

The image features the TypeScript logo, which consists of a dark red arrow pointing to the right. The word "TypeScript" is written in white, sans-serif font inside the arrow. The background is a light green color with several thin, curved lines in a slightly darker shade of green, creating a modern and abstract design.

TypeScript



What is typescript

- TS is an enhanced version of JavaScript.
- We Could say superset of javascript.
- You can understand Like c++ is an enhanced version of c language.
- TypeScript is building on JavaScript.
- TS is not a new Language. this is just add some new features in JS.



Benefits of ts

- But TS can not run on the browser directly
- So what is the benefit of it? if we can not run it on browser!
- TS compiler covert the code from TypeScript to Javascript
- Check Error on Compile Time
- Add type like string, number, bool etc
- We can add Object-Oriented Way with TS
- Code is well Managed

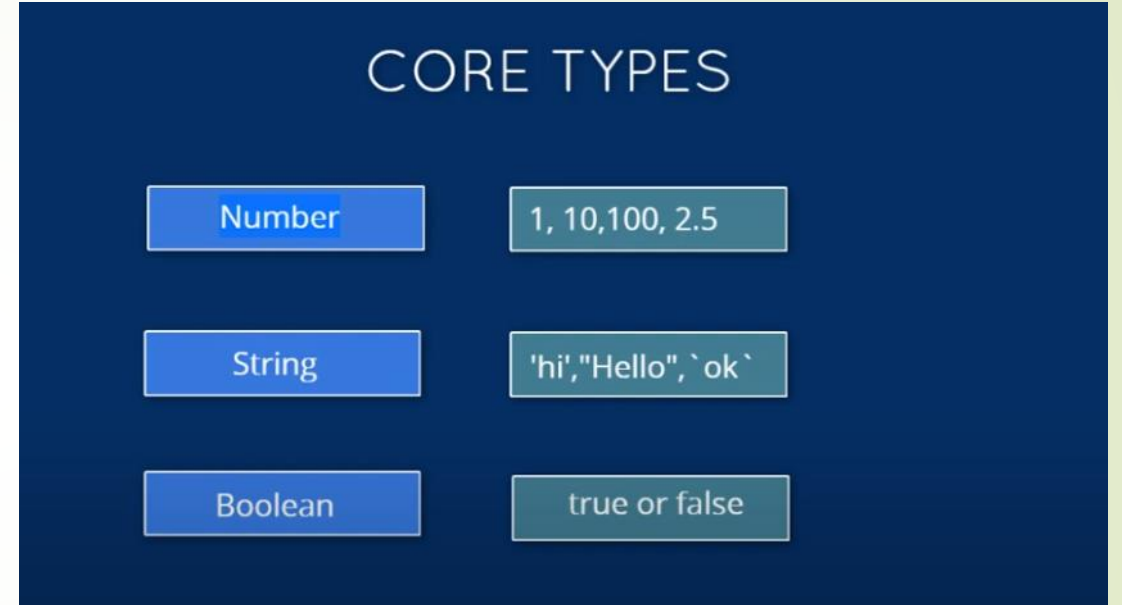



VERSION AND HISTORY

- Developed By Microsoft
- Current Version: 4.2.2
- First Released: 1-oct-2012
- Influenced by: JavaScript ‘

Core types

- There are three main primitives in JavaScript and TypeScript.
- boolean - true or false values
- number - whole numbers and floating point values
- string - text values like "TypeScript Rocks"





Data Type - Number

Number Methods

Method	Description
<u>toExponential()</u>	Returns the exponential notation in string format.
<u>toFixed()</u>	Returns the fixed-point notation in string format.
<u>toLocaleString()</u>	Converts the number into a local specific representation of the number.
<u>toPrecision()</u>	Returns the string representation in exponential or fixed-point to the specified precision.
<u>toString()</u>	Returns the string representation of the number in the specified base.
<u>valueOf()</u>	Returns the primitive value of the number.

Data Type - Number

- `let myNumber: number = 123456;`
-
- `myNumber.toExponential();`
`// returns 1.23456e+5`
- `myNumber.toExponential(1);`
`//returns 1.2e+5`
- `let myNumber: number = 10.8788;`
- `myNumber.toFixed(); // returns 11`
- `myNumber.toFixed(1); //returns 10.9`
- `let myNumber: number = 10667.987;`
-
- `myNumber.toLocaleString(); // returns`
`10,667.987 - US English`
- `myNumber.toLocaleString('de-DE'); //`
- `let myNumber: number = 10.5679;`
-
- `myNumber.toPrecision(1); // returns 1e+1`
- `myNumber.toPrecision(2); // returns 11`
- `let myNumber: number = 123;`
- `myNumber.toString(); // returns '123'`
- `myNumber.toString(2); // returns '1111011'`
- `let num1 = new Number(123);`
- `console.log(num1) //Output: a number`
`object with value 123`
- `console.log(num1.valueOf()) //Output: 123`



String



Method	Description
<u>charAt()</u>	Returns the character at the given index
<u>concat()</u>	Returns a combination of the two or more specified strings
<u>indexOf()</u>	Returns an index of first occurrence of the specified substring from a string (-1 if not found)
<u>replace()</u>	Replaces the matched substring with a new substring
<u>split()</u>	Splits the string into substrings and returns an array
<u>toUpperCase()</u>	Converts all the characters of the string into upper case



literals



- In TypeScript, literals are values that represent themselves in code. TypeScript provides several different types of literals that can be used to define constants or narrow down the types of variables.
- Types of literals discussed here : // side...
- Literals can be used to define constants or to narrow down the types of variables in TypeScript. For example, you can use a string literal to define a constant string value, or a boolean literal to narrow down the type of a variable to boolean.
- Numeric Literals: Numeric literals are used to represent numbers in TypeScript. Numeric literals can be written in decimal, hexadecimal, binary, or octal format.
- String Literals: String literals are used to represent strings in TypeScript. String literals can be enclosed in single or double quotes.
- Boolean Literals: Boolean literals are used to represent true or false values in TypeScript.
- Array Literals: Array literals are used to represent arrays in TypeScript. Array literals are enclosed in square brackets and can contain any combination of literals and expressions.
- Object Literals: Object literals are used to represent objects in TypeScript. Object literals are enclosed in curly braces and consist of key-value pairs.
- Null and Undefined Literals: Null and undefined literals are used to represent the absence of a value in TypeScript.

TypeScript Special Types

- Type: any
- any is a type that disables type checking and effectively allows all types to be used.
- Example with any :-
- ```
let u = true;
u = "string"; // Error: Type 'string' is not assignable to type 'boolean'.
Math.round(u); // Error: Argument of type 'boolean' is not assignable to parameter of type 'number'.
```
- Example with any :-
- ```
let v: any = true;  
v = "string"; // no error as it can be "any" type  
Math.round(v); // no error as it can be "any" type
```

Arrays

- `let fruits: string[] = ['Apple', 'Orange', 'Banana'];`
- `let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];`
- `let arr = [1, 3, 'Apple', 'Orange', 'Banana', true, false];`
-

Method	Description
<code>pop()</code>	Removes the last element of the array and return that element
<code>push()</code>	Adds new elements to the array and returns the new array length
<code>sort()</code>	Sorts all the elements of the array
<code>concat()</code>	Joins two arrays and returns the combined result



tuple



- a tuple is a specific type of array that has a fixed number of elements, where each element can have a different type. Unlike regular arrays, where all elements have the same type, tuples allow you to define a fixed number of elements with different types.
- Ex: `let myTuple: [string, number] = ['hello', 123];`
- `console.log(myTuple[0]); // Output: 'hello'`
- `console.log(myTuple[1]); // Output: 123`
- Tuples can be useful in situations where you need to store a fixed number of values of different types, such as coordinates, dates, or other structured data. They can also be used to enforce type safety in functions that accept a fixed number of arguments with different types.
- It's worth noting that while tuples can be useful in some situations, they can also be less flexible than regular arrays, since the number of elements and their types are fixed at the time of declaration. Therefore, you should consider whether a tuple or a regular array is more appropriate for your specific use case.



enum

Enums or enumerations are a new data type supported in TypeScript. Most object-oriented languages like Java and C# use enums. This is now available in TypeScript too.

In simple words, enums allow us to declare a set of named constants i.e. a collection of related values that can be numeric or string values.

There are three types of enums:

- Numeric enum
- String enum
- Heterogeneous enum

enum Type in typescript

- A Group Of constant
- That can assign a number to your string and make an easy comparison.
- `enum Role {values}`



Numeric Enum

- ▶ Numeric enums are number-based enums i.e. they store string values as numbers.
- ▶ Enums are always assigned numeric values when they are stored. The first value always takes the numeric value of 0, while the other values in the enum are incremented by 1.
- ▶ Ex:
enum PrintMedia {
- ▶ Newspaper = 1,
- ▶ Newsletter = 5,
- ▶ Magazine = 5,
- ▶ Book = 10
- ▶ }



String Enum

- String enums are similar to numeric enums, except that the enum values are initialized with string values rather than numeric values.
- The benefits of using string enums is that string enums offer better readability. If we were to debug a program, it is easier to read string values rather than numeric values.
- Ex:

```
enum PrintMedia {  
    Newspaper = "NEWSPAPER",  
    Newsletter = "NEWSLETTER",  
    Magazine = "MAGAZINE",  
    Book = "BOOK"  
}  
  
// Access String Enum  
PrintMedia.Newspaper; //returns NEWSPAPER  
PrintMedia['Magazine'];//returns MAGAZINE
```



Heterogeneous Enum

- Heterogeneous enums are enums that contain both string and numeric values.
 - Ex:
enum Status {
 - Active = 'ACTIVE',
 - Deactivate = 1,
 - Pending
 - }



union

- ▶ TypeScript allows us to use more than one data type for a variable or a function parameter. This is called union type.
- ▶ Ex:
`let code: (string | number);`
- ▶ `code = 123; // OK`
- ▶ `code = "ABC"; // OK`
- ▶ `code = false; // Compiler Error`

Union Type in typescript

Union type describes a value that can be one of several types



void

- ▶ Similar to languages like Java, void is used where there is no data. For example, if a function does not return any value then you can specify void as return type.
- ▶ Ex:
function sayHi(): void {
- ▶ console.log('Hi!')
- ▶ }
- ▶
- ▶ let speech: void = sayHi();
- ▶ console.log(speech); //Output: undefined
- ▶ There is no meaning to assign void to a variable, as only null or undefined is assignable to void.
- ▶ let nothing: void = undefined;
- ▶ let num: void = 1; // Error



Type Assertion

In TypeScript, type assertion is a way to tell the compiler the type of a value when the compiler cannot infer it automatically. It allows you to treat an expression as if it has a different type than its original type.

Type assertion can be done in two ways:

1. Using the angle-bracket syntax (**<Type>**):

```
Ex: let myVariable: any = "Hello World";  
let myString: string(<string>myVariable);
```

In this example, **myVariable** is of type **any**, which means the compiler doesn't know its type. By using angle-bracket syntax with **<string>**, we are telling the compiler that we know the type of **myVariable** is a string.

2. Using the as keyword:

```
Ex: let myVariable: any = "Hello World";  
let myString: string = (myVariable as string);
```



Functions

- functions can be of two types: named and anonymous.
- Named Functions
- A named function is one where you declare and call a function by its given name.
- Ex:
function Sum(x: number, y: number) : number {
➤ return x + y;
➤ }
➤
➤ Sum(2,3); // returns 5



Anonymous Function

➤ An anonymous function is one which is defined as an expression. This expression is stored in a variable. So, the function itself does not have a name. These functions are invoked using the variable name that the function is stored in.

➤ Ex:

```
let greeting = function() {  
    console.log("Hello TypeScript!");  
};
```

```
greeting(); //Output: Hello TypeScript!
```

Function Parameters

- Ex:
function Greet(greeting: string, name: string) : string {
- return greeting + ' ' + name + '!';
- }
-
- Greet('Hello','Steve');//OK, returns "Hello Steve!"
- Greet('Hi'); // Compiler Error: Expected 2 arguments, but got 1.
- Greet('Hi','Bill','Gates');//Compiler Error: Expected 2 arguments, but got 3.
- This is unlike JavaScript, where it is acceptable to pass less arguments than what the function expects. The parameters that don't receive a value from the user are considered as undefined.



Optional Parameters

- TypeScript has an optional parameter functionality. The parameters that may or may not receive a value can be appended with a '?' to mark them as optional.
- Ex:
function Greet(greeting: string, name?: string) : string {
- return greeting + ' ' + name + '!';
- }
-
- Greet('Hello','Steve');//OK, returns "Hello Steve!"
- Greet('Hi'); // OK, returns "Hi undefined!".
- Greet('Hi','Bill','Gates');//Compiler Error: Expected 2 arguments, but got 3.



Default Parameters

- `function Greet(name: string, greeting: string = "Hello") : string {`
- `return greeting + ' ' + name + '!';`
- `}`
-
- `Greet('Steve');//OK, returns "Hello Steve!"`
- `Greet('Steve', 'Hi');// OK, returns "Hi Steve!".`
- `Greet('Bill');//OK, returns "Hello Bill!"`
- Here if we passed more than require argument it was ignored that argument.

Function Overloading

- TypeScript provides the concept of function overloading. You can have multiple functions with the same name but different parameter types and return type. However, the number of parameters should be the same.
- Ex:
function add(a:string, b:string):string;
-
- function add(a:number, b:number): number;
-
- function add(a: any, b:any): any {
- return a + b;
- }
-
- add("Hello ", "Steve"); // returns "Hello Steve"
- add(10, 20); // returns 30

Rest Parameters

- ▶ When the number of parameters that a function will receive is not known or can vary, we can use rest parameters. In JavaScript, this is achieved with the "arguments" variable. However, with TypeScript, we can use the rest parameter denoted by ellipsis
- ▶ We can pass zero or more arguments to the rest parameter. The compiler will create an array of arguments with the rest parameter name provided by us.
- ▶ Ex:
function Greet(greeting: string, ...names: string[]) {
- ▶ return greeting + " " + names.join(", ") + "!";
- ▶ }
- ▶
- ▶ Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"
- ▶
- ▶ Greet("Hello");// returns "Hello !"
- ▶ Remember, rest parameters must come last in the function definition, otherwise the TypeScript compiler will show an error.



multiple files compile at a time

- All files compile at a time
- `tsc -init`
- `tsc -watch`
- It create tsconfig.json file.



Some files if isn't compile

- some files not compile when we can compile all file then,
- Added files/folders in tsconfig.json in last
- "exclude": [- "file2.ts"
-]



Only Some files are compile at a time

- Only Some files compile at a time
- Added files/folder name in tsconfig.json in last
- "include": [- "*/file2.ts"
-]



Compiled file separate

- All compiled file get in one folder
- Uncomment this code and put path in it.
- "outDir": "./build",

aliases

- ▶ TypeScript Type Aliases are a way to define custom data types that can simplify complex types or provide descriptive names for types. Type aliases can be used to create a new name for an existing type, or to define a union or intersection of types.
- ▶ In the above example, we define a type alias called "Employee" that describes an object with three properties: "name" of type string, "age" of type number, and "title" of type string. We then create an object called "employee" that matches the structure of the "Employee" type.

```
type Employee = {  
  name: string;  
  age: number;  
  title: string;  
}
```

```
const employee: Employee = {  
  name: 'John Doe',  
  age: 30,  
  title: 'Software Engineer'  
}
```

Summary

Type aliases can help make your code more readable and maintainable by providing descriptive names for complex types, and by allowing you to create reusable types that can be used throughout your codebase.



class

- ▶ JavaScript does not have a concept of class like other programming languages such as Java and C#. In ES5, you can use a constructor function and prototype inheritance to create a “class”.
- ▶ This is a constructor function for creating Person objects. It takes three parameters: ssn, firstName, and lastName. When a new Person object is created using this constructor function, the ssn, firstName, and lastName properties are set to the values passed in as arguments.
- ▶ The Person prototype has a method called getFullName which returns the full name of the person by concatenating the firstName and lastName properties with a space in between. This method is available to all Person objects created using this constructor function.
- ▶ Ex given bellow slide:



Example of constructor class

```
class Person {  
    ssn: string;  
    firstName: string;  
    lastName: string;  
  
    constructor(ssn: string, firstName: string, lastName: string) {  
        this.ssn = ssn;  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
    getFullName(): string {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```



Example of constructor class

Using the Person class is the same as the Person constructor function:

- `let person = new Person('171-28-0926','Kirti','Koyani');`
- `console.log(person.getFullName());`
- It would output the following to the console:
- **Kirti Koyani**
- Use class keyword to define a class in TypeScript.
- TypeScript leverages the ES6 class syntax and adds type annotations to make the class more robust.



Access modifiers

Access modifiers change the visibility of the properties and methods of a class. TypeScript provides three access modifiers:

- private
- protected
- public

Note that TypeScript controls the access logically during compilation time, not at runtime.

- The private modifier allows access within the same class.
- The protected modifier allows access within the same class and subclasses.
- The public modifier allows access from any location.

Privet modifier

- The private modifier limits the visibility to the same-class only.
- When you add the private modifier to a property or method, you can access that property or method within the same class.
- Any attempt to access private properties or methods outside the class will result in an error at compile time.
- Ex:- class Person {

```
private ssn: string;
private firstName: string;
private lastName: string;
private salary: number = 0;
constructor(ssn: string, firstName: string, lastName: string) {
    this.ssn = ssn;
    this.firstName = firstName;
    this.lastName = lastName; } // side...
}
```

```
class Person {
    private salary: number = 0;
    private calculateBonus(): number {
        return 0;
    }
    public setSalary(amount: number): void {
        this.salary = amount;
    }
    public getSalary(): number {
        return this.salary + this.calculateBonus();
    }
}

const person1 = new Person('123-45-6789', 'John', 'Doe');
person1.setSalary(50000);
console.log(person1.getSalary()); // output: 50000
```



Public modifier

The public modifier allows class properties and methods to be accessible from all locations. If you don't specify any access modifier for properties and methods, they will take the public modifier by default.

For example, the `getFullName()` method of the `Person` class has the public modifier. The following explicitly adds the public modifier to the `getFullName()` method:

```
Ex:- class Person {  
    private ssn: string;  
    private firstName: string;  
    private lastName: string;  
    constructor(ssn: string, firstName: string, lastName: string) {  
        this.ssn = ssn;  
        this.firstName = firstName;  
        this.lastName = lastName; } // side...  
}
```

```
class Person {  
    // ...  
    public getFullName(): string {  
        return `${this.firstName}  
${this.lastName}`;  
    }  
    // ...  
}
```

Protected modifier

- The protected modifier allows properties and methods of a class to be accessible within same class and within subclasses.
- When a class (child class) inherits from another class (parent class), it is a subclass of the parent class.
- The TypeScript compiler will issue an error if you attempt to access the protected properties or methods from anywhere else.

```
class Animal {  
    protected name: string;  
    constructor(name: string) {  
        this.name = name;  
    }  
    move(distanceInMeters: number) {  
        console.log(`${this.name} moved ${distanceInMeters}m.`);  
    }  
}  
  
class Snake extends Animal {  
    constructor(name: string) {  
        super(name);  
    }  
    crawl(distanceInMeters: number) {  
        console.log(`${this.name} crawled ${distanceInMeters}m.`);  
    }  
}  
  
const sam = new Snake("Sammy the Python");  
sam.crawl(5); // Output: Sammy the Python crawled 5m.  
sam.move(10); // Error: Property 'move' is protected and only  
              accessible within class 'Animal' and its subclasses.
```


readonly

TypeScript readonly access modifier to mark class properties as immutable property.

TypeScript provides the readonly modifier that allows you to mark the properties of a class immutable. The assignment to a readonly property can only occur in one of two places:

- In the property declaration.
- In the constructor of the same class.

```
➤ Ex1: class Person {  
    readonly birthDate: Date;  
    constructor(birthDate: Date) {  
        this.birthDate = birthDate;  
    }  
}
```

```
➤ Ex2: class Person {  
    constructor(readonly birthDate: Date) {  
        this.birthDate = birthDate;  
    }  
}
```



Inheritance



- A class can reuse the properties and methods of another class. This is called inheritance in TypeScript.
- The class which inherits properties and methods is called the **child class**. And the class whose properties and methods are inherited is known as the **parent class**. These names come from the nature that children inherit genes from parents.
- Inheritance allows you to reuse the functionality of an existing class without rewriting it.
- Use the extends keyword to allow a class to inherit from another class.
- Use super() in the constructor of the child class to call the constructor of the parent class.
- Also, use the super.methodInParentClass() syntax to invoke the methodInParentClass() in the method of the child class.



Example of inheritance

```
class Person {  
  constructor(private firstName: string, private lastName:  
string) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
  }  
  getFullName(): string {  
    return `${this.firstName} ${this.lastName}`;  
  }  
  describe(): string {  
    return `This is ${this.firstName} ${this.lastName}`;  
  }  
}
```

```
class Employee extends Person {  
  constructor(  
    firstName: string,  
    lastName: string,  
    private jobTitle: string) {  
  
    // call the constructor of the Person class:  
    super(firstName, lastName);  
  }  
}  
  
let employee = new Employee('John', 'Doe', 'Web  
Developer');  
  
console.log(employee.getFullName());  
console.log(employee.describe());
```

Method overriding


- When you call the `employee.describe()` method on the employee object, the `describe()` method of the Person class is executed that shows the message: This is John Doe.
- If you want the Employee class has its own version of the `describe()` method, you can define it in the Employee class like this: // side...
- In the `describe()` method, we called the `describe()` method of the parent class using the syntax `super.methodInParentClass()`.

```
class Employee extends Person {  
  constructor(  
    firstName: string,  
    lastName: string,  
    private jobTitle: string) {  
  
    super(firstName, lastName);  
  }  
  
  describe(): string {  
    return super.describe() + `I'm a ${this.jobTitle}.`;  
  }  
}
```

```
let employee = new Employee('John', 'Doe', 'Web Developer');  
console.log(employee.describe());
```



Static Methods and Properties

- In TypeScript, a **static** method or property is a member of a class that is associated with the class itself, rather than with instances of the class. This means that you can access the **static** member without creating an instance of the class.
 - Static properties and methods are shared by all instances of a class.
 - Use the static keyword before a property or a method to make it static.
- 

Example

```
class Circle {  
    static pi: number = 3.14159;  
  
    static calculateArea(radius: number): number  
    {  
        return this.pi * radius * radius;  
    }  
}
```

```
console.log(Circle.calculateArea(5)); // Output:  
78.53975
```

- In this example, we have defined a **Circle** class with a **static** property **pi** and a **static** method **calculateArea**. We can access the **static** property **pi** and the **static** method **calculateArea** without creating an instance of the **Circle** class.
- To access **static** members, we use the class name followed by the **.** operator, as shown in the example above. The **this** keyword in the **calculateArea** method refers to the **Circle** class itself, since **calculateArea** is a **static** method.
- **static** members are often used for utility functions and constants that are related to the class, but not specific to any instance of the class. Using **static** members can also help reduce memory usage, as you do not need to create an instance of the class to access the **static** members.

TypeScript interfaces

- ▶ TypeScript interfaces define contracts in your code and provide explicit names for type checking.
- ▶ Interfaces may have optional properties or readonly properties.
- ▶ Interfaces can be used as function types.
- ▶ Interfaces are typically used as class types that make a contract between unrelated classes.
- ▶ In this example, we have defined an interface called **Person** that defines the properties and methods that a person should have. The **Employee** class implements the **Person** interface by implementing the **age**, and **getAddress** properties and method.
- ▶ The **Employee** class constructor takes in the **age**, and **address** parameters, and assigns them to the class properties. The **getAddress** method returns the address of the employee.
- ▶ Finally, we create an instance of the **Employee** class and call the **getAddress** method to output the address of the employee.

```
interface Person {  
    age: number;  
    getAddress(): string;  
}  
  
class Employee implements Person {  
    age: number;  
    address: string;  
    constructor(age: number, address: string) {  
        this.age = age;  
        this.address = address;  
    }  
    getAddress(): string {  
        return this.address;  
    }  
}  
  
const employee = new Employee(30, "123 Main St");  
console.log(employee.getAddress()); // Output: 123 Main St
```

Type Guards

- ▶ TypeScript Type Guards are a way to check the type of a variable or object at runtime, and perform actions based on the type of the variable. A Type Guard is a TypeScript expression that returns a boolean, and is used to narrow down the type of a variable.
- ▶ **typeof** Type Guard: This type guard is used to check the type of a primitive value, such as a string, number, boolean, or symbol. For example:

```
function exampleFunc(param:
string | number) {
  if (typeof param === "string") {
    console.log("The parameter is a
string.");
  } else {
    console.log("The parameter is a
number.");
  }
}
```


instanceof Type Guard

- ▀ **instanceof** Type Guard: This type guard is used to check if an object is an instance of a particular class. For example:

```
▀ class MyClass {  
▀   prop: string;  
▀   constructor(prop: string) {  
▀     this.prop = prop;  
▀   }  
▀ }  
▀  
▀ function exampleFunc(param: MyClass | number) {  
▀   if (param instanceof MyClass) {  
▀     console.log("The parameter is an instance of  
▀     MyClass.");  
▀   } else {  
▀     console.log("The parameter is a number.");  
▀   }  
▀ }
```


User-defined Type Guard

- User-defined Type Guard: This type guard is a custom function that checks if an object satisfies certain conditions. For example:
- In this way, we can use Type Guards in TypeScript to perform runtime type checks and take actions based on the type of a variable or object.

```
■ interface MyInterface {  
■   prop: string;  
■ }  
■ function isMyInterface(obj: any): obj is MyInterface {  
■   return "prop" in obj;  
■ }  
■ function exampleFunc(param: MyInterface | number) {  
■   if (isMyInterface(param)) {  
■     console.log("The parameter is an object that  
■       implements MyInterface.");  
■   } else {  
■     console.log("The parameter is a number.");  
■   }  
■ }
```

type casting

In TypeScript, type casting is a way of changing the data type of a variable from one type to another. There are two ways to do type casting in TypeScript: implicit type casting and explicit type casting.

- Implicit Type Casting:
 - Implicit type casting occurs automatically by the TypeScript compiler when it is safe to do so. For example, if you assign a number to a variable of type "any", TypeScript will implicitly cast the number to the "any" type. Here's an example:
 - `let anyVariable: any = 5;`
 - `let stringValue: string = anyVariable; //` Implicit type casting from number to string
 - `console.log(stringVariable); // Output: "5"`
- Explicit Type Casting:
 - Explicit type casting is when you manually change the type of a variable using the "as" keyword. Here's an example:
 - `let anyVariable: any = 5;`
 - `let stringValue: string = anyVariable as string; //` Explicit type casting from number to string
 - `console.log(stringVariable); // Output: "5"`
 - In the above example, we are using the "as" keyword to explicitly cast the "anyVariable" from a number to a string. The resulting "stringValue" will still have the value "5", but its type will be "string" instead of "number". Note that using explicit type casting can be risky if the cast is not valid, as it can lead to type errors at runtime.



Difference between assertion and casting

- The key difference between type casting and assertion in TypeScript is that type casting actually changes the data type of a variable, while assertion does not. Type casting is a way of converting a value from one type to another, while assertion is a way of telling the TypeScript compiler what the type of a value is, without actually changing its data type.
- Type casting can be done implicitly or explicitly, while assertion can be done using angle bracket syntax or the "as" keyword. Type casting is useful when you need to change the data type of a value, while assertion is useful when you know the type of a value but TypeScript is not able to infer it on its own.
- Another important difference is that type casting can be risky if you are not careful, as it can lead to type errors at runtime. On the other hand, assertion is generally safer, as it does not change the actual data type of a value.

TypeScript Generics

- ▶ TypeScript Generics are a powerful feature that allows you to write reusable code that can work with different data types. Generics allow you to create functions, classes, and interfaces that can accept a variety of data types without sacrificing type safety.
- ▶ In the above example, the function "printValue" is defined with a type parameter "T", which is used to represent any data type. The function takes a single argument "value" of type "T", and prints it to the console. When we call the function, we specify the type of "T" using angle brackets (<>). We can call this function with any data type we want, and TypeScript will ensure that the type is correct at compile time.

```
function printValue<T>(value:T): void {  
    console.log(`Value: ${value}`);  
}
```

// Using the generic function

```
printValue<string>('Hello World!'); // Output: "Value:  
Hello World!"
```

```
printValue<number>(123); // Output: "Value: 123"
```

- ▶ In summary, TypeScript Generics allow you to write reusable code that can work with different data types while maintaining type safety. Generics can be used with functions, classes, and interfaces, and are a powerful tool for writing flexible and maintainable code.



Thank You