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.

```
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`:

```
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:

```
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.