {
  return arr.length > 0 ? arr[0] : undefined;
}

const numbers = [10, 20, 30];
const firstNum = getFirstElement(numbers); // Inferred as type 'number'
console.log(`First number: ${firstNum}`);

const words = ["Apple", "Banana", "Cherry"];
const firstWord = getFirstElement(words); // Inferred as type 'string'
console.log(`First word: ${firstWord}`);
```

Utility Types (20 minutes)

TypeScript comes with several built-in utility types that help with common type transformations. They take an existing type and modify it in some way. Let's look at the most common ones.

1. Overview of Commonly Used Utility Types

Let's start with a base interface to work with:

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

- **`Partial<T>`:** Constructs a type with all properties of `T` set to optional. This is great for update functions where you might only be changing a few properties.

  ```
  // All properties of UserUpdate are optional
  type UserUpdate = Partial<User>;
  // const userToUpdate: UserUpdate = { name: "New Name" };
  ```

- **`Readonly<T>`:** Constructs a type with all properties of `T` set to `readonly`. This is useful for creating immutable data.

  ```
  // All properties of ReadonlyUser cannot be reassigned
  type ReadonlyUser = Readonly<User>;
  ```

- **`Pick<T, K>`:** Constructs a type by picking a set of properties `K` from `T`.

  ```
  // UserDisplay only has 'name' and 'email' properties
  type UserDisplay = Pick<User, "name" | "email">;
  ```

- **`Omit<T, K>`:** Constructs a type by picking all properties from `T` and then removing `K`.

  ```
  // PublicUser has all properties of User EXCEPT 'isAdmin'
  type PublicUser = Omit<User, "isAdmin">;
  ```

2. Activity: Using Utility Types

Let's use these utility types in a practical example. Add this to your `index.ts`:

```
interface ToDo {
  title: string;
  description: string;
  completed: boolean;
}

// 1. A preview of the ToDo item, only showing the title and completion status.
type ToDoPreview = Pick<ToDo, "title" | "completed">;

const todoPreview: ToDoPreview = {
  title: "Clean room",
  completed: false,
};
console.log("ToDo Preview:", todoPreview);

// 2. A function that takes a ToDo and an object with properties to update.
function updateToDo(todo: ToDo, fieldsToUpdate: Partial<ToDo>): ToDo {
  return { ...todo, ...fieldsToUpdate };
}

const myTodo: ToDo = {
  title: "Learn TypeScript",
  description: "Finish the 4-day course",
  completed: false,
};

const updatedTodo = updateToDo(myTodo, { completed: true });
console.log("Updated ToDo:", updatedTodo);
```

Course Review and Next Steps (15 minutes)

1. Recap of All Topics Covered

- **Day 1:** We learned what TypeScript is, how it adds static typing to JavaScript, and how to use basic types like `string`, `number`, `boolean`, `array`, and `tuple`.
- **Day 2:** We dove into typing functions, defining object shapes, and creating reusable structures with `interfaces` and `type` aliases.
- **Day 3:** We explored object-oriented programming with `classes`, learned how to create named constants with `enums`, and combined types using `union` and `intersection` types.
- **Day 4:** We finished with powerful, advanced features like `generics` for creating reusable components and `utility types` for transforming existing types.

2. Best Practices and Further Learning Resources

- **Enable Strict Mode:** In your `tsconfig.json`, set `"strict": true`. This enables a wide range of type-checking behavior that helps you catch more errors.
- **Avoid any:** The any type disables type-checking. Use it only as a last resort. Prefer unknown when you don't know the type.
- **Official TypeScript Website:** The official documentation at typescriptlang.org is the best place to learn. The Handbook and Playground are excellent resources.
- **Practice:** The best way to learn is by building projects. Try converting a small JavaScript project to TypeScript.

3. Final Q&A Session

This is your opportunity to ask any lingering questions about the topics we've covered or about using TypeScript in general.