TypeScript fundamentals

Authors: Goran Kukic

Sources: TypeScript Documentation (typescriptlang.org)

Table of contents

Hello fellow developers and learners, 👋

If you're looking to level up your JavaScript skills by diving into TypeScript, you're in the right place. TypeScript is a game-changer that brings static typing to JavaScript, making your code more predictable, scalable, and easier to debug.

This tutorial follows the official TypeScript documentation, ensuring you receive accurate and up-to-date information. To enrich the content further, I've seamlessly integrated my own invaluable insights, enhancing the overall learning experience. For more detailed information, you can check out the TypeScript documentation.

Access to the "TypeScript Fundamentals" tutorial is available on my public GitHub repository at:
https://github.com/GoranKukic/typescript-fundamentals
Also feel free to check my "JavaScript Fundamentals" tutorial on public GitHub repository at:
https://github.com/GoranKukic/javascript-fundamentals

What You'll Learn Here 📘

In this mini handbook, we'll cover the core concepts that make TypeScript unique and powerful:

- **Data Types**: Understanding string, number, boolean, and more.
- **Interfaces & Types**: Defining object structures and contracts.
- **Functions**: Type-safe function signatures and return types.
- **Generics**: Building reusable components with flexible types.
- **Modules**: Organizing and exporting code for better project structure.
- **Advanced Types**: Exploring union, intersection, and mapped types.

Each section will include easy-to-follow examples and practical tips to help you apply TypeScript in your projects right away.

Who Is This For? 🎯

- **JavaScript developers**: Looking to transition to TypeScript to enhance code reliability.
- **Beginner devs**: Who want to understand the advantages of typed systems.
- **Professionals**: Preparing for interviews or improving project maintainability.

Let's Get Started 🚀

By the end of this handbook, I hope you'll have a solid understanding of how to use TypeScript effectively in your day-to-day coding and why it's worth adding to your developer toolkit.

Happy coding with TypeScript! 🔥💻

Introduction

What is TypeScript?

TypeScript is a programming language based on JavaScript, offering everything JavaScript does but with additional features.

TypeScript was developed by Microsoft and first released in 2012, with Anders Hejlsberg, the creator of C#, leading the design of the language.

One of the key reasons to use TypeScript is that it introduces static typing to JavaScript. With static typing, once you define the type of a variable, it can't change throughout the program, which helps avoid many bugs!

In contrast, JavaScript is dynamically typed, meaning the type of a variable can change. Let's see an example:

```
// JavaScript Example
let greeting = "hello";
greeting = 42; // The variable type changes from string to number—JavaScript
                  is okay with this.
```

Now, compare that with TypeScript:

```
// TypeScript Example
let greeting: string = "hello";
greeting = 42; // Error: TypeScript doesn't allow changing from a string to
                  a number.
```

Since browsers can’t directly understand TypeScript, it needs to be converted (or compiled) into JavaScript using the TypeScript Compiler (TSC).

Is TypeScript Worth Using

Why You Might Choose TypeScript

- **Fewer Bugs**: Research suggests that TypeScript can help catch around 15% of common coding mistakes early.
- **Readability**: Code is easier to understand, especially when working in teams. It's clearer to see the intent behind variables and functions.
- **In-Demand Skill**: Learning TypeScript can open doors to more opportunities in the job market.
- **Deeper Understanding**: It also deepens your understanding of JavaScript by introducing concepts like types and interfaces

Potential Downsides

- **Takes More Time**: Writing in TypeScript can be more time-consuming, as you need to define variable types, which may not be ideal for smaller projects.
- **Compilation Overhead:** Since TypeScript has to be compiled into JavaScript, there's a time cost, especially with larger projects.

Despite these drawbacks, for medium to large applications, TypeScript can save time in the long run by preventing common errors and improving maintainability.

If you're already comfortable with JavaScript, transitioning to TypeScript won't be too difficult and will give you an additional tool for better coding.

How to Set Up a TypeScript Project

Install Node and TypeScript

Start by making sure you have Node.js installed on your computer. Then, install the TypeScript compiler globally using the following command:

```
npm install -g typescript
```

You can check if TypeScript is installed by running:

```
tsc -v
```

This will display the current version of TypeScript installed.

Compiling TypeScript Code

Create a new file in your text editor with the .ts extension (for example, app.ts), then write some TypeScript code like:

```
let sport = 'soccer';
let id = 7;
```

You can convert (compile) this TypeScript file into JavaScript using:

```
tsc app.ts
```

This will generate a file app.js with the following JavaScript code:

```
var sport = 'soccer';
var id = 7;
```

If you want to compile and specify a different output file:

```
tsc app.ts --outfile newfile.js
```

To have TypeScript automatically recompile your code each time you make changes, you can use the watch flag:

```
tsc app.ts -w
```

TypeScript's Flexibility with Errors

While TypeScript helps spot errors in your code as you write, it doesn't stop you from compiling the code—even if there are mistakes. For example:

```
let sport = 'soccer';
let id = 7;
id = 'seven'; // Error: Type 'string' is not assignable to type 'number'
```

Even though TypeScript will flag an error here, you can still compile the code if you wish:

```
tsc app.ts
```

Setting Up `tsconfig.json`

To configure TypeScript for your project, create a tsconfig.json file by running:

```
tsc --init
```

This file allows you to customize how TypeScript works in your project. Here's an example configuration:

```
{
       "compilerOptions": {
       "target": "ES2015", // Set the target to ES6 or ES2015
       "rootDir": "./src", // Source directory for the TypeScript files
       "outDir": "./dist", // Output directory for the compiled JavaScript
       "allowJs": true, // Allow JavaScript files to be compiled
       "sourceMap": true, // Generate source maps for easier debugging
       "removeComments": true // Strip out comments in the compiled JavaScript
},
"include": ["src"] // Only compile files in the "src" directory
}
```

After setting up your configuration file, you can compile the entire project and watch for file changes using:

```
tsc -w
```

Keep in mind, when you specify input files in the command (like tsc app.ts), the tsconfig.json is ignored.

Data Types in TypeScript

Primitive Types

In JavaScript, primitive values are the most basic types of data, such as numbers, strings, or booleans, which do not possess methods. There are seven primary primitive data types in JavaScript:

1. **string**
2. **number**
3. **bigint**
4. **boolean**
5. **undefined**
6. **null**
7. **symbol**

Primitive values are immutable, meaning they cannot be modified once created. However, it's possible to change the variable that holds a primitive by assigning it a new value, but the primitive itself remains unchanged.

For example:

```
let userName = 'Alex';
userName.toUpperCase();
console.log(userName); // Outputs: 'Alex' - the original string remains unchanged

let numbers = [2, 4, 6, 8];
numbers.shift();
console.log(numbers); // Outputs: [4, 6, 8] - the array is modified

userName = 'Sophie'; // Reassignment changes the variable's value
```

JavaScript provides object wrappers (`String`, `Number`, `BigInt`, `Boolean`, and `Symbol`) for primitive types (except for `null` and `undefined`), allowing them to use methods temporarily. For instance, in the example above, the `toUpperCase()` method is available because `userName` is wrapped in a `String` object.

TypeScript Usage

In TypeScript, you can specify the data type of a variable using a type annotation. This helps TypeScript to validate the data types used in your code. Here's how it looks:

```
      let id: number = 42;
let firstName: string = 'Alex';
let isEmployed: boolean = false;

let quantity: number; // Declaration without assignment
quantity = 100;
```

Most of the time, you don't need to explicitly define the type since TypeScript uses type inference:

```
      let age = 30; // Inferred as number
let city = 'Paris'; // Inferred as string
let hasPet = true; // Inferred as boolean

hasPet = 'yes'; // ERROR: Type 'string' is not assignable to type 'boolean'
```

Union Types

In TypeScript, you can use union types to specify that a variable can hold more than one type:

```
  let identifier: string | number;
  identifier = 123;
  identifier = 'ABC123';
```

Reference Types

In TypeScript, reference types refer to complex data structures that can store collections of values or more complex entities, such as objects and functions. Unlike primitive types, reference types can be modified even if they are assigned to a constant.

Here's a list of reference types in TypeScript:

1. **Arrays** (including Tuples)
2. **Objects**
3. **Functions**
4. **Class instances**

Reference types are different from primitive types because they store a reference (or a pointer) to the actual data in memory rather than the data itself. This means when you modify a reference type, the changes are reflected everywhere that reference is used.

Example: Primitive vs. Reference Types

To better understand the distinction, let's look at a simple example:

Primitive Type:

```
let score = 10;
let newScore = score;
newScore += 5;

console.log(score); // Outputs: 10 - original primitive value is unchanged
console.log(newScore); // Outputs: 15 - newScore is a separate copy
```

Reference Type:

```
let scores = [10, 20, 30];
let newScores = scores;
newScores.push(40);

console.log(scores); // Outputs: [10, 20, 30, 40] - both variables reference
                        the same array
console.log(newScores); // Outputs: [10, 20, 30, 40] - changes affect all
                           references
```

In the example above, the primitive value (`score`) remains unaffected when `newScore` is modified. However, the reference type (`scores`) is updated for both scores and `newScores` since they both point to the same array in memory.

Summary

Reference types in TypeScript provide more flexibility than primitive types, as they can store collections of values or more complex data. However, they also introduce the concept of "sharing" data, which means modifications to a reference type can impact other variables that refer to the same object. This is a key distinction between primitive and reference types in TypeScript.

Arrays

Arrays in TypeScript allow you to work with collections of values where you can define the type of elements that the array can contain ensuring type safety and consistency throughout your code.

Defining Arrays

You can specify the type of elements an array can hold using TypeScript's type annotations:

```
let numbers: number[] = [1, 2, 3, 4, 5]; // Restricted to numbers only
let names: string[] = ['Alice', 'Bob', 'Charlie']; // Restricted to strings only
let flags: boolean[] = [true, false, true]; // Restricted to boolean values only
let books: { title: string, author: string }[] = [
    { title: 'The Black Swan', author: 'Nassim Taleb' },
    { title: 'Homo Deus', author: 'Yuval Noah Harari' },
]; // This array is restricted to objects with specific properties
```

Arrays can also be declared with the any type, which bypasses TypeScript's type checking, effectively allowing any type of value:

```
let mixedArray: any[] = ['hello', 1, false]; // Can contain any type of value
```

However, using any can defeat the purpose of TypeScript's type safety, so it's generally better to specify the expected types whenever possible.

Type Safety in Arrays

TypeScript enforces type constraints on arrays. For example, attempting to push an incorrect type of value into an array will result in an error:

```
numbers.push(6);
numbers.push('seven'); // ERROR: Argument of type 'string' is not assignable to
                          parameter of type 'number'
```

To allow arrays to hold multiple types, you can use **union types**:

```
let mixedTypeArray: (string | number | boolean)[] = ['Alice', 30, true];
mixedTypeArray[0] = 50; // Allowed, as number is part of the union type
mixedTypeArray[1] = { title: 'New Book' }; // ERROR: Objects are not part of the
                                              allowed types
```

Type Inference

When you initialize an array with values, TypeScript can infer the array's type based on the provided values. This means you often don't need to explicitly declare the type, and it allows any of the specified types to be assigned to any index of the array:

```
let data = [1, 'text', false]; // Inferred type: (number | string | boolean)[]
data[1] = 2; // Allowed
data[2] = 'hello'; // Allowed
data[0] = { name: 'Bob' }; // ERROR: Type '{}' is not assignable to type 'number'
```

Tuples

A tuple is a specialized **type of array where the number and types of elements are fixed**. Tuples provide a way to work with arrays that have a predefined structure:

```
let personTuple: [string, number, boolean] = ['Alice', 30, true];
personTuple[0] = 'Bob'; // Allowed, as the first element must be a string
personTuple[1] = 40; // Allowed, as the second element must be a number
personTuple[2] = 'yes'; // ERROR: The third element must be a boolean
```

Tuples are useful when you need an array with a specific number of elements and types, and they help enforce consistency in your data structures.

Objects

In TypeScript, objects must have all the specified properties with the correct types:

```
// Define an object type for a person
let person: {
  name: string;
  location: string;
  isProgrammer: boolean;
};
// Assign values to the person object with all the necessary properties and types
person = {
  name: 'Alice',
  location: 'USA',
  isProgrammer: true,
};

person.isProgrammer = 'Yes'; // ERROR: Should be a boolean
```

If you try to assign a new object without all the properties or with incorrect types, TypeScript will throw an error:

```
person = {
  name: 'John',
  location: 'Canada',
};
// ERROR: Missing the 'isProgrammer' property
```

Optional Properties

In some cases, an object might have properties that aren't always required. Use `?` to mark them as optional:

```
interface Car {
  brand: string;
  model: string;
  year?: number; // Optional property
}
```

```
let myCar: Car = {
  brand: 'Toyota',
  model: 'Corolla',
}; // No error, 'year' is optional
```

Read-Only Properties

Use `readonly` to make a property immutable after the object is created:

```
interface Book {
  readonly title: string;
  author: string;
}

let myBook: Book = {
  title: '1984',
  author: 'George Orwell',
};

myBook.title = 'Animal Farm'; // ERROR: Cannot assign to 'title' because it is
                                  a read-only prope
```

Index Signatures

For objects with dynamic keys, use an index signature to define the types for all possible properties:

```
interface AddressBook {
  [name: string]: string; // The keys are strings, and the values are strings
}

let contacts: AddressBook = {
  Alice: '123-456-7890',
  Bob: '987-654-3210',
};

contacts['Charlie'] = '555-555-5555'; // Adding new properties dynamically
```

Using Interfaces for Objects

An interface in TypeScript is a way to define the shape or structure of an object. It acts like a blueprint that describes what properties and methods an object should have, along with its types.

Defining the structure of an object using an interface is useful when you want to enforce the same properties and types across multiple objects:

```
interface Person {
  name: string;
  location: string;
  isProgrammer: boolean;
}

let person1: Person = {
  name: 'Alice',
  location: 'USA',
  isProgrammer: true,
};

let person2: Person = {
  name: 'Bob',
  location: 'Germany',
  isProgrammer: false,
};
```

With this interface, you can ensure that any object assigned to a `Person` type has the `name`, `location`, and `isProgrammer` properties, all with the correct types.

Extending Interfaces

You can extend existing interfaces to create more complex structures:

```
interface Person {
  name: string;
  location: string;
}

interface Programmer extends Person {
  languages: string[];
}
```

```
let developer: Programmer = {
  name: 'Eve',
  location: 'Canada',
  languages: ['JavaScript', 'TypeScript'],
};
```

Type Aliases as an Alternative to Interfaces

Type aliases (type) can also define the shape of objects and are more flexible for combining different types:

```
type Animal = {
  name: string;
  age: number;
};

type Cat = Animal & {
  isIndoor: boolean;
};

let myCat: Cat = {
  name: 'Whiskers',
  age: 2,
  isIndoor: true,
};
```

Adding Function Properties in Interfaces

You can also define functions within interfaces, using either traditional function syntax or arrow functions:

```
interface Speech {
  sayHi(name: string): string;
  sayBye: (name: string) => string;
}

let greeter: Speech = {
  sayHi: function (name: string) {
    return `Hi ${name}`;
```

```
  },
  sayBye: (name: string) => `Bye ${name}`,
};

console.log(greeter.sayHi('Bob')); // Hi Bob
console.log(greeter.sayBye('Bob')); // Bye Bob
```

In the `greeter` object, `sayHi` and `sayBye` can use either traditional function syntax or arrow functions – TypeScript is flexible about how you define them as long as they match the signature in the `Speech` interface.

Optional Chaining

If you're working with nested objects and want to access a property safely, good practice is to use optional chaining (`?.`).

```
const user = { address: { street: '123 Main St' } };
console.log(user?.address?.street); // 123 Main St
```

Functions

In TypeScript, you can define the types of function arguments and the return type of the function. This helps catch errors during development and provides better code documentation.

```
// Function that takes a 'diam' parameter of type number, and returns a string

function circle(diam: number): string {
  return 'The circumference is ' + Math.PI * diam;
}

console.log(circle(10)); // The circumference is 31.41592653589793
```

Using Arrow Functions

The same function can be written using an ES6 arrow function:

```
const circle = (diam: number): string => {
  return 'The circumference is ' + Math.PI * diam;
};

console.log(circle(10)); // The circumference is 31.41592653589793
```

You don't always need to say that `circle` is a function; TypeScript can figure it out on its own. It also guesses the return type based on the code. But for bigger or more complex functions, it's helpful to specify the return type for clarity.

Explicit Typing

For cases where you prefer explicit typing for readability or documentation:

```
// Using explicit typing
const circle: Function = (diam: number): string => {
  return 'The circumference is ' + Math.PI * diam;
};

// Inferred typing - TypeScript sees that circle is a function returns a string,
                     so no need to explicitly state it
const circle = (diam: number) => {
  return 'The circumference is ' + Math.PI * diam;
};
```

Optional Parameters and Union Types

You can use a question mark (`?`) after a parameter to make it optional. Below, the `c` parameter is optional and can be either a number or a string (a union type):

```
const add = (a: number, b: number, c?: number | string) => {
  console.log(c); // Will log the value of 'c' if provided
  return a + b;
};

console.log(add(5, 4, 'Optional parameter here!')); // 9
console.log(add(5, 4)); // 9, since 'c' is optional
```

Default Parameters

You can also provide default values to function parameters, which will be used if no value is passed:

```
const multiply = (a: number, b: number = 2): number => {
  return a * b;
};

console.log(multiply(5)); // 10, uses default value for 'b'
console.log(multiply(5, 3)); // 15, overrides default value
```

Void Return Type

A function that doesn't return a value is said to return `void`. Although TypeScript infers this automatically, you can state it explicitly:

```
const logMessage = (msg: string): void => {
  console.log('Message: ' + msg);
};

logMessage('TypeScript is awesome!'); // Message: TypeScript is awesome!
```

Function Signatures

If you want to declare a variable to hold a function without defining it immediately, use a function signature. The variable must match the signature when a function is assigned to it:

```
// Declare the variable 'sayHello' with a function signature that takes a string
   and returns void
let sayHello: (name: string) => void;

// Define the function, satisfying the signature
sayHello = (name) => {
  console.log('Hello ' + name);
};

sayHello('Alice'); // Hello Alice
```

Rest Parameters

TypeScript allows you to use rest parameters to handle functions with a variable number of arguments:

```
const sumAll = (...numbers: number[]): number => {
  return numbers.reduce((acc, curr) => acc + curr, 0);
};

console.log(sumAll(1, 2, 3, 4)); // 10
console.log(sumAll(5, 10, 15)); // 30
```

Function Overloading

TypeScript supports function overloading, allowing you to define multiple function signatures for a single function:

```
function format(input: number): string;
function format(input: string): string;

function format(input: number | string): string {
  if (typeof input === 'number') {
    return `Number: ${input}`;
  } else {
    return `String: ${input}`;
  }
}

console.log(format(100)); // Number: 100
console.log(format('Hello')); // String: Hello
```

Callback Functions

Functions in TypeScript can also accept other functions as arguments, known as callback functions:

```
const greet = (name: string, callback: (message: string) => void) => {
  const greeting = `Hello, ${name}!`;
  callback(greeting);
};
```

```
greet('Alice', (message) => {
  console.log(message); // Hello, Alice!
});
```

Classes

Classes in TypeScript allow us to define a blueprint for creating objects with specific properties and methods. We can also define the types for each property, ensuring type safety throughout our code.

Basic Class Structure

Here's how you can define a simple class with properties and a method in TypeScript:

```
class Person {
   name: string;
   isCool: boolean;
   pets: number;

   constructor(n: string, c: boolean, p: number) {
      this.name = n;
      this.isCool = c;
      this.pets = p;
   }

   sayHello(): string {
      return `Hi, my name is ${this.name} and I have ${this.pets} pets.`;
   }
}
```

To create an object using this class, we use the `new` keyword. This process is called **instantiation**, where we create a new **instance** of the `Person` class:

```
const person1 = new Person('Danny', false, 1);
const person2 = new Person('Sarah', true, 6);
```

In this example, `person1` and `person2` are both instances of the `Person` class. When we use the `new` keyword, the `constructor` method is called to initialize the properties of each instance with the values provided.

```
console.log(person1.sayHello()); // Output: Hi, my name is Danny and I have 1 pets
```

In this example, the `Person` class has three properties: `name`, `isCool`, and `pets`. The constructor initializes these properties, ensuring that they match the types specified. The method `sayHello` returns a string describing the person.

Type Safety in Classes

TypeScript enforces type safety. If you try to pass an incorrect type to a parameter, you'll get an error:

```
const person3 = new Person('Mike', 'yes', 4);
// Error: Argument of type 'string' not assignable to parameter of type 'boolean'
```

TypeScript prevents you from assigning values of the wrong type, reducing the chance of runtime errors.

Using Arrays of Class Instances

We can create an array that only holds instances of the `Person` class:

```
let people: Person[] = [person1, person2];
```

Here, the `people` array is strictly typed to only contain objects constructed from the `Person` class.

Access Modifiers in TypeScript

TypeScript allows you to control the accessibility of class properties and methods using access modifiers:

- `public`: The property or method can be accessed from anywhere (default if not specified).
- `private`: The property or method can only be accessed within the class itself.
- `protected`: The property or method can be accessed within the class and its subclasses.
- `readonly`: The property can only be read; it cannot be modified after being assigned in the constructor.

Here's how access modifiers work:

```
class Person {
  readonly name: string; // Can only be read, not modified
  private isCool: boolean; // Accessible only within this class
  protected email: string; // Accessible within this class and its subclasses
  public pets: number; // Accessible from anywhere

  constructor(n: string, c: boolean, e: string, p: number) {
    this.name = n;
    this.isCool = c;
    this.email = e;
    this.pets = p;
  }

  sayMyName(): void {
    console.log(`You're not Heisenberg, you're ${this.name}`);
  }
}

const person1 = new Person('Danny', false, 'dan@e.com', 1);
console.log(person1.name); // Output: Danny
person1.name = 'James'; // Error: Cannot assign to 'name' because it is a
                           read-only property
```

In the above example:

- `name` is marked as `readonly`, so it can only be read, not modified.
- `isCool` is private, so it cannot be accessed outside of the `Person` class.
- `email` is protected, meaning it's accessible in `Person` and any class that extends `Person`.
- `pets` is `public` and can be accessed and modified from anywhere.

Concise Property Declaration in Constructor

TypeScript allows you define and assign properties in the constructor, reducing code duplication:

```
class Person {
  constructor(
    readonly name: string,
    private isCool: boolean,
    protected email: string,
    public pets: number
  ) {}

  sayMyName(): void {
    console.log(`You're not Heisenberg, you're ${this.name}`);
  }
}

const person1 = new Person('Danny', false, 'dan@e.com', 1);
console.log(person1.name); // Output: Danny
```

By declaring properties directly in the constructor, we streamline the class definition and eliminate the need to manually assign values to properties.

Extending Classes

Classes can be extended using the `extends` keyword, allowing you to create new classes based on existing ones. The new class inherits all the properties and methods of the base class:

```
class Programmer extends Person {
  programmingLanguages: string[];

  constructor(
    name: string,
    isCool: boolean,
    email: string,
    pets: number,
    pL: string[]
  ) {
    super(name, isCool, email, pets); // Calls the constructor of the base class
    this.programmingLanguages = pL;
  }

  listLanguages(): void {
    console.log(`${this.name} knows: ${this.programmingLanguages.join(', ')}`);
  }
```

```
}

const programmer1 = new Programmer('Sarah', true, 'sarah@code.com', 3,
   ['JavaScript', 'TypeScript']);
programmer1.listLanguages(); // Output: Sarah knows: JavaScript, TypeScript
```

Here, the `Programmer` class extends the `Person` class, adding a new property `programmingLanguages` and a method `listLanguages`. The `super` call in the constructor is necessary to initialize the properties inherited from the `Person` class.

Getters and Setters

You can add getters and setters to your class to control how properties are accessed and modified:

```
class Person {
  private _age: number;

  constructor(public name: string, age: number) {
    this._age = age;
  }

  get age(): number {
    return this._age;
  }

  set age(newAge: number) {
    if (newAge >= 0) {
      this._age = newAge;
    } else {
      console.log('Age must be a positive number.');
    }
  }
}

const person1 = new Person('John', 30);
console.log(person1.age); // Output: 30
person1.age = -5; // Output: Age must be a positive number.
```

In this example, age is controlled using getter and setter methods, allowing validation before updating the property.

Summary

- Classes in TypeScript enable structured object creation with defined property types.
- Access modifiers (`public private`, `protected`, `readonly`) control access to properties.
- Constructors can be used to initialize properties directly, making the code more concise.
- Classes can be extended using `extends` to create new classes based on existing ones.
- Getters and setters provide more control over how properties are accessed and modified.

This covers the essentials of classes in TypeScript and shows how you can use them to define structured, type-safe objects.

Dynamic (any) Types

Using the any type in TypeScript allows you to bypass type checking and essentially revert to JavaScript's dynamic typing:

```
let age: any = '100';
age = 100;
age = {
  years: 100,
  months: 2,
};
```

While any can be helpful in certain situations, it's generally best to avoid using it. Relying on any can lead to bugs, as it disables TypeScript's ability to catch errors. It’s better to be specific about the types you want to work with.

Type Aliases

Type aliases can simplify your code and reduce repetition, helping you adhere to the DRY (Don't Repeat Yourself) principle. They act as reusable definitions for complex types:

```
type StringOrNumber = string | number;

type PersonObject = {
  name: string;
  id: StringOrNumber;
};
```

```
const person1: PersonObject = {
  name: 'John',
  id: 1,
};

const person2: PersonObject = {
  name: 'Delia',
  id: 2,
};

const sayHello = (person: PersonObject) => {
  return 'Hi ' + person.name;
};

const sayGoodbye = (person: PersonObject) => {
  return 'Seeya ' + person.name;
};
```

Here, `PersonObject` is a type alias that defines what properties a person object should have, making it easy to use and maintain.

The DOM and Type Casting

TypeScript doesn't have direct access to the DOM like JavaScript does, so it can't always be sure that an element exists. For example:

```
const link = document.querySelector('a');
console.log(link.href); // ERROR: Object is possibly 'null'.
```

TypeScript throws an error because it can't be certain that the a tag exists. You can use the non-null assertion operator (`!`) to tell TypeScript that you're sure the element is not `null` or `undefined`:

```
const link = document.querySelector('a')!;
console.log(link.href); // www.example.com
```

TypeScript automatically infers that link is of type `HTMLAnchorElement`, so you don't need to manually specify the type.

Selecting Elements by Class or ID

When selecting an element using `getElementById`, TypeScript isn't sure what type of element is:

```
const form = document.getElementById('signup-form');
console.log(form.method);
// ERROR: Object is possibly 'null'.
// ERROR: Property 'method' does not exist on type 'HTMLElement'.
```

Here, TypeScript complains because `form` might be `null` and because it doesn't know that form is an `HTMLFormElement`. You can use type casting to resolve this:

```
const form = document.getElementById('signup-form') as HTMLFormElement;
console.log(form.method); // post
Now, TypeScript knows that form exists and is of type HTMLFormElement.
```

Using the Event Object

TypeScript also provides built-in support for event objects. When you add an event listener, TypeScript can enforce the correct event properties and methods:

```
const form = document.getElementById('signup-form') as HTMLFormElement;

form.addEventListener('submit', (e: Event) => {
  e.preventDefault(); // Prevents the page from refreshing
  console.log(e.tarrget); // ERROR: Property 'tarrget' does not exist on type
                             'Event'. Did you mean 'target'?
});
```

Here, TypeScript catches the typo (tarrget instead of target) and suggests the correct property. This feature helps prevent bugs related to incorrect property names.

Advanced Type Handling in TypeScript

Interfaces

In TypeScript, interfaces define the structure of objects, specifying what properties and types those properties should have. Let's look at a basic example:

```
interface User {
  username: string;
  age: number;
}

function greetUser(user: User) {
  console.log(`Hello, ${user.username}`);
}

greetUser({
  username: 'Alice',
  age: 30,
}); // Hello, Alice
```

In this example, the `User` interface ensures that any object passed into `greetUser` has a `username` and age property.

You can also define object types using **type aliases**, which is an alternative way to describe object structures:

```
type User = {
  username: string;
  age: number;
};

function greetUser(user: User) {
  console.log(`Hello, ${user.username}`);
}
greetUser({
  username: 'Alice',
  age: 30,
}); // Hello, Alice
```

Additionally, you can define object types **inline** without creating a separate interface or type alias:

```
function greetUser(user: { username: string; age: number }) {
  console.log(`Hello, ${user.username}`);
}

greetUser({
  username: 'Alice',
  age: 30,
}); // Hello, Alice
```

Interfaces vs Type Aliases

While interfaces and type aliases are often interchangeable, there are important distinctions. A key difference is that interfaces can be extended or reopened to add more properties, whereas type aliases are fixed once declared.

Extending an Interface:

```
interface Animal {
  name: string;
}

interface Dog extends Animal {
  breed: string;
}

const myDog: Dog = {
  name: "Buddy",
  breed: "Golden Retriever",
};
```

In this example, the `Dog` interface extends the `Animal` interface, inheriting its properties and adding a new one (`breed`).

Extending a Type Alias with Intersections:

```
type Animal = {
  name: string;
};

type Dog = Animal & {
  breed: string;
};

const myDog: Dog = {
  name: "Buddy",
  breed: "Golden Retriever",
};
```

Here, we use an intersection (`&`) to combine two types into one.

Reopening Interfaces

One powerful feature of interfaces is that you can "reopen" them to add new properties:

```
interface Animal {
  name: string;
}

// Adding a new property to the existing Animal interface
interface Animal {
  legs: number;
}

const cat: Animal = {
  name: "Whiskers",
  legs: 4,
};
```

This flexibility allows you to extend existing interfaces in different parts of your code.

Type Aliases Cannot Be Reopened

Unlike interfaces, **type aliases** cannot be modified once declared:

```
type Animal = {
  name: string;
};

// Trying to redefine the same type will result in an error
type Animal = {
  legs: number;
};
// ERROR: Duplicate identifier 'Animal'.
```

For this reason, it's often recommended to use interfaces when defining object types, especially if you anticipate the need to extend them.

Function Signatures with Interfaces

Interfaces can also describe functions and their expected parameters:

```
interface Person {
  name: string;
  age: number;
  speak(phrase: string): void;
}

const person: Person = {
  name: "Alice",
  age: 30,
  speak: (phrase) => console.log(phrase),
};

person.speak("Hello, world!"); // Output: Hello, world!
```

In this example, the `Person` interface includes a method `speak`, which expects a string as an argument.

Interfaces vs Classes

You might wonder when to use an interface instead of a class. While both can define object structures, an interface only exists in TypeScript and does not affect the compiled JavaScript. It's

purely for type-checking. On the other hand, classes are part of both TypeScript and JavaScript, and they get transpiled into JavaScript code.

- **Interface**: Used purely for type-checking. Does not exist in the final JavaScript output.
- **Class**: A blueprint for creating objects that exist in both TypeScript and JavaScript, and include properties, methods, and initialization logic.

Here's an example of a class:

```
class Person {
  name: string;
  age: number;

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

  speak(phrase: string): void {
    console.log(`${this.name} says: ${phrase}`);
  }
}

const person1 = new Person("Bob", 25);
person1.speak("Good morning!"); // Output: Bob says: Good morning!
```

In this example, the class `Person` defines both the structure and the behavior (methods) of the objects we create from it.

Summary

- **Interfaces** define object structures in TypeScript and allow for flexible, reusable types.
- **Type aliases** can also define object types but cannot be reopened like interfaces.
- When deciding between the two, prefer interfaces for objects and extendability, and use types for other use cases like unions or intersections.
- Interfaces can define both properties and function signatures, making them versatile tools for type-checking in TypeScript.

Implementing Interfaces in Classes

In TypeScript, you can use interfaces to enforce that a class contains specific properties and methods. This helps ensure that classes follow a consistent structure, especially when multiple classes need to adhere to the same blueprint.

For example, let's define an interface called `CanFormat`, which requires a `format` method:

```
interface CanFormat {
  format(): string;
}

class User implements CanFormat {
  constructor(public firstName: string, private secretCode: string) {}

  format(): string {
    return this.firstName.toUpperCase();
  }
}

// The User class now adheres to the CanFormat interface
let user1: CanFormat;
let user2: CanFormat;

user1 = new User('Alice', 'secret123');
user2 = new User('Bob', 'superSecret789');

console.log(user1.format()); // ALICE
```

In this example, the `User` class implements the `CanFormat` interface, ensuring that each instance of `User` will have a `format` method. The `format` method in this case returns the `firstName` property in uppercase.

Enforcing Interface Compliance in Arrays

You can also ensure that an array only contains objects that follow a specific interface. For instance, let's store multiple `User` objects, all of which implement the `CanFormat` interface:

```
let users: CanFormat[] = [];

users.push(user1);
users.push(user2);

users.forEach(user => {
  console.log(user.format()); // Outputs: ALICE, BOB
});
```

By specifying the type `CanFormat[]`, we ensure that all elements of the `users` array are objects that have the `format` method, as required by the interface.

Why Use Interfaces with Classes?

Interfaces offer a flexible way to define contracts between different parts of your code. This ensures that any class implementing an interface will conform to a certain structure. In contrast to classes, interfaces are only used during development and are not compiled into the final JavaScript, making them useful purely for type-checking without adding overhead to your code.

In our example:

- The `User` class have a `format` method because it implements the `CanFormat` interface.
- This provides consistency across any class that adheres to this contract.

Summary

Using interfaces with classes helps create a clear and structured way to enforce consistent behavior across multiple objects in your TypeScript projects. This is especially useful when working with larger codebases where consistency and type safety are key.

Generics

Generics in TypeScript allow us to create reusable components that can work with various data types instead of being limited to a specific one. This approach ensures that components or functions are more flexible and type-safe. Let’s dive into how generics work and why they're so useful.

Example: A Function that Adds an ID

Consider the following `addID` function. It accepts an object and returns a new object with all the original properties, plus a random `id` property.

```
const addID = (obj: object) => {
  let id = Math.floor(Math.random() * 1000);
  return { ...obj, id };
};

let person1 = addID({ name: 'Alice', age: 30 });
console.log(person1.id); // Random ID
console.log(person1.name); // ERROR: Property 'name' does not exist on
                              type '{ id: number; }'
```

In this example, TypeScript throws an error when we try to access the `name` property. This is because TypeScript isn't aware of what properties the input object has. The only thing TypeScript knows about the returned object is that it contains an `id`.

Using Generics for Flexibility

To fix this, we can use generics. A generic allows us to "capture" the type of the object passed into the function, so TypeScript knows about its properties.

```
const addID = <T>(obj: T) => {
  let id = Math.floor(Math.random() * 1000);
  return { ...obj, id };
};

let person1 = addID({ name: 'Alice', age: 30 });
console.log(person1.id); // Random ID
console.log(person1.name); // Alice
```

Now, by using `<T>`, we've told TypeScript to capture the type of the object passed in. `T` will represent the type of whatever object we pass, so the return type will have both the original properties and the `id`.

Constraints with Generics

While using generics, we might run into issues where anything can be passed into the function. For instance, if we pass a string instead of an object, TypeScript won't throw an error until we try to access object properties that don't exist.

```
let invalidInput = addID('Hello'); // No error at this point
console.log(invalidInput.id); // Random ID
console.log(invalidInput.name); // ERROR: Property 'name' does not exist on type
                                    '"Hello" & { id: number; }'
```

To prevent this, we can constrain our generic type to ensure that only objects are allowed:

```
const addID = <T extends object>(obj: T) => {
  let id = Math.floor(Math.random() * 1000);
  return { ...obj, id };
};
let person1 = addID({ name: 'Alice', age: 30 });
let invalidInput = addID('Hello'); // ERROR: Argument of type 'string' is not
                                      assignable to parameter of type 'object'
```

This ensures that only objects can be passed into the `addID` function.

Further Restricting Types

We can further narrow down the type constraint by specifying that the input object must have a specific property, such as `name`.

```
const addID = <T extends { name: string }>(obj: T) => {
  let id = Math.floor(Math.random() * 1000);
  return { ...obj, id };
};

let person1 = addID({ name: 'Alice', age: 30 }); // Valid
let invalidInput = addID({ age: 30 }); // ERROR: Argument must have a 'name'
                                          property
```

Generics with Interfaces

Generics also work well with interfaces. Imagine we have an interface `Person` but the type of one of its properties is unknown. We can use generics to specify the type later.

```
interface Person<T> {
  name: string;
  age: number;
  documents: T;
}

// Use the Person interface, specifying that documents is an array of strings
const person1: Person<string[]> = {
  name: 'John',
  age: 48,
  documents: ['passport', 'bank statement'],
};

// Use the Person interface, specifying that documents is a single string
const person2: Person<string> = {
  name: 'Delia',
  age: 46,
  documents: 'passport',
};
```

In this example, the `documents` property can take on different types, depending on how we define it when we implement the `Person` interface.

Generic Functions with Multiple Types

Generics can also be used to create functions that operate over multiple types. This is especially useful when the relationship between the input and output types needs to be maintained. Here's an example:

```
const mergeObjects = <T, U>(obj1: T, obj2: U) => {
  return { ...obj1, ...obj2 };
};

let merged = mergeObjects({ name: 'Alice' }, { age: 30 });
console.log(merged); // { name: 'Alice', age: 30 }
```

In this case, `mergeObjects` takes two objects with potentially different types, and returns a new object that combines the properties of both.

Type Safety with Generics

One of the main benefits of using generics is maintaining type safety. For instance, when using the any type, TypeScript won't enforce type checks:

```
function logLength(value: any) {
  console.log(value.length); // No error, but may cause runtime issues
}

logLength(123); // Undefined length property
```

Using generics with constraints, we can restrict the function to values that have a `length` property:

```
interface HasLength {
  length: number;
}

function logLength<T extends HasLength>(value: T) {
  console.log(value.length);
}

logLength('Hello'); // 5
logLength([1, 2, 3]); // 3
logLength(123); // ERROR: Type 'number' does not have a 'length' property
```

Summary

Generics in TypeScript allow for:

- Flexibility and type safety in reusable components.
- Working with a variety of types while still ensuring TypeScript's type-checking mechanisms.
- Creating relationships between input and output types for more complex functions.
- Using constraints to narrow down the accepted types for added safety.

By utilizing generics, you can write code that is reusable, scalable, and type-safe.

Literal Types in TypeScript

In TypeScript, literal types allow us to **specify exact values for variables**, rather than just general types like `string`, `number`, or `boolean`. This provides greater type safety by restricting the values a variable can hold. Literal types can be either string literals, numeric literals, or even boolean literals.

String Literal Types

A common use case is defining a variable that can only take on a few predefined string values. For example, when dealing with a limited set of options like color choices, we can define a union of string literal types:

```
let favouriteColor: 'red' | 'blue' | 'green' | 'yellow';

favouriteColor = 'blue'; // OK
favouriteColor = 'crimson'; // ERROR: Type '"crimson"' is not assignable to type
                                '"red" | "blue" | "green" | "yellow"'
```

Here, the variable `favouriteColor` is restricted to four possible values: `'red'`, `'blue'`, `'green'`, or 'yellow'. Any assignment outside these values will trigger a TypeScript error.

Numeric Literal Types

Similarly, we can use numeric literals to enforce specific number values. This can be useful in scenarios where only certain numeric values are valid, such as in configurations or settings:

```
let diceRoll: 1 | 2 | 3 | 4 | 5;

diceRoll = 4; // OK
diceRoll = 6; // ERROR: Type '6' is not assignable to type '1 | 2 | 3 | 4 | 5'
```

In this example, the `diceRoll` variable is restricted to the numbers 1 through 5. Assigning any number outside this range results in a type error.

Boolean Literal Types

You can also define boolean literals, though their use is less common since a boolean can naturally only be `true` or `false`:

```
let isComplete: true | false;

isComplete = true;  // OK
isComplete = false; // OK
isComplete = 'yes'; // ERROR: Type '"yes"' is not assignable to type 'true | false'
```

This provides strict control over the boolean state, especially in cases where the status must be a defined binary value.

Combining Literal Types with Other Types

Literal types are not limited to basic values; you can combine them with other types for more complex scenarios. For instance, combining **literal types with union types** allows for broader flexibility while still restricting the range of values.

```
type Shape = 'circle' | 'square' | 'rectangle';
type Size = 'small' | 'medium' | 'large';

let shape: Shape;
let size: Size;

shape = 'circle';  // OK
size = 'medium';   // OK
shape = 'triangle'; // ERROR: Type '"triangle"' is not assignable to type 'Shape'.
```

Use Cases for Literal Types

1. Enums: While TypeScript has an enum feature, literal types can sometimes provide a more lightweight alternative, especially when the enum values are simple strings or numbers.

```
type Direction = 'north' | 'south' | 'east' | 'west';
let travelDirection: Direction = 'north';
```

2. Function Parameters: Literal types are often used to restrict the input values for function parameters, ensuring that only specific values are passed to the function.

```
function move(direction: 'left' | 'right' | 'up' | 'down') {
  console.log(`Moving ${direction}`);
}

move('left'); // OK
move('forward'); // ERROR: Argument of type '"forward"' is not assignable to
                    parameter of type '"left" | "right" | "up" | "down"'
```

3. Configuration Options: Literal types are useful for defining configuration options or settings, where only certain predefined values are allowed.

```
type Mode = 'dark' | 'light' | 'auto';

function setMode(mode: Mode) {
  console.log(`Setting mode to ${mode}`);
}

setMode('dark'); // OK
setMode('blue'); // ERROR: Type '"blue"' is not assignable to type 'Mode'.
```

Literal Types with Type Aliases

To make your code cleaner, you can use type aliases with literal types, which gives a name to your literal union:

```
type Color = 'red' | 'green' | 'blue';
type Status = 'active' | 'inactive' | 'pending';

let currentColor: Color;
let userStatus: Status;
```

```
currentColor = 'red';    // OK
userStatus = 'active';   // OK
userStatus = 'deleted';  // ERROR: Type '"deleted"' is not assignable to type
                            'Status'.
```

By defining `Color` and `Status` as aliases, you can reuse these types in multiple places, making your code more maintainable and readable.

Literal Inference with `const`

When you use the `const` keyword to declare a variable, TypeScript automatically infers the literal type of that variable. This prevents it from being reassigned to a different value later.

```
const buttonState = 'pressed';

buttonState = 'released'; // ERROR: Type '"released"' is not assignable to type
                             '"pressed"'.
```

Because `buttonState` was declared as a constant with the value 'pressed', it cannot be changed to any other string value later in the code.

Summary

Literal types in TypeScript allow you to constrain variables to specific, predefined values, enhancing type safety and preventing invalid assignments. Whether you're working with string, number, or boolean literals, they provide a useful mechanism for enforcing stricter type checking. These types are commonly used in situations where only a limited set of values is valid, such as in enums, configuration options, or function parameters.

TypeScript Code Safety and Optimization

Strict Mode

One of TypeScript's most powerful features is its strict type-checking capabilities, which can help catch potential bugs early in the development process. By enabling strict mode, you instruct TypeScript to perform more comprehensive checks, ensuring that your code adheres to safer practices.

To enable strict mode, you need to modify your `tsconfig.json` file and set the `"strict"` option to true. This activates a set of type-checking features that can drastically improve your code quality.

```
// tsconfig.json
{
  "compilerOptions": {
     "strict": true
   }
}
```

Enabling this option might result in more errors, but it helps you write more robust, bug-free code.

No Implicit any

By default, TypeScript attempts to infer types wherever it can. However, sometimes it doesn't have enough information to do so and falls back to the any type. This can be dangerous because any disables type safety, meaning TypeScript will not check what you're doing with that variable.

Consider this function:

```
function logName(a) {
  // No error here??
  console.log(a.name);
}

logName(97);
```

Here, the parameter `a` is implicitly typed as `any`, which means TypeScript will not complain if we pass in something unexpected like a number. If `a` is a number, trying to access the `name` property will result in `undefined`, but no error will be raised. This can cause unpredictable behavior.

When "`noImplicitAny`" is enabled through strict mode, TypeScript will require you to explicitly define the type of `a`:

```
// Error: Parameter 'a' implicitly has an 'any' type.
function logName(a: any) {
  console.log(a.name);
}
```

You can fix this by explicitly stating the type or using a more appropriate one:

```
function logName(a: { name: string }) {
  console.log(a.name);
}

logName({ name: 'John' }); // Works fine
```

This change forces you to think about the shape and type of the data your function will handle, improving type safety.

Strict Null Checks

In JavaScript, `null` and `undefined` are common sources of runtime errors. Without strict null checks, TypeScript treats `null` and `undefined` as acceptable values for any type, leading to possible issues when accessing properties or methods on values that don't exist.

Here's an example:

```
let whoSangThis: string = getSong();

const singles = [
  { song: 'touch of grey', artist: 'grateful dead' },
  { song: 'paint it black', artist: 'rolling stones' },
```

```
];

const single = singles.find((s) => s.song === whoSangThis);

console.log(single.artist); // What if `single` is undefined?
```

In this code, `singles.find` might not find a match, which would return `undefined`. However, we're assuming that it always finds something and directly access the `artist` property, which could lead to a runtime error.

With `"strictNullChecks": true`, TypeScript won't let you make such assumptions. You'll get an error if you try to access properties on a value that could potentially be `null` or `undefined`.

```
// ERROR: Object is possibly 'undefined'.
console.log(single?.artist);
```

To fix this, we need to add a check to ensure that `single` is defined before accessing its properties:

```
if (single) {
  console.log(single.artist); // Now safe
}
```

This forces you to handle cases where `null` or `undefined` might sneak in, making your code more resilient to unexpected failures.

Extra Features in Strict Mode

In addition to the two major checks we've discussed, strict mode enables a few more useful features that improve type safety across your code:

1. Strict Function Types

   Strict function types ensure that functions are checked more rigorously when passed as arguments to other functions. TypeScript will enforce that function parameters and return types align correctly, preventing common mismatches.

```
function callWithNumber(callback: (n: number) => void) {
  callback(42);
}

callWithNumber((x) => console.log(x.toFixed(2))); // Good
callWithNumber((x) => console.log(x.length)); // Error: 'number' has no
                                                 'length' property
```

2. Always Strict Mode

When strict mode is enabled, TypeScript always operates in strict JavaScript mode. This means it will perform additional runtime checks and behave in a more predictable manner in terms of scoping and reserved words.

3. Strict Property Initialization

This feature ensures that all properties of a class are initialized when an instance is created. This can prevent issues where certain properties are accidentally left undefined due to forgotten assignments.

```
class User {
  name: string;
  age: number; // Error: Property 'age' has no initializer and is not
                  definitely assigned in the constructor

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

This error can be resolved by either initializing age or marking it as optional with `?`:

```
class User {
  name: string;
  age?: number;

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

Summary

Strict mode in TypeScript helps enforce stricter type-checking rules, which leads to fewer bugs and more maintainable code. By catching potential issues early, such as unhandled null values or missing property initializations, you can write cleaner, safer, and more reliable code.

To summarize, enabling strict mode enables:

- **No implicit any**: Requires explicit typing and prevents unsafe code from creeping in.
- **Strict null checks**: Forces you to handle `null` and `undefined` explicitly, making your code more resilient.
- **Strict function types**: Ensures function arguments and return types are always correctly typed.
- **Strict property initialization**: Enforces class property initialization to prevent `undefined` values.

Strict mode may seem like it generates more errors, but those errors are helping you spot potential issues before they arise in production. It's a highly recommended setting for any TypeScript project.

Narrowing

In TypeScript, it's common to work with variables that can hold multiple types, such as `string | number` or custom union types. **Type narrowing** is the process of refining the type of a variable from a broader type (like a union) to a more specific type based on runtime checks.

Narrowing is essential because it allows TypeScript to understand which specific type a variable holds at a given point in the code, enabling you to perform operations safely and take advantage of TypeScript's powerful type system.

Basic Narrowing with `typeof`

A common way to narrow types in TypeScript is by using the `typeof` operator. This is useful when dealing with primitive types like `string`, `number`, and `boolean`. Here's a simple example:

```
function repeatValue(val: string | number): string {
  if (typeof val === 'string') {
    // Since TypeScript knows `val` is a string here, we can call string methods
    return val.repeat(2);
  }

  // Otherwise, `val` must be a number, so we handle that case
  return (val * 2).toString();
}

console.log(repeatValue('Hello')); // "HelloHello"
console.log(repeatValue(10));      // "20"
```

In this example, TypeScript uses `typeof` to narrow the `val` type from `string | number` to either `string` or `number` within the corresponding `if` and `else` blocks, making it safe to use type-specific methods.

Narrowing with `in` Operator

The `in` operator is useful when you're working with objects that share some properties but differ in others. Here's an example where we have two types, `Car` and `Bicycle`, and we use the `in` operator to narrow the type:

```
interface Car {
  type: 'Car';
  horsepower: number;
  doors: number;
}

interface Bicycle {
  type: 'Bicycle';
  gears: number;
}

type Vehicle = Car | Bicycle;

function describeVehicle(vehicle: Vehicle): string {
  if ('horsepower' in vehicle) {
   // TypeScript now knows `vehicle` is of type `Car`
   return `Car with ${vehicle.horsepower} horsepower and ${vehicle.doors} doors.`;
  }
  // Otherwise, it's a `Bicycle`
  return `Bicycle with ${vehicle.gears} gears.`;
}
```

```
const car: Car = { type: 'Car', horsepower: 150, doors: 4 };
const bike: Bicycle = { type: 'Bicycle', gears: 21 };
console.log(describeVehicle(car));  // "Car with 150 horsepower and 4 doors."
console.log(describeVehicle(bike)); // "Bicycle with 21 gears."
```

Here, `in` checks for the presence of a property (`horsepower`) that only exists on the `Car` type, allowing TypeScript to narrow `vehicle` from `Vehicle` (a union type) to `Car`.

Literal Type Narrowing with Discriminated Unions

Discriminated unions use a common literal property to distinguish between types. This is a more structured approach to narrowing, often used in situations where types need to share a key but hold different values:

```
interface Dog {
  kind: 'Dog';
  barkVolume: number;
}

interface Cat {
  kind: 'Cat';
  meowVolume: number;
}

type Pet = Dog | Cat;

function petNoise(pet: Pet): string {
  if (pet.kind === 'Dog') {
    // TypeScript narrows `pet` to `Dog` type here
    return `The dog barks at volume: ${pet.barkVolume}`;
  }

  // TypeScript automatically narrows `pet` to `Cat` type here
  return `The cat meows at volume: ${pet.meowVolume}`;
}

const dog: Dog = { kind: 'Dog', barkVolume: 80 };
const cat: Cat = { kind: 'Cat', meowVolume: 50 };

console.log(petNoise(dog)); // "The dog barks at volume: 80"
console.log(petNoise(cat)); // "The cat meows at volume: 50"
```

The property `kind` is a literal value (`'Dog'` or `'Cat'`), which helps TypeScript to differentiate between the types and allows the compiler to infer the correct type within the `if` block.

Narrowing with `instanceof`

When working with classes or custom objects, you can use the `instanceof` operator to narrow down the type of an object. This works when you need to check if an object is an instance of a specific class:

```
class Fish {
  swim() {
    console.log('Fish is swimming');
  }
}

class Bird {
  fly() {
    console.log('Bird is flying');
  }
}

type Animal = Fish | Bird;

function moveAnimal(animal: Animal) {
  if (animal instanceof Fish) {
    // TypeScript knows `animal` is a Fish
    animal.swim();
  } else {
    // TypeScript knows `animal` is a Bird
    animal.fly();
  }
}

const goldfish = new Fish();
const parrot = new Bird();

moveAnimal(goldfish); // Fish is swimming
moveAnimal(parrot);   // Bird is flying
```

In this example, `instanceof` helps TypeScript recognize whether the `animal` is a `Fish` or `Bird`, allowing the appropriate method (`swim` or `fly`) to be safely invoked.

Custom Type Guards

For more complex type narrowing scenarios, you can create your own **custom type guards**. These are functions that help TypeScript understand what type a variable holds by returning a boolean value and using the `is` keyword to specify the type:

```
interface Dog {
  bark: () => void;
}

interface Cat {
  meow: () => void;
}

function isDog(pet: Dog | Cat): pet is Dog {
  return (pet as Dog).bark !== undefined;
}

function petSound(pet: Dog | Cat) {
  if (isDog(pet)) {
    pet.bark(); // TypeScript now knows pet is a Dog
  } else {
    pet.meow(); // TypeScript now knows pet is a Cat
  }
}

const dog: Dog = { bark: () => console.log('Woof!') };
const cat: Cat = { meow: () => console.log('Meow!') };

petSound(dog); // "Woof!"
petSound(cat); // "Meow!"
```

In this example, the custom type guard `isDog` helps TypeScript determine whether the `pet` is a `Dog`. Inside the `if` block, TypeScript narrows the type and allows us to use `Dog`-specific methods.

Exhaustiveness Checking with `never`

When working with union types, it's a good idea to ensure all possible cases are handled. You can use the `never` type in combination with `switch` statements to enforce exhaustiveness:

```
interface Square {
  kind: 'square';
  size: number;
}

interface Circle {
  kind: 'circle';
  radius: number;
}

type Shape = Square | Circle;

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case 'square':
      return shape.size * shape.size;
    case 'circle':
      return Math.PI * shape.radius * shape.radius;
    default:
      const _exhaustiveCheck: never = shape;
      throw new Error(`Unknown shape: ${_exhaustiveCheck}`);
  }
}

const square: Square = { kind: 'square', size: 10 };
const circle: Circle = { kind: 'circle', radius: 5 };

console.log(getArea(square)); // 100
console.log(getArea(circle)); // 78.5398...
```

The never type in the default case helps ensure that all possible cases in the Shape union are handled. If a new shape is added but not handled in the switch, TypeScript will raise an error during compilation, helping you avoid runtime issues.

Modules and Enums in TypeScript

Modules in TypeScript

Modules are a key concept in JavaScript and TypeScript for structuring and organizing code. A module is essentially a file containing related pieces of code (e.g., functions, variables, or classes) that can be exported and imported into other modules or files.

In TypeScript, this concept works similarly to how it does in JavaScript. TypeScript files get compiled into multiple JavaScript files that can be linked together. This modular approach improves maintainability and clarity in larger applications.

Setting Up Your `tsconfig.json`

Before you begin working with modules in TypeScript, you need to configure your `tsconfig.json` file to enable modern import/export functionality. Here's how you can do that:

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "es2015"
  }
}
```

Alternatively, for Node.js projects, it's common to set `"module": "CommonJS"`, since Node.js does not fully support ES modules yet.

Using Modules in HTML

When using modules in the browser, you'll need to adjust the `<script>` tag in your HTML file. Instead of the typical script tag, you'll add a `type="module"` attribute to indicate that the script uses ES module syntax.

```
<script type="module" src="/public/script.js"></script>
```

Importing and Exporting Modules

With this configuration in place, you can start using `import` and `export` syntax to share code between different TypeScript files. For example, if you have a function in one file and want to use it in another, you can export it like this:

```
// src/hello.ts
export function sayHello() {
  console.log('Hello, world!');
}
```

Now, in another file, you can import and use the `sayHello` function:

```
// src/main.ts
import { sayHello } from './hello.js';

sayHello(); // Output: Hello, world!
```

Note: Even when working with TypeScript files, when importing, always use the `.js` extension.

Named Exports vs Default Exports

There are two main types of exports in TypeScript: named exports and default exports. In the previous example, we used a named export. Here's a quick overview of both:

Named Exports: You can export multiple variables, functions, or classes from a file and import them using curly braces.

```
// utils.ts
export const PI = 3.14159;
export function calculateArea(radius: number) {
  return PI * radius * radius;
}
// main.ts
import { PI, calculateArea } from './utils.js';

console.log(PI); // 3.14159
console.log(calculateArea(5)); // 78.53975
```

Default Exports: You can export a single value as the default from a module, which can be imported without curly braces.

```
// math.ts
export default function add(a: number, b: number): number {
  return a + b;
}

// main.ts
import add from './math.js';

console.log(add(2, 3)); // 5
```

Organizing Code with Modules

Modules help organize your application code into different concerns. For example, in a large project, you could split code into modules like `user.ts`, `product.ts`, and `order.ts`. Each module can handle specific parts of the functionality.

Creating a Utility Module

Imagine you have a set of utility functions that can be used across your project. You can organize them into a `utils.ts` file.

```
// utils.ts
export function formatDate(date: Date): string {
  return date.toISOString().split('T')[0];
}

export function capitalize(str: string): string {
  return str.charAt(0).toUpperCase() + str.slice(1);
}
```

You can then import these functions wherever needed:

```
// main.ts
import { formatDate, capitalize } from './utils.js';
```

```
console.log(formatDate(new Date())); // 2024-09-23
console.log(capitalize('hello')); // Hello
```

Re-Exporting

TypeScript also allows re-exporting, which means you can import something and immediately export it from a module. This can be useful when you want to group multiple exports into a single module:

```
// moduleA.ts
export function foo() {
  return 'foo';
}

// moduleB.ts
export function bar() {
  return 'bar';
}

// index.ts (Re-export both)
export { foo } from './moduleA.js';
export { bar } from './moduleB.js';
```

Now, you can import `foo` and `bar` from `index.ts`:

```
import { foo, bar } from './index.js';

console.log(foo()); // foo
console.log(bar()); // bar
```

Module Resolution

TypeScript uses several strategies to resolve modules. The most common ones are:

1. **Relative Imports**: If the file is in the same directory or a nested one, you use `./` or `../` in the import path.

2. **Non-Relative Imports**: Used for modules installed via npm (e.g., `import * as express from 'express'`).
3. **Path Mapping**: You can define custom module paths in your `tsconfig.json` using `paths` and `baseUrl`.

Example of path mapping in `tsconfig.json`:

```
{
  "compilerOptions": {
    "baseUrl": "./",
    "paths": {
      "@utils/*": ["src/utils/*"]
    }
  }
}
```

Now you can import from `src/utils/` like this:

```
import { formatDate } from '@utils/date.js';
```

Dynamic Imports

Another cool feature TypeScript inherits from JavaScript is dynamic imports. Sometimes you may want to load a module only when it's needed, rather than at the start of your script. This can be achieved using the `import()` function:

```
// main.ts
async function loadModule() {
  const { sayHello } = await import('./hello.js');
  sayHello();
}

loadModule();
```

This approach can be useful for optimizing large applications by lazy loading code when it's required.

Summary

Modules are a powerful way to organize your code and make it reusable. By using TypeScript's module system, you ensure that your code remains clean, maintainable, and efficient.

- Always configure `tsconfig.json` for proper module resolution.
- Use `export` and `import` syntax to share functionality across files.
- Named and default exports help structure the way you expose functions or variables.
- Dynamic imports provide flexibility in loading code when needed.

With these tools, you can manage large codebases and ensure that every piece of your application works seamlessly together.

Enums in TypeScript

Enums are a special feature of TypeScript that allow developers to define a set of named constants, which can be either numeric or string values. Enums provide a way to organize related values under meaningful names, making the code more readable and reducing potential bugs. Let's dive into how they work and how they can enhance your TypeScript projects.

Basic Numeric Enums

The simplest form of an enum is a numeric enum, where the values are automatically assigned incremental numbers, starting from `0` unless otherwise specified. Here's an example:

```
enum Status {
  PENDING,
  IN_PROGRESS,
  COMPLETED,
  CANCELLED
}
console.log(Status.PENDING);      // 0
console.log(Status.COMPLETED);    // 2
console.log(Status.CANCELLED);    // 3
```

In this example, TypeScript automatically assigns the value `0` to `PENDING`, `1` to `IN_PROGRESS`, and so on. This automatic incrementing behavior can simplify code where sequential numeric values are meaningful.

Customizing Numeric Enums

You can customize the starting value of an enum by explicitly setting the value of the first element. The following elements will continue incrementing from the initial value:

```
enum OrderStatus {
  NEW = 1,
  PROCESSING,
  SHIPPED,
  DELIVERED
}

console.log(OrderStatus.NEW);       // 1
console.log(OrderStatus.PROCESSING);// 2
console.log(OrderStatus.DELIVERED); // 4
```

Here, `NEW` starts at `1`, and the subsequent values increment automatically. If you need specific values for each constant, you can manually assign them too:

```
enum ResponseCode {
  SUCCESS = 200,
  UNAUTHORIZED = 401,
  NOT_FOUND = 404,
  SERVER_ERROR = 500
}

console.log(ResponseCode.SUCCESS);      // 200
console.log(ResponseCode.SERVER_ERROR); // 500
```

This is useful when you want to directly match enums to predefined numeric values, such as HTTP status codes or error codes.

String Enums

In addition to numeric enums, TypeScript also supports string-based enums. These are useful when the enum values represent more descriptive, non-numeric values:

```
enum Direction {
  North = 'NORTH',
  South = 'SOUTH',
  East = 'EAST',
  West = 'WEST'
}

console.log(Direction.North); // "NORTH"
console.log(Direction.West);  // "WEST"
```

String enums provide more meaningful output and are commonly used when you need more readable and user-friendly constants. Unlike numeric enums, string enums don't increment values automatically, so you need to assign each value explicitly.

Enum Properties and Reverse Mappings

TypeScript's numeric enums also have a reverse mapping feature. This means that you can get the name of an enum member by its value:

```
enum Color {
  Red = 1,
  Green,
  Blue
}

console.log(Color.Red);    // 1
console.log(Color[2]);     // "Green"
```

In this example, accessing `Color[2]` returns the name `Green`, thanks to TypeScript's reverse mapping for numeric enums. Note that this feature is only available for numeric enums, not for string enums.

Use Cases for Enums

Enums are particularly helpful when you have a set of related values that don't change often, as:

- **Statuses**: For tracking different states in a workflow, like `PENDING`, `COMPLETED`, or `FAILED`.
- **Directions**: Like `NORTH`, `SOUTH`, `EAST`, `WEST` in a navigation system.

- **Error Codes**: Matching error codes to meaningful constants.
- **Roles or Permissions**: Like `ADMIN`, `USER`, `GUEST` to represent different access levels.

Enums can also make your code easier to maintain and prevent mistakes, because tools like IntelliSense will provide you with the possible options as soon as you start typing.

```
enum Role {
  Admin = 'ADMIN',
  User = 'USER',
  Guest = 'GUEST'
}

function checkAccess(role: Role) {
  if (role === Role.Admin) {
    console.log('Full access granted.');
  } else if (role === Role.User) {
    console.log('Limited access.');
  } else {
    console.log('No access.');
  }
}

checkAccess(Role.Admin); // Full access granted.
```

Constant Enums

In cases where you want to avoid the overhead of generating an object at runtime, TypeScript provides constant enums. These are completely removed during the compilation phase, and the values are inlined into your code.

```
const enum Days {
  Monday,
  Tuesday,
  Wednesday
}

let day: Days = Days.Monday;
console.log(day); // 0
```

Using `const enum` improves performance by reducing the footprint of your final JavaScript output. However, you lose features like reverse mapping, since the enum doesn't exist as an object in the compiled JavaScript.

Heterogeneous Enums

Though less common, TypeScript allows heterogeneous enums, where you mix string and numeric values:

```
enum Result {
  Success = 1,
  Failure = 'FAIL'
}

console.log(Result.Success); // 1
console.log(Result.Failure); // "FAIL"
```

This type of enum can be helpful in very specific scenarios where some values must be numbers and others need to be descriptive strings, but it's generally best to stick to either all string or all numeric values for consistency and readability.

Enum as Flags

Enums can also be used to represent **bitwise flags**. This is especially useful when you need to store multiple related options in a single value. Here's how it works:

```
enum Permissions {
  Read = 1,       // 0001
  Write = 2,      // 0010
  Execute = 4     // 0100
}
let userPermissions = Permissions.Read | Permissions.Write;

function hasPermission(perm: Permissions, checkPerm: Permissions): boolean {
  return (perm & checkPerm) === checkPerm;
}

console.log(hasPermission(userPermissions, Permissions.Read));   // true
console.log(hasPermission(userPermissions, Permissions.Execute));// false
```

In this case, we use the bitwise OR (|) operator to combine permissions and the bitwise AND (&) operator to check if a specific permission exists.

Summary

Enums in TypeScript are a powerful feature that can help make your code more readable, maintainable, and less prone to errors. Whether you're working with numeric values, descriptive strings, or using enums as bitwise flags, they give you a clean and concise way to manage sets of related constants.

With their versatility and the ability to easily integrate into your project, enums are a great addition to your TypeScript toolbox. Don't forget to use them whenever you need a fixed set of options or states!

Bonus: TypeScript with React

TypeScript is a powerful tool for bringing type safety and developer productivity to JavaScript codebases. It works seamlessly with JSX, and by including the type definitions for React (`@types/react`) and ReactDOM (`@types/react-dom`), you get full support for React in a TypeScript environment.

Adding TypeScript to an Existing React Project

To install the type definitions for React and ReactDOM, run the following command in your terminal:

```
npm install @types/react @types/react-dom
```

Then, in your `tsconfig.json`, make sure you have the following options:

- Include "`dom`" in the "`lib`" array to ensure DOM support (this is typically the default behavior).
- Set the `jsx` option to either "`react-jsx`" or "`react`" depending on your React version. If you're working with a library, check the React documentation for the appropriate option.

TypeScript in React Components

When working with React and TypeScript, every file containing JSX should use the `.tsx` extension. This tells TypeScript that the file includes JSX syntax, enabling type-checking for React components.

A key feature is the ability to define types for a component's props. This ensures correctness and provides better documentation in your editor. Let's modify the `MyButton` component example by adding type annotations for its `title` prop:

```
function MyButton({ title }: { title: string }) {
  return <button>{title}</button>;
}

export default function App() {
```

```
  return (
    <div>
      <h1>Hello, React with TypeScript!</h1>
      <MyButton title="Click me!" />
    </div>
  );
}
```

This example shows how you can type props inline. However, for more complex props or to improve readability, you can define types using `interface` or `type`. For a more in-depth explanation and examples, please refer to the Working with React Props and TypeScript lesson.

Working with React Props and TypeScript

One of the core features of React is passing props to components, and TypeScript allows you to explicitly define the types of those props. This ensures the values passed into components are of the expected types, avoiding common bugs. You can define the shape of the props using either interface or type. Both approaches achieve the same outcome, but each has slight differences and personal preference plays a role in which to choose.

Using `interface` for Props

An `interface` is a natural way to define an object structure in TypeScript. It's often used for larger applications because interfaces are extendable, meaning you can add to them later or combine them with others.

```
interface ButtonProps {
  label: string;
  disabled?: boolean; // Optional prop
}

function MyButton({ label, disabled = false }: ButtonProps) {
  return <button disabled={disabled}>{label}</button>;
}
```

In the above example:

- The `ButtonProps` interface defines the shape of the props.
- `label` is a required `string`, and `disabled` is an optional `boolean`.

- `MyButton` component receives `label` and `disabled`, defaulting `disabled` to `false` if not provided.

Using `type` for Props

A `type` is another way to define the structure of an object, including props. Unlike `interface`, `type` is often used for more specific cases like unions or primitives but can also be used for props.

```
type MyComponentProps = {
  name: string;
  age?: number; // Optional prop
};

const MyComponent: React.FC<MyComponentProps> = ({ name, age }) => (
  <div>
    <p>Name: {name}</p>
    {age && <p>Age: {age}</p>}
  </div>
);
```

In this example:

- The `MyComponentProps` type defines the props structure.
- `name` is a required `string`, while `age` is an optional `number`.
- `MyComponent` displays the `name` and conditionally renders the `age` if provided.

When to Use `interface` vs. `type`

- `interface` is preferred when you need to extend or merge definitions. It's more commonly used in large-scale applications.
- `type` is more flexible for complex types like unions and intersections but can also be used for defining props.

Both approaches are valid, so it's up to you to choose the what fits your use case or coding style.

Typing React Hooks

TypeScript works out of the box with React Hooks. Here are some examples of how to use TypeScript with common hooks like `useState`, `useReducer`, and `useContext`.

useState

When you use the `useState` hook, TypeScript will infer the type from the initial value. However, you can also explicitly define the state type:

```
const [count, setCount] = useState<number>(0); // Setting the type to number
```

For union types, such as status states:

```
type Status = "idle" | "loading" | "success" | "error";
const [status, setStatus] = useState<Status>("idle");
```

useReducer

The `useReducer` hook allows for more complex state management. You can type the reducer's state and actions:

```
interface State {
  count: number;
}

type Action = { type: "increment" } | { type: "decrement" };

function reducer(state: State, action: Action): State {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      throw new Error("Unknown action");
  }
}

const [state, dispatch] = useReducer(reducer, { count: 0 });
```

useContext

The `useContext` hook lets you share values across components without explicitly passing props down the tree:

```
const ThemeContext = createContext<"light" | "dark">("light");

const MyComponent = () => {
  const theme = useContext(ThemeContext);
  return <div>Current theme: {theme}</div>;
};
```

If the context has no initial value or needs null handling, make sure to manage the null state explicitly:

```
const MyContext = createContext<string | null>(null);

const MyComponent = () => {
  const value = useContext(MyContext);
  if (!value) throw new Error("MyComponent must be within MyContext.Provider");
  return <div>{value}</div>;
};
```

useMemo

The `useMemo` Hook helps optimize performance by caching the result of a function and recalculating it only when its dependencies change. In TypeScript, you can either rely on TypeScript's inference or be more explicit with a type argument.

```
interface Todo {
  id: number;
  task: string;
  completed: boolean;
}

const visibleTodos = useMemo<Todo[]>(() => filterTodos(todos, tab), [todos, tab]);

function filterTodos(todos: Todo[], tab: string): Todo[] {
  // Filter logic here...
}
```

Here, `visibleTodos` is of type `Todo[]`, inferred from the `filterTodos` function. We can also provide the return type directly as a type argument to `useMemo`.

useCallback

`useCallback` returns a stable reference to a function and only re-creates it when dependencies change. It's especially useful for performance optimization in child components or event handlers.

```
const handleClick = useCallback(() => {
  // Function logic...
}, [todos]);

// When strict mode is on, or for explicit typing:
const handleChange =
   useCallback<React.ChangeEventHandler<HTMLInputElement>>((event) => {
      setValue(event.currentTarget.value);
}, [setValue]);
```

Here, `useCallback` is used to memoize the `handleChange` function, and `React.ChangeEventHandler<HTMLInputElement>` is added for explicit typing of the event handler.

useRef

The `useRef` Hook returns a mutable object that persists for the lifetime of the component. You can specify the type of the DOM element it refers to using TypeScript.

```
interface InputProps {
  placeholder: string;
}

const Person = ({ placeholder }: InputProps) => {
  const inputRef = useRef<HTMLInputElement>(null);

  return (
    <div>
      <input type="text" ref={inputRef} placeholder={placeholder} />
    </div>
  );
};
```

In this example, we use `useRef` to refer to an `HTMLInputElement`.

Useful Types

The `@types/react` package provides a wide variety of types that make working with React and TypeScript more efficient. Here, we'll cover some of the most common types you'll encounter when working with React components and event handlers.

DOM Events

When handling DOM events in React, the event type is often inferred by TypeScript from the event handler. However, when passing a handler to a component or refactoring, it's best to explicitly set the event type.

Example:

```
import { useState } from 'react';

export default function Form() {
  const [value, setValue] = useState("Change me");

  function handleChange(event: React.ChangeEvent<HTMLInputElement>) {
    setValue(event.currentTarget.value);
  }

  return (
    <>
      <input value={value} onChange={handleChange} />
      <p>Value: {value}</p>
    </>
  );
}
```

There are many types of DOM events provided by React. The full list can be found in the `@types/react` package, but common examples include:

- `React.MouseEvent`
- `React.KeyboardEvent`
- `React.ChangeEvent`

If you need a generic event type, `React.SyntheticEvent` serves as a base type for all React events.

Children

When defining a component's children in TypeScript, there are two common types you can use to specify what can be passed as children.

1. Using `React.ReactNode`: This is a broad type that can represent anything that can be rendered in JSX (elements, strings, numbers, etc.).

```
interface ModalRendererProps {
  title: string;
  children: React.ReactNode;
}
```

2. Using `React.ReactElement`: This type restricts children to only JSX elements and excludes primitives like strings or numbers.

```
interface ModalRendererProps {
  title: string;
  children: React.ReactElement;
}
```

TypeScript cannot enforce that children must be of a specific JSX type (e.g., only allowing `<li>` elements). This means you cannot specify that a component only accepts certain types of JSX elements through TypeScript's type system.

Style Props

When passing inline styles in React, use the `React.CSSProperties` type to ensure your styles are valid CSS properties and benefit from editor auto-completion.

Example:

```
interface MyComponentProps {
  style: React.CSSProperties;
}

const MyComponent = ({ style }: MyComponentProps) => {
  return <div style={style}>Styled Component</div>;
};
```

Event Types

To determine the correct event type, you can hover over the event handler, and TypeScript will provide the expected event type. If you need a more generic handler or your event isn't included, fall back on `React.SyntheticEvent`.

Summary

Using TypeScript in a React project brings safety, improved developer experience, and self-documenting code through type annotations. Whether you're managing props, hooks, or context, TypeScript ensures that your React code remains clean and free of common runtime errors. With the flexibility to describe component props and states in detail, TypeScript becomes an invaluable tool for building robust and scalable React applications.

For more advanced use cases, consider exploring union types, generics, and custom hooks with TypeScript.