TypeScript (TS)

Amr A Khllaf

August 1, 2025

Contents

1 TS History, Wht We need it and How To Convert it into JS Code? 2
1.1 What is TypeScript? . . . 2
1.2 Why Do We Need TypeScript? . . . 3
1.3 How to install TypeScript? . . . 3
1.4 How to Convert TypeScript to JavaScript? . . . 3

2 TS Types and Solving the Re-declared Variables Scope 4
2.1 Types in TypeScript . . . 4
2.2 Solving the Re-declared Variables Scope . . . 4
2.3 Primitive Data Type . . . 5
2.4 Array Type . . . 6
2.5 Any Type . . . 6
2.6 Tuple Type . . . 7
2.7 Union Type . . . 7
2.8 Intersection Type . . . 7
2.9 Intersection types can also be used with arrays: . . . 8
2.10 Enum Type . . . 8
2.10.1 Enums can also be used with arrays: . . . 9
2.11 Never Type . . . 9
2.12 Void Type . . . 9

3 Functions types in TypeScript 10
3.1 Function Types . . . 10
3.1.1 Parameter With Data Types . . . 11
3.2 Determining the return type of a function . . . 11

4 Objects (Classes) and Interfaces in TypeScript 12
4.1 Objects and Classes in TypeScript . . . 12
4.2 Interfaces in TypeScript . . . 13
4.3 Key Differences: Class vs. Interface . . . 13
4.4 Using Interfaces with Classes . . . 14
4.5 Type Aliases for Object Types . . . 14
4.6 Summary . . . 15

5 Types and Differences Between Them and Interfaces 15
5.1 Types in TypeScript . . . 15

5.2 Differences Between Types and Interfaces 15
5.3 Extends in Types and Interfaces 16
5.3.1 Extending Interfaces 16
5.3.2 Extending Types 17
5.4 When to Use Types vs. Interfaces 17

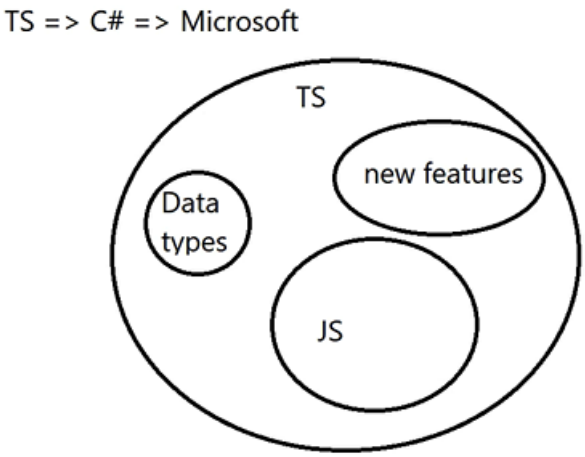
6 Generics in TypeScript **18**
6.1 What are Generics? 18
6.2 Why Use Generics? 18
6.3 How to Use Generics 18
6.4 Why Do We Need Generics? (The Problem Generics Solve) 19
6.5 Solution: Using Generics 19
6.6 Generics with the Types and Interfaces 19
6.6.1 The Problem: Repeated Type and Interface Declarations 19
6.6.2 Solution: Using Generics with Types and Interfaces 20

7 TS in React and Generics Hooks **21**
7.1 Steps to create a React app with TypeScript 21

1 TS History, Wht We need it and How To Convert it into JS Code?

1.1 What is TypeScript?

- TypeScript is a superset of JavaScript that **adds static types (Data Types)** to the language.
- It was developed by Microsoft and first released in 2012.
- TypeScript is designed to help developers write more robust and maintainable code by catching errors at compile time rather than runtime.
- It compiles down to plain JavaScript, which means it can run anywhere JavaScript runs.
- TypeScript supports modern JavaScript features and provides additional features like interfaces, enums, and generics.
- Learn More from the Official Site: TypeScript Site
- TypScript is super set of JavaScript, which means that all JavaScript code is also valid



TypeScript code.

- TypeScript is made to reduce the number of runtime errors by providing a type system that can catch errors at compile time.
- TypeScript is more predictable than JavaScript, which makes it easier to maintain and refactor code.

1.2 Why Do We Need TypeScript?

- **Static Typing**: TypeScript allows developers to define types for variables, function parameters, and return values, which helps catch errors early in the development process.
- **Enhanced Tooling**: TypeScript provides better tooling support, including autocompletion, type checking, and refactoring tools, which can improve developer productivity.
- **Improved Readability**: Type annotations make the code more readable and self-documenting, making it easier for developers to understand the codebase.
- **Compatibility**: TypeScript is compatible with existing JavaScript code, allowing developers to gradually adopt it in their projects without needing to rewrite everything.
- **Community and Ecosystem**: TypeScript has a large and active community, with many libraries and frameworks supporting it, making it easier to find resources and solutions.

1.3 How to install TypeScript?

- TypeScript can be installed globally using npm (Node Package Manager) with the following command:

```
npm install -g typescript
```

- This command installs the TypeScript compiler (`tsc`) globally on your system, allowing you to compile TypeScript files from the command line.

1.4 How to Convert TypeScript to JavaScript?

- TypeScript code is transpiled to JavaScript using the TypeScript compiler (`tsc`).
- The compiler checks for type errors and generates equivalent JavaScript code that can run in any JavaScript environment.
- To convert TypeScript to JavaScript, you can use the following command in your terminal:

```
tsc yourfile.ts
```

- This command will generate a JavaScript file (`yourfile.js`) in the same directory as the TypeScript file.
- If i want to compile all the TypeScript files in a directory, you can run:

```
tsc *.ts
```

- If i want to make the compile process automatic, i can use the `--watch` flag:

```
tsc yourfile.ts --watch
```

- This will watch for changes in TypeScript files and automatically recompile them to JavaScript whenever a change is detected.

---

2 TS Types and Solving the Re-declared Variables Scope

2.1 Types in TypeScript

- TypeScript supports various data types, including:
  - **Primitive Types**: `number`, `string`, `boolean`, `null`, `undefined`, and `symbol`.
  - **Array Types**: Arrays can be defined using the syntax `type[]` or `Array<type>`.
  - **Tuple Types**: Fixed-length arrays with specific types for each element, defined as `[type1, type2]`.
  - **Enum Types**: Enumerations that define a set of named constants.
  - **Any Type**: A type that can hold any value, used when the type is unknown.
  - **Void Type**: Used for functions that do not return a value.
  - **Never Type**: Represents values that never occur, such as functions that throw errors or infinite loops.
- **Object Types**: Used to define complex data structures with properties and methods.
- **Function Types**: Functions can be typed by specifying the parameter types and return type.
- **Union Types**: Allow a variable to hold values of multiple types, defined using the `|` operator.
- **Intersection Types**: Combine multiple types into one, allowing a variable to have properties of all combined types.

2.2 Solving the Re-declared Variables Scope

- In TypeScript, variables have block scope, which means they are only accessible within the block they are defined in.
- This helps prevent issues with variable hoisting and re-declaration that can occur in JavaScript.
- To further enhance variable scoping, TypeScript introduces the `let` and `const` keywords, which allow for more precise control over variable lifetimes and mutability.

- The `let` keyword allows you to declare variables that can be reassigned, while `const` is used for variables that should not be reassigned.
- TypeScript also supports namespaces and modules to organize code and avoid naming conflicts, providing a way to encapsulate variables and functions within a specific scope.
- Here's an example of how to use TypeScript types and variable scoping:

```
let x: number = 10; // Number type
const y: string = "Amr"; // String type
function greet(name: string): void {
  console.log(`Hello, ${name}!`);
}
```

2.3 Primitive Data Type

- In this example, `x` is a number, `y` is a string, and the `greet` function takes a string parameter and returns nothing (`void`). The use of types helps catch errors at compile time, ensuring that the correct types are used throughout the code.
- If you try to re-declare a variable with the same name in the same scope, TypeScript will throw an error, helping you avoid potential bugs:

```
let x: number = 10;
let x: string = "Hello"; // Error: Cannot redeclare block-scoped variable
↪  'x'.
```

- This ensures that variables are not accidentally re-declared, which can lead to unexpected behavior in your code.
- There is an Error with let when i compile it to JavaScript, it will give me an error that says `Cannot redeclare block-scoped variable 'x'` because `let` and `const` are block-scoped, meaning they cannot be re-declared in the same scope.
- **We Can Solve it by IIFE (Immediately Invoked Function Expression)** is the best way to avoid this error, as it creates a new scope for the variable:

```
(function () {
  let x: number = 10;
  let y: string = "Hello";
  console.log(x, y);
})();

---

(function () {
  let x: number = 20; // This 'x' is different from the previous 'x'
  console.log(x); // Outputs: 20
})();

---

(function () {
  const z: boolean = true; // 'const' variable, cannot be re-declared
  console.log(z); // Outputs: true
```

```
})();

---

(function () {
  let a: number = 5;
  let b: string = "TypeScript";
  console.log(`a: ${a}, b: ${b}`); // Outputs: a: 5, b: TypeScript
})();

---
(function () {
  let c: number = 15;
  console.log(`c: ${c}`); // Outputs: c: 15
})();
```

2.4 Array Type

- TypeScript allows you to define arrays with specific types, ensuring that only values of the specified type can be added to the array. Here's an example of defining an array of numbers:

```
let numbers: number[] = [1, 2, 3, 4, 5]; // Array of numbers

---

let strings: Array<string> = ["Hello", "World"]; // Array of strings

let strings2: string[] = ["TypeScript", "JavaScript"]; // Another way to
↪  define an array of strings

---

let mixed: (number | string)[] = [1, "two", 3]; // Array of mixed types
↪  (number or string)
```

2.5 Any Type

- The `any` type allows you to define a variable that can hold any value, bypassing TypeScript's strict type checking. This is useful when you are unsure of the type of a variable or when working with dynamic data. Here's an example:

```
let anyValue: any = 42;
anyValue = "Hello"; // No error
anyValue = true; // No error
```

- With Arrays, you can also use the `any` type to create arrays that can hold mixed types:

```
let anyArray: any[] = [1, "two", true, { key: "value" }]; // Array of any
↪  type
let mixedArray: Array<any> = [42, "Hello", false, [1, 2, 3]]; // Another
↪  way to define an array of any type
```

```
let mixedArray2: (number | string | boolean)[] = [1, "two", true]; // Array
↪  of mixed types
let mixedArray3: Array<number | string | boolean> = [1, "two", true]; //
↪  Another way to define an array of mixed types
```

2.6 Tuple Type

- Tuple types allow you to express an array with a fixed number of elements whose types are known, but need not be the same. Here's an example:

```
let tuple: [number, string, boolean];
tuple = [1, "Hello", true]; // OK
tuple = [2, "World"]; // Error: Source has 2 element(s) but target
↪  requires 3
```

2.7 Union Type

- Union types allow a variable to **hold values of multiple types**. This is useful when you want to allow flexibility in the types that can be assigned to a variable. Here's an example:

```
let value: number | string;
value = 42; // OK
value = "Hello"; // OK
```

- Union types can also be used with arrays:

```
let mixedArray: (number | string)[] = [1, "two", 3, "four"]; // Array of
↪  mixed types
```

2.8 Intersection Type

- Intersection types allow you to combine multiple types into one, meaning a variable can have properties of all combined types. This is useful for creating complex types that require multiple behaviors or properties. Here's an example:

```
type A = { a: number };
type B = { b: string };
type C = A & B; // Intersection type

let obj: C = { a: 1, b: "Hello" }; // OK

---

let obj2: A & B & { c: boolean } = { a: 2, b: "World", c: true }; // OK

let obj3: { a: number } & { b: string } = { a: 3, b: "TypeScript" }; // OK

let obj4: { a: number } & { b: string } & { c: boolean } = { a: 4, b:
↪  "JavaScript", c: false }; // OK

let obj5: { a: number } & { b: string } & { c: boolean } & { d: string } =
↪  { a: 5, b: "Union", c: true, d: "Intersection" }; // OK
```

```
let obj6: { a: number } & { b: string } & { c: boolean } & { d: string } &
↪  { e: number } = { a: 6, b: "Complex", c: false, d: "Type", e: 10 }; //
↪  OK
```

2.9 Intersection types can also be used with arrays:

```
let mixedArray: (number & string)[] = [1, "two", 3, "four"]; // Array of
↪  mixed types
let mixedArray2: Array<number & string> = [1, "two", 3, "four"]; // Another
↪  way to define an array of mixed types
let mixedArray3: (number | string)[] & (boolean | object)[] = [
  1,
  "two",
  true,
  { key: "value" },
]; // Array of mixed types
let mixedArray4: Array<number | string | boolean | object> = [
  1,
  "two",
  true,
  { key: "value" },
]; // Another way to define an array of mixed types
let mixedArray5: (number | string | boolean | object)[] &
  (number | string | boolean | object)[] = [1, "two", true, { key: "value"
↪  }]; // Array of mixed types
```

2.10 Enum Type

- Enums are a way to define a set of named constants in TypeScript. They can be numeric or string-based. Here's an example of a numeric enum:

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
let move: Direction = Direction.Up; // move will be of type Direction
console.log(move); // Outputs: 1
```

- Here's an example of a string enum:

```
enum Color {
  Red = "RED",
  Green = "GREEN",
  Blue = "BLUE",
}
let favoriteColor: Color = Color.Green; // favoriteColor will be of type
↪  Color
console.log(favoriteColor); // Outputs: GREEN
```

2.10.1 Enums can also be used with arrays:

```
let directions: Direction[] = [
  Direction.Up,
  Direction.Down,
  Direction.Left,
  Direction.Right,
]; // Array of Direction enum values
let colors: Color[] = [Color.Red, Color.Green, Color.Blue]; // Array of
↪  Color enum values
let mixedEnums: (Direction | Color)[] = [
  Direction.Up,
  Color.Red,
  Direction.Down,
  Color.Green,
]; // Array of mixed enum types
let mixedEnums2: Array<Direction | Color> = [
  Direction.Left,
  Color.Blue,
  Direction.Right,
  Color.Red,
]; // Another way to define an array of mixed enum types
```

2.11 Never Type

- The `never` type represents values that never occur, such as functions that throw errors or infinite loops. It is often used to indicate that a function will not return a value. Here's an example:

```
function throwError(message: string): never {
  throw new Error(message); // This function will never return
}
function infiniteLoop(): never {
  while (true) {
    // This function will never terminate
  }
}
```

- The `never` type is useful for functions that are expected to throw errors or run indefinitely, as it helps TypeScript understand that these functions will not return a value.

2.12 Void Type

- The `void` type is used to indicate that a function does not return a value. It is often used for functions that perform side effects, such as logging or modifying global state. Here's an example:

```
function logMessage(message: string): void {
  console.log(message); // This function does not return a value
```

```
}
function doNothing(): void {
  // This function does nothing and does not return a value
}
```

- The `void` type is useful for functions that are not intended to return a value, as it helps TypeScript understand that these functions are meant for side effects rather than producing a result.
- The `void` type can also be used with arrays:

```
let voidArray: void[] = []; // Array of void type
let voidArray2: Array<void> = []; // Another way to define an array of
↪  void type
let mixedVoidArray: (void | string)[] = [
  undefined,
  "Hello",
  undefined,
  "World",
]; // Array of mixed types (void or string)
let mixedVoidArray2: Array<void | string> = [
  undefined,
  "TypeScript",
  undefined,
  "JavaScript",
]; // Another way to define an array of mixed types (void or string)
```

---

3 Functions types in TypeScript

- Functions in TypeScript can be typed by specifying the types of their parameters and return values. This helps ensure that functions are called with the correct types and that they return the expected type.

3.1 Function Types

- Here's an example of a simple function with typed parameters and return value:

```
function add(a: number, b: number): number {
  return a + b; // Returns a number
}

let result: number = add(5, 10); // result will be of type number
// You must send the Parameters with the same type as the function
↪  parameters
console.log(result); // Outputs: 15
```

- In this example, the `add` function takes two parameters of type `number` and returns a value of type `number`.

3.1.1 Parameter With Data Types

- You can also define functions with optional parameters and default values:

```
function greet(name: string, age?: number): string {
  if (age) {
    return `Hello, ${name}! You are ${age} years old.`;
  }
  return `Hello, ${name}!`;
}
let greeting1: string = greet("Alice", 30); // With age
let greeting2: string = greet("Bob"); // Without age
```

- In this example, the `age` parameter is optional, meaning you can call the function without providing it. If not provided, it will be `undefined`.
- Another Example:

```
function sayHello(name: string): string {
  return `Hello, ${name}!`;
}

sayHello("Amr").toUpperCase();
```

- Since TypeScript is a superset of JavaScript and provides static typing, it can infer that the return type of `sayHello` is a `string`. This allows you to immediately use string methods like `.toUpperCase()` on the result, with full type safety and autocompletion support in your editor. This predictability makes TypeScript code easier to maintain and refactor.
- If i try to use an Array method on a function that returns a string, TypeScript will throw an error:

```
function sayHello(name: string): string {
  return `Hello, ${name}!`;
}
sayHello("Amr").map((char) => char.toUpperCase()); // Error: Property 'map'
↪  does not exist on type 'string'.
```

- This is because the `map` method is not available on strings, and TypeScript catches this error at compile time, preventing potential runtime errors.

3.2 Determining the return type of a function

- TypeScript can automatically infer the return type of a function based on its implementation. However, you can also explicitly specify the return type if needed. Here's an example:

```
function add(a: number, b: number): number {
  return a + b;
}
let result: number = add(5, 10); // result will be of type number
console.log(result); // Outputs: 15
```

- In this example, TypeScript infers that the return type of the `add` function is `number` based on the return statement. If you try to return a value of a different type, TypeScript

will throw an error:

```
function add(a: number, b: number): number {
  return a + b;
}
function addString(a: number, b: number): string {
  return (a + b).toString(); // Error: Type 'string' is not assignable to
  ↪  type 'number'.
}
```

- If you want to explicitly specify the return type, you can do so like this:

```
function add(a: number, b: number): number {
  return a + b;
}
function addString(a: number, b: number): string {
  return (a + b).toString(); // Now this is valid
}
function addNumbers(a: number, b: number): number {
  return a + b; // Explicitly returning a number
}
```

---

4 Objects (Classes) and Interfaces in TypeScript

TypeScript provides powerful ways to define the structure and behavior of objects through **classes** and **interfaces**. Understanding the differences and use cases for each is essential for writing robust, maintainable code.

4.1 Objects and Classes in TypeScript

A **class** in TypeScript is a blueprint for creating objects with specific properties and methods. Classes can include constructors, methods, and property declarations, and they support features like inheritance and access modifiers.

Example: Defining a Class

```
class Person {
  name: string;
  age: number;
  isStudent?: boolean; // Optional property

  constructor(name: string, age: number, isStudent?: boolean) {
    this.name = name;
    this.age = age;
    this.isStudent = isStudent;
  }

  greet(): string {
```

```
    return `Hello, my name is ${this.name} and I am ${this.age} years
    ↪  old.`;
  }
}

const person1 = new Person("Alice", 30, true);
console.log(person1.greet()); // Outputs: Hello, my name is Alice and I am
↪  30 years old.
```

- **Classes** can be instantiated using the `new` keyword.
- They can provide concrete implementations for methods and properties.
- Classes exist in the compiled JavaScript output and can be used at runtime.

4.2 Interfaces in TypeScript

An **interface** defines the shape or contract of an object. It specifies what properties and methods an object should have, but does not provide any implementation. Interfaces are used only at compile time for type checking and do not appear in the generated JavaScript.

Example: Defining and Using an Interface

```
interface User {
  username: string;
  email: string;
  age?: number; // Optional property
  login?(): void; // Optional method
}

const user: User = {
  username: "Amr_Khllaf",
  email: "amr@example.com",
  age: 25,
  login: function () {
    console.log(`User ${this.username} logged in.`);
  },
};

user.login?.(); // Outputs: User Amr_Khllaf logged in.
```

- **Interfaces** cannot be instantiated.
- They are used for type-checking and ensuring consistency in object structures.
- Interfaces are erased during compilation and do not exist at runtime.

4.3 Key Differences: Class vs. Interface

Feature	Class	Interface
Purpose	Blueprint for creating objects (with implementation)	Contract for object structure (no implementation)
Instantiation	Can be instantiated with `new` keyword	Cannot be instantiated
Implement	Can provide method/property implementations	Only method/property signatures
Inheritance	Single inheritance (can extend one class)	Multiple inheritance (can extend multiple interfaces)
Output	Exists in compiled JavaScript	Does **not** exist in compiled JavaScript

---

4.4 Using Interfaces with Classes

Classes can **implement** interfaces to guarantee they adhere to a specific contract.

```
interface Animal {
  name: string;
  sound(): void;
}

class Dog implements Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  sound(): void {
    console.log(`${this.name} says Woof!`);
  }
}

const dog = new Dog("Buddy");
dog.sound(); // Outputs: Buddy says Woof!
```

- The `Dog` class must provide all properties and methods defined in the `Animal` interface.

4.5 Type Aliases for Object Types

TypeScript also allows you to define object shapes using the `type` keyword, which is similar to interfaces but can represent more complex types (like unions and intersections).

```
type Point = {
  x: number;
  y: number;
};

const point: Point = { x: 10, y: 20 };
console.log(`Point coordinates: (${point.x}, ${point.y})`);
```

4.6 Summary

- **Classes** define both the structure and behavior of objects and exist at runtime.
- **Interfaces** define the expected shape of objects for type-checking and are removed during compilation.
- Use **interfaces** to enforce contracts and ensure consistency, and **classes** to implement logic and create objects.
- Both can be combined for maximum type safety and code clarity in TypeScript.

---

5 Types and Differences Between Them and Interfaces

5.1 Types in TypeScript

- In TypeScript, a **type** can represent a wide range of data structures, including primitive types, object types, union types, intersection types, and more. Types are used to define the shape and behavior of data in your application.

5.2 Differences Between Types and Interfaces

Feature	Type	Interface
Definition	Defines a type alias for a specific shape or union	Defines a contract for object structure
Instantiation	Can be used to create instances (e.g., with `new`)	Cannot be instantiated
Implementation	Can provide implementations for methods	Cannot provide implementations
Extensibility	Can use intersections to combine types	Can extend multiple interfaces
Output	Exists in compiled JavaScript	Does **not** exist in compiled JavaScript
Use Case	Use for complex types, unions, and intersections	Use for defining object shapes and contracts
Compatibility	Can be used with primitive types, arrays, etc.	Primarily used for object-oriented structures
Declaration	Can use `type` keyword	Can use `interface` keyword
Merging	Cannot be merged	Can be merged with other interfaces
Declaration Merging	Not supported	Supported (multiple declarations merge)
Optional Properties	Can use `?` for optional properties	Can use `?` for optional properties
Readonly Properties	Can use `readonly` keyword	Can use `readonly` keyword

Feature	Type	Interface
Data Types	Can represent primitive, union, and intersection types	Primarily used for object types
Function Types	Can define function types with parameters and return	Can define function types with parameters and return
Generics	Can use generics to create reusable types	Can use generics to create reusable interfaces

---

- We can use Enum with Types:

```
type Color = "Red" | "Green" | "Blue";
enum ColorEnum {
  Red = "Red",
  Green = "Green",
  Blue = "Blue",
}
let myColor: Color = "Red"; // Using type
let myColorEnum: ColorEnum = ColorEnum.Red; // Using enum
```

5.3 Extends in Types and Interfaces

- Both types and interfaces can extend other types or interfaces, allowing you to create more complex structures by building on existing ones.

5.3.1 Extending Interfaces

```
interface Animal {
  name: string;
}
interface Dog extends Animal {
  bark(): void;
}

class Labrador implements Dog {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  bark(): void {
    console.log(`${this.name} barks!`);
  }
}

const labrador = new Labrador("Buddy");
labrador.bark(); // Outputs: Buddy barks!
```

5.3.2 Extending Types

```ts
type Animal = {
  name: string;
};
type Dog = Animal & {
  bark(): void;
};
type Labrador = Dog & {
  breed: string;
};
const myLabrador: Labrador = {
  name: "Buddy",
  breed: "Labrador",
  bark: () => console.log("Woof!"),
};
myLabrador.bark(); // Outputs: Woof!

type myDog = Dog & Labrador; // Combining types
const myDog: myDog = {
  // Using combined type
  name: "Charlie",
  breed: "Golden Retriever",
  bark: () => console.log("Woof Woof!"),
};
```

- In this example, both the `Dog` interface and the `Labrador` type extend the `Animal` structure, allowing you to create more specialized types while maintaining the base properties.
- This demonstrates how you can use both interfaces and types to create flexible and reusable structures in TypeScript.

5.4 When to Use Types vs. Interfaces

- **Use Interfaces** when you want to define the shape of an object or class, especially when you need to extend or implement multiple structures. Interfaces are ideal for defining contracts and ensuring consistency in object-oriented programming.
- **Use Types** when you need to create complex types, unions, or intersections, or when you want to define a type alias for a specific structure. Types are more versatile and can represent a wider range of data structures.
- In practice, you can often use either types or interfaces interchangeably for defining object shapes. However, interfaces are generally preferred for defining object-oriented structures due to their extensibility and support for declaration merging.
- Types are more suitable for defining complex data structures, unions, and intersections, making them a powerful tool for creating flexible and reusable types.
- Ultimately, the choice between types and interfaces depends on your specific use case and coding style. Both provide powerful features for type safety and code clarity in TypeScript.

---

6 Generics in TypeScript

- Generics allow you to create reusable components that can work with any data type while maintaining type safety. They enable you to define functions, classes, and interfaces that can operate on different types without losing the benefits of static typing.

6.1 What are Generics?

- Generics are a way to create components that can work with any data type while maintaining type safety. They allow you to define functions, classes, and interfaces that can operate on different types without losing the benefits of static typing.
- Generics are defined using angle brackets (`<T>`) where `T` is a placeholder for the type that will be provided when the generic is used.
- This allows you to create flexible and reusable code that can adapt to different data types without sacrificing type safety.

6.2 Why Use Generics?

- **Reusability**: Generics allow you to write code that can work with multiple data types, reducing code duplication and improving maintainability.
- **Type Safety**: Generics provide compile-time type checking, ensuring that the correct types are used throughout your code, reducing runtime errors.
- **Flexibility**: Generics enable you to create functions and classes that can handle various data types without losing type safety.
- **Improved Readability**: Generics make your code more readable and self-documenting, as the type parameters provide context for the data being used.

6.3 How to Use Generics

- You can define a generic function by using angle brackets (`<T>`) to specify a type parameter. Here's an example of a simple generic function that takes an array of any type and returns the first element:

```
function getFirstElement<T>(arr: T[]): T {
  return arr[0];
}
let numbers = [1, 2, 3, 4, 5];
let firstNumber = getFirstElement(numbers); // firstNumber will be of type
↪  number
let strings = ["Hello", "World"];
let firstString = getFirstElement(strings); // firstString will be of type
↪  string
let mixed = [1, "two", true];
let firstMixed = getFirstElement(mixed); // firstMixed will be of type
↪  (number | string | boolean)
```

- In this example, the `getFirstElement` function is defined as a generic function that takes an array of type `T` and returns an element of type `T`. When you call the function with different types of arrays, TypeScript infers the type parameter `T` based on the provided argument.

6.4 Why Do We Need Generics? (The Problem Generics Solve)

When working with functions that operate on different data types, you might find yourself writing multiple versions of the same function for each type. This leads to code duplication and makes maintenance harder.

Example: Repeated Function Declarations Without Generics

```
function getFirstNumber(arr: number[]): number {
  return arr[0];
}

function getFirstString(arr: string[]): string {
  return arr[0];
}
```

In this example, the logic is identical, but you need separate functions for each type. As the number of types increases, so does the repetition.

6.5 Solution: Using Generics

Generics allow you to write a single, reusable function that works with any data type, eliminating the need for repeated declarations.

Example: Solving with Generics

```
function getFirstElement<T>(arr: T[]): T {
  return arr[0];
}

const firstNum = getFirstElement([1, 2, 3]); // number
const firstStr = getFirstElement(["a", "b", "c"]); // string
const firstMixed = getFirstElement([1, "two", true]); // (number | string |
↪  boolean)
console.log(firstNum); // Outputs: 1
console.log(firstStr); // Outputs: a
console.log(firstMixed); // Outputs: 1
```

- With generics, you write the function once and TypeScript infers the type based on usage, making your code DRY (Don't Repeat Yourself) and type-safe.

6.6 Generics with the Types and Interfaces

6.6.1 The Problem: Repeated Type and Interface Declarations

Suppose you want to define a structure for a container that holds a value, but the value could be of any type (number, string, boolean, etc.). Without generics, you would need to create separate types or interfaces for each possible value type:

```
// Without generics: repeated declarations
interface NumberContainer {
  value: number;
}
```

```ts
interface StringContainer {
  value: string;
}

const numBox: NumberContainer = { value: 42 };
const strBox: StringContainer = { value: "Hello" };
```

This approach leads to code duplication and is not scalable as the number of types increases.

6.6.2 Solution: Using Generics with Types and Interfaces

Generics allow you to define a single, reusable type or interface that works with any value type:

```ts
// With generics: one reusable interface
interface Container<T> {
  value: T;
}

const numBox: Container<number> = { value: 42 };
const strBox: Container<string> = { value: "Hello" };
const boolBox: Container<boolean> = { value: true };
```

You can do the same with type aliases:

```ts
type Box<T> = {
  value: T;
};

const numberBox: Box<number> = { value: 100 };
const stringBox: Box<string> = { value: "TypeScript" };
```

With generics, you avoid repetition and gain flexibility, making your code more maintainable and type-safe.

> **Note:** Most React hooks, such as `useState` and `useEffect`, leverage generics to provide type safety and flexibility. For example, when using `useState`, you can specify the type of state to ensure type correctness:

```ts
import { useState } from "react";

// State with a number type
const [count, setCount] = useState<number>(0);

// State with a string type
const [name, setName] = useState<string>("");

// State with a custom type
type User = { id: number; username: string };
const [user, setUser] = useState<User | null>(null);
```

Generics in hooks allow TypeScript to infer and enforce the correct types for state values and updater functions, reducing runtime errors and improving developer experience.

---

7 TS in React and Generics Hooks

- TypeScript can be seamlessly integrated into React applications, providing type safety and improved developer experience.
- When using TypeScript with React, you can define types for props, state, and context, ensuring that your components are type-safe and reducing the likelihood of runtime errors.

7.1 Steps to create a React app with TypeScript

1. **Create a new React app with TypeScript** using vite:

```
npm create vite@latest my-app -- --template react-ts
cd my-app
npm install
```

- Explain this Command:
  - `npm create vite@latest my-app`: This command creates a new Vite project named `my-app`.
  - `-- --template react-ts`: This specifies that the project should use the React + TypeScript template.
  - `cd my-app`: Change into the newly created project directory.
  - `npm install`: Install the necessary dependencies for the project.

2. **Start the development server**:

```
npm run dev
```

- Use the traditional steps:

1. **Install TypeScript and React types**:

```
npm create vite
```

2. **Choose the React + TypeScript template** when prompted.
3. **Install dependencies**:

```
npm install
```

4. **Start the development server**:

```
npm run dev
```