# 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. This feature ensures that your arrays maintain 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 their 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!
});
```

# 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.