Topic 2: Generics (Reusable, Type-Safe Components and Functions)

1. Problem Statement

EduFlow's Growing Toolbox Challenge

EduFlow Academy is expanding its content types: quizzes, video lessons, coding exercises, and more. The development team notices they keep writing very similar code for handling lists of different content types, grading submissions, and managing feedback.

The problem:

- Duplicated code everywhere, hard to maintain.
- Every new content type means rewriting similar functions and classes.
- Risk of bugs and inconsistent behavior.

Your mission:
Create reusable, flexible tools that work with **any** kind of content or data, while keeping the safety and clarity of TypeScript's types.

Expected outcome:

- Write generic classes and functions that adapt to different data types.
- Avoid code duplication and improve maintainability.
- Keep type safety so errors are caught early.

2. Learning Objectives

By the end of this tutorial, you will be able to:

- Understand what generics are and why they matter.
- Write generic functions and classes that work with any data type.
- Use generics to build reusable components for EduFlow's diverse content.
- Maintain type safety while writing flexible code.

3. Concept Introduction with Analogy

The Magic Toolbox Analogy

Imagine you have a magical toolbox that can change shape to hold any tool you need:

- Today it's a box for quiz papers.
- Tomorrow it transforms into a shelf for video lessons.
- Next week it becomes a cabinet for coding exercises.

Generics in TypeScript work like this magical toolbox. You write one tool or function, and it adapts to whatever data you give it-without losing track of what's inside.

What Are Generics?

Generics allow you to write **reusable components** that work with multiple types while retaining type information.
They are a form of **parametric polymorphism**: you write code with type "parameters" (like `<T>`) that are filled in with actual types when your code is used.

- **Without generics:** You write the same function/class over and over for each type.
- **With generics:** You write it once, and TypeScript ensures it works for any type you specify

A **type parameter** is a placeholder for a type, just like a function parameter is a placeholder for a value.

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

-  `T`  is a type parameter.

-  `identity<number>(42)`  returns a  `number`.

-  `identity<string>("hello")`  returns a  `string`.

-  TypeScript  **infers**  `T`  if you don't specify it:  `identity(true)`  →
`T`  is  `boolean`.
```

Type Inference with Generics

TypeScript is smart:
If you call `identity("test")` , it knows `T` is `string` and enforces that everywhere inside the function.

```
const result = identity("test"); // Type: string

If you try to use  `identity<number>("oops")`, TypeScript will error:
```

> Argument of type 'string' is not assignable to parameter of type 'number'.

Generic Functions vs. Any

Compare:

```
function echoAny(arg: any): any { return arg; }
function echoGeneric<T>(arg: T): T { return arg; }
```

```
let x = echoAny(123); // x: any (no type safety)
let y = echoGeneric(123); // y: number (type-safe)
```

Generics preserve type information, `any` does not.

Generic Classes

You can use generics in classes to make them reusable for any type.

```
class Box<T> {
contents: T;
constructor(value: T) {
  this.contents = value;
}
}

const stringBox = new Box("hello"); // Box<string>
const numberBox = new Box(42);      // Box<number>
```

- Each instance of `Box` remembers the type you gave it.

Generic Interfaces and Types

You can define interfaces and type aliases with generics:

```
interface ApiResponse<T> {
data: T;
status: number;
}

type Pair<K, V> = { key: K; value: V };
```

Generic Constraints

You can restrict what types a generic can accept using `extends`.

```
interface HasId { id: string; }

function printId<T extends HasId>(item: T): void {
console.log(item.id);
}

printId({ id: "abc", name: "Alice" }); // OK
printId({ name: "Bob" }); // Error: Property 'id' is missing
```

- `T extends HasId` means “T must have at least the properties of HasId”.

Multiple Type Parameters

Generics can take more than one type parameter:

```
function merge<A, B>(a: A, b: B): A & B {
return { ...a, ...b };
}

const merged = merge({ id: 1 }, { name: "Alice" }); // { id: 1, name: "Alice" }
```

Default Type Parameters

You can provide default types for generics:

```
type ApiResponse<T = string> = {
data: T;
status: number;

const resp: ApiResponse = { data: "OK", status: 200 }; // T is string by default
};
```

Utility Types (Built-in Generics)

TypeScript provides many built-in generic types:

```
-   `Partial<T>`: All properties optional.

-   `Readonly<T>`: All properties readonly.

-   `Record<K, T>`: Object with keys of type K and values of type T.

-   `Pick<T, K>`: Object with only properties K from T.

-   `Omit<T, K>`: Object with all properties of T except K.

type User = { id: string; name: string; age: number; };
type UserPreview = Pick<User, "id" | "name">; // { id: string; name: string }
```

Example: Generic Function

4. Step-by-Step Data Modeling

Let's start with a simple list class that can hold any kind of content:

```
class List<T> {
 private items: T[] = [];

 add(item: T) {
   this.items.push(item);
 }

 getAll(): T[] {
   return [...this.items];
 }
}
```

- `T` is a placeholder for any type you want to use.
- When you create a `List`, you specify what type it holds (e.g., `Quiz`, `Lesson`).

5. Live Code Walkthrough

A. The Pain Without Generics

Suppose EduFlow wants to store feedback for quizzes and lessons.
Without generics, you'd write two almost identical classes:

```
// One for quizzes
class QuizFeedbackBox {
 private feedbacks: string[] = [];
 addFeedback(feedback: string) { this.feedbacks.push(feedback); }
 getAllFeedback(): string[] { return [...this.feedbacks]; }
}

// One for lessons
class LessonFeedbackBox {
 private feedbacks: string[] = [];
 addFeedback(feedback: string) { this.feedbacks.push(feedback); }
 getAllFeedback(): string[] { return [...this.feedbacks]; }
}
```

Problem:

- Lots of copy-pasting.
- If you want to store feedback as objects (not just strings), you have to rewrite everything again.
- Easy to make mistakes or forget to update both classes.

B. The Magic of Generics

Now, let's solve it with a single, flexible class:

```
// Generic FeedbackBox: works for any type!
class FeedbackBox<T> {
 private feedbacks: T[] = [];

 addFeedback(feedback: T) {
   this.feedbacks.push(feedback);
 }

 getAllFeedback(): T[] {
   return [...this.feedbacks];
 }
}
```

- `T` is a placeholder for any type (string, object, number, etc.).
- When you use `FeedbackBox`, you decide what `T` is.

C. Using the Generic Class

```
// For quiz feedback (as strings)
const quizFeedback = new FeedbackBox<string>();
quizFeedback.addFeedback("Great quiz!");
quizFeedback.addFeedback("Too hard!");
console.log(quizFeedback.getAllFeedback()); // ["Great quiz!", "Too hard!"]

// For lesson feedback (as objects)
type LessonFeedback = { rating: number; comment: string };
const lessonFeedback = new FeedbackBox<LessonFeedback>();
lessonFeedback.addFeedback({ rating: 5, comment: "Loved it!" });
console.log(lessonFeedback.getAllFeedback()); // [{ rating: 5, comment: "Loved it!" }]
```

Why is this better?

- You write the code once, use it everywhere.

- TypeScript checks that you only add the right kind of feedback (prevents mistakes).

- If you want to store new types of feedback in the future, you don't need to rewrite the class.

D. Generic Functions in Action

```
function getFirstItem<T>(items: T[]): T | undefined {
 return items[0];
}

const firstQuizFeedback = getFirstItem(quizFeedback.getAllFeedback()); // string
const firstLessonFeedback = getFirstItem(lessonFeedback.getAllFeedback());
// LessonFeedback object
```

- The function works for any array type.

- TypeScript always knows what type you're working with.

```
| Feature         | Syntax Example                            | Purpose
|-----------------|-------------------------------------------|-----------------------------
| Generic Func    | `function f<T>(x: T): T { ... }`          | Reusable, type-safe function
| Generic Class   | `class Box<T> { ... }`                    | Reusable, type-safe classes
| Constraint      | `function f<T extends U>(x: T) { ... }`   | Restrict types
| Multiple Params | `function f<A, B>(a: A, b: B)`            | Combine types
| Default Param   | `type Resp<T = string> = ...`             | Fallback type
| Utility Types   | `Partial<T>`, `Pick<T, K>`, etc.          | Common transformations
```

6. Challenge

Your Turn!

```
1.  Write a generic class  `FeedbackBox<T>`  that stores feedback items of any type and
lets you retrieve them all.

3.  Write a generic function  `getFirstItem<T>`  that returns the first item from any array
```

7. Quick Recap & Key Takeaways

- Generics allow you to write flexible, reusable code.

- They keep type safety by remembering what type you're working with.

- You can create generic classes and functions that work with any data type.

- This reduces duplication and improves maintainability.

8. Optional: Programmer's Workflow Checklist

- Look for repeated code that only differs by data type.

```
-   Replace specific types with generics (`<T>`).
```

- Use generics in classes and functions.

- Test with multiple data types to ensure flexibility and safety.

9. Coming up next

Master **Advanced Types**-your toolkit for combining, transforming, and adapting types to handle complex real-world data scenarios in EduFlow!
Think of it as customizing your magical toolbox with special attachments.