Angular TS – Generics in TypeScript

[**Task 4: Generics — Focus: Reusable Components** 1](#_Toc205979837)

[**1. Purpose** 1](#_Toc205979838)

[**2. Theory** 1](#_Toc205979839)

[**3. Prerequisites** 2](#_Toc205979840)

[**4. Code Example (Step-by-Step)** 2](#_Toc205979841)

[**5. Build & Run Procedure** 3](#_Toc205979842)

[**6. Project Structure Snapshot** 3](#_Toc205979843)

[**7. Summary** 3](#_Toc205979844)

**Task 4: Generics — Focus: Reusable Components**

**1. Purpose**

Generics in TypeScript allow you to create reusable, type-safe components, functions, and classes that can work with different data types while still maintaining strong type checking.  
They help avoid duplicate code and make your code more flexible without losing type safety.

**2. Theory**

* **Why Generics?**  
  Without generics, you might use any type for flexibility, but you lose type safety. Generics allow flexibility *and* type safety.
* **Generic Syntax:**
* function identity<T>(value: T): T {
* return value;
* }
* **Where Generics Can Be Used**
  + Functions
  + Interfaces
  + Classes
  + Type Aliases
* **Multiple Type Parameters:**  
  You can define more than one type parameter:
* function pair<K, V>(key: K, value: V): [K, V] {
* return [key, value];
* }
* **Generic Constraints:**  
  Restrict the kind of types allowed:
* function logLength<T extends { length: number }>(item: T) {
* console.log(item.length);
* }

**3. Prerequisites**

* Node.js and TypeScript installed (npm install -g typescript)
* Working TypeScript project folder (like we created earlier)
* Familiarity with TypeScript basic types, functions, and classes

**4. Code Example (Step-by-Step)**

**File:** ang-generics.ts

// 1. Generic function

function identity<T>(value: T): T {

return value;

}

let numResult = identity<number>(100);

let strResult = identity<string>("Hello Generics");

console.log("Generic Function Results:");

console.log(numResult);

console.log(strResult);

// 2. Generic interface

interface KeyValuePair<K, V> {

key: K;

value: V;

}

let kv1: KeyValuePair<number, string> = { key: 1, value: "One" };

let kv2: KeyValuePair<string, boolean> = { key: "isActive", value: true };

console.log("Generic Interface Results:");

console.log(kv1);

console.log(kv2);

// 3. Generic class

class Box<T> {

private \_content: T;

constructor(content: T) {

this.\_content = content;

}

getContent(): T {

return this.\_content;

}

}

let numberBox = new Box<number>(123);

let stringBox = new Box<string>("TypeScript Box");

console.log("Generic Class Results:");

console.log(numberBox.getContent());

console.log(stringBox.getContent());

// 4. Generic constraint

function logLength<T extends { length: number }>(item: T) {

console.log("Length:", item.length);

}

logLength("Hello");

logLength([1, 2, 3]);

**5. Build & Run Procedure**

# Compile TypeScript file to JavaScript

tsc ang-generics.ts

# Run the compiled JavaScript file

node ang-generics.js

**6. Project Structure Snapshot**

typescript-fundamentals/

│ ang-typesafety.ts

│ ang-interfaces-enums.ts

│ ang-oops.ts

│ ang-generics.ts

│ tsconfig.json

**7. Summary**

Generics allow you to write reusable and type-safe code in TypeScript.  
They can be used with functions, interfaces, classes, and type aliases, ensuring flexibility without sacrificing type safety.