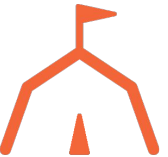


TypeScript



TypeScript

TypeScript is a strongly typed, object-oriented, compiled language.

Think of TypeScript as C# or Java for JavaScript developers.



TypeScript is Strongly-Typed

Variables and functions have pre-set, unchanging types in the code. When a variable is declared, a type is specified, such as string, number, or boolean.

```
let name: string = "Martha";  
let age: number = 97;
```



TypeScript is Strongly-Typed

This helps you catch typos and other errors as you type your code!

```
let greeting:string;  
  
greating = "Hello!"; // ERROR misspelled variable  
greeting = 123; // ERROR wrong type
```



TypeScript is Compiled

TypeScript cannot be run in the browser or directly with Node.js.

It must be compiled to JavaScript using the **tsc** (TypeScript Compiler) command.



Compiled to JavaScript

```
const user:string = "Ivan";  
const length:number = user.length;
```

TypeScript

Compiled with **tsc**

JavaScript

```
const user = "Ivan";  
const length = user.length;
```



TypeScript Project Setup



TypeScript: Type Annotations



Type Annotations

Type annotations are used to define the intended data type for variables, function parameters, the intended number of parameters for a function, as well as what data type a function should return.



Variables

Specify type with colon after variable name.

```
let firstName: string = "Martha";
```

```
let age: number = 97;
```

```
let retired: boolean = true;
```

```
retired = "Heck, Yeah!"; // ERROR, wrong type
```

Arrays

Specify array with element type and brackets.

```
let names: string[] = [ "Martha", "Barry", "Tim" ];  
let ages: number[] = [ 97, 23, 2 ];  
  
names.push("Lakshmi"); // OK  
ages.push("four"); // ERROR wrong element type
```

Function Parameters

Parameter types are the same as variables.

```
function find(names: string[], maxlen: number) {  
    ...  
}
```

```
find(["a", "an", "the"], 2); // OK
```

```
find([1, 2, 3], 3); // ERROR wrong array type
```

Function Return Type

Annotation after function parameters.

```
function countLetters(word: string): number {  
    ...  
}  
  
let count:number = countLetters("ABC"); // OK  
let x:string = countLetters("ABC"); // ERROR
```

Void Return Type

Use **void** if a function does not return anything

```
function greet(name:string): void {  
  console.log(`Hello, ${name}!`);  
}
```



Arrow Functions

Arrow function syntax is similar.

```
const countLetters = (word: string): number =>
{
    ...
}
```



Optional Parameters

If a parameter is optional, add a question mark.

```
function greet(name: string, title?: string) {  
    ...  
}  
  
greet("Velma");  
greet("Isaac Newton", "Sir");
```


Default Parameters

A parameter can also be optional by adding a default value.

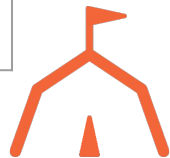
```
function add(a: number, b: number = 1) {  
    return a + b;  
}
```

```
add(2, 3); // 5
```

```
add(7);    // 8
```

Basic Types

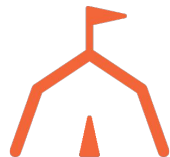
Type	Annotation	Description
Boolean	<code>let done: boolean = false</code>	Same as in JavaScript.
Number	<code>let num: number = 10</code>	Same as in JavaScript.
String	<code>let word: string = 'Hi'</code>	Same as in JavaScript.
Array	<code>let nums: number[] = [1,2]</code>	An Array of a specific type of element.



Basic Types

Type	Annotation	Description
Any	<pre>let something: any = 1; something = 'hi'</pre>	Allows for dynamic type of a variable.
Void	<pre>const print(): void => console.log('something')</pre>	Used with functions that return nothing.
(or)	<pre>let pet: string null; let age: string number;</pre>	Allow multiple types.

Find more information on types [here](#).



Interfaces

An interface is an outline for an Object.

Using the **interface** keyword allows an Object's properties, methods, and types to be defined.

An interface looks like an object literal except properties are not separated by commas, but by semicolons.

```
interface Person {  
  firstName: string;  
  lastName: string;  
}
```

Interfaces

Add this interface to our greeter example and see how that changes the type annotations.

```
interface Person {  
  firstName: string;  
  lastName: string;  
}  
  
const greeter = (person: Person): string =>  
  `Hello ${person.firstName} ${person.lastName}`;  
  
const user: Person = { firstName: 'Ivan', lastName: 'Herndon' };  
  
console.log(greeter(user));
```

Interfaces

When using an interface as a type annotation, all previous benefits remain the same.

If the function argument or variable is missing a property or it is of the wrong type, TypeScript will throw an error during compilation to assist in prevent runtime errors.



TypeScript: Null & Undefined



Null & Undefined Not Allowed

If a variable is specified as a string, it must always contain an actual string, **not null or undefined**.

The same is true for any type, e.g. number, etc.

```
let pet: string;  
  
pet = "Muffins";  
pet = null; // ERROR  
pet = undefined; // ERROR
```


Type Annotations: Nullable Variables

If a variable should be able to be null or undefined, specify the option with a pipe (|).

```
let vendor: string|undefined;
```

```
let pet: string|null;
```

```
pet = "Muffins";
```

```
pet = null; // OK!
```

Required Properties on Interfaces

Every property on an interface is required by default.

```
interface Airplane {  
  pilot: string;  
  copilot: string;  
}
```

```
// ERROR! copilot is missing.
```

```
let myPlane: Airplane = { pilot: "Snoopy "};
```

Optional Properties on Interfaces

But properties may be specified as optional by adding a question mark.

```
interface Airplane {  
  pilot: string;  
  copilot?: string;  
}
```

```
let myPlane: Airplane = { pilot: "Snoopy "};  
let yourPlane: Airplane = { pilot: "Snoopy",  
                             copilot: "Woodstock" };
```

Dealing with Nullable & Optional

TypeScript is very careful about nullable variables and optional properties. You may run into errors related to this when you try to use these variables.



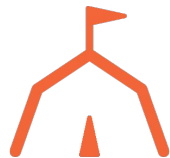
Examples of Errors

```
const words = [ "Apple", "Berry", "Chip", "Dip" ];  
const word: string|undefined =  
    words.find(aWord => aWord.startsWith("J"));  
  
// ERROR: Object is possibly 'undefined'.  
console.log(word.length);  
  
// ERROR: Argument of type 'string | undefined' is  
not assignable to parameter of type 'string'.  
"Apple Pie".indexOf(word);
```

Solutions

Here are some situations. Each has a different solution.

1. We are certain it will actually not be null or undefined.
2. We want to provide a backup "default" value in case it is null or undefined.
3. We need different logic if it is null or undefined.
4. We're accessing a property, and it's okay if either the object or the property is null or undefined.



Solution 1: It's Not Null

If we are certain it will actually not be null or undefined, use the "non-null assertion operator", which is an exclamation point after the variable.

```
console.log(word!.length);  
"Apple Pie".indexOf(word!);
```

Solution 2: Provide a Default

If we want to provide a backup "default" value, use the "nullish coalescing operator" (??). The value after the operator is the backup value.

```
"Apple Pie".indexOf(word ?? "None");
```



Solution 3: Different Logic

If we need different logic when it is null or undefined, use an if/else or ternary

```
if (word !== undefined) {  
    "Apple Pie".indexOf(word);  
} else {  
    console.log("Word not found.");  
}
```

Solution 4: Null Object is Okay

If we're accessing a property of an object that might be null or undefined, the "optional chaining operator" (?.) will stop and return null/undefined rather than attempting to access the property and crashing the code.

```
console.log(word?.length);
```



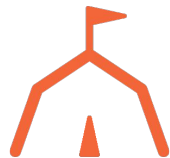
Import & Export



ES6 Modules

In Node.js programs, we've been using **require()** and **module.exports** to use other modules. This is part of the CommonJS module system.

TypeScript prefers the ES6 Module System. The systems are similar, but the syntax is a bit different.



Individual Exports

Export individual items like this.

```
export const city = "Detroit";  
export const state = "MI";
```

And import them in another file like this.

```
import { city, state } from "./first-file";  
console.log(city + ", " + state); // Detroit, MI
```



Export Variables, Interfaces, etc.

```
export let planet = "Earth";  
export interface Person {  
  name: string;  
  age: number;  
}  
export function birthday(person: Person):void {  
  person.age++;  
}
```



Default Export

Each file can also export one default item.

```
const PI = 3.14159265;  
export default PI;
```

Import it in another file like this.

```
import PI from "./pi";  
console.log(PI); // 3.14159265
```



Default & Individual Exports

It's possible to combine both.

```
export default interface Person {  
  name: string;  
  age: number;  
}  
export function birthday(person: Person):void {  
  person.age++;  
}
```

```
import Person, { birthday } from './Person';
```



TypeScript Reference

Read more about TypeScript at typescriptlang.org.

For import/export, see the section on [Modules](#).



Classes



Classes

- TypeScript classes are similar to Java classes.
- Fields are called properties and are public by default. We don't usually use getters and setters in TypeScript or JavaScript.



Class Example

class name

```
class Player {
```

```
  name: string;
```

```
  jersey: number;
```

properties

```
  constructor(name: string, jersey: number) {
```

```
    this.name = name;
```

```
    this.jersey = jersey;
```

constructor

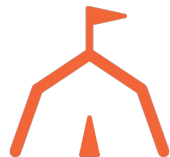
```
  }
```

```
}
```

Class Instances

Use the **new** keyword to create objects from the class blueprint. These are called **instances**.

```
let mike: Player = new Player("Michael Jordan", 23);  
let p1: Player = new Player("Isiah Thomas", 11);  
let p2: Player = new Player("Lionel Messi", 10);
```



Property Initial Value

Properties should either have their value set in the constructor or given an initial value.

```
class Timer {  
  name: string;  
  time: number = 0; // value starts at 0  
  constructor (name: string) {  
    this.name = name;  
  }  
}
```

Methods

```
class Circle {  
  radius: number;  
  constructor(radius: number) {  
    this.radius = radius;  
  }  
}
```

classes can have
methods just like Java

```
  getArea(): number {  
    return Math.PI * this.radius * this.radius;  
  }  
  getCircumference(): number {  
    return 2 * Math.PI * this.radius;  
  }  
}
```

this Keyword

this. must **always** be used inside the class to refer to properties and methods of the current instance. Unlike Java, it is never optional.

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



Modifiers

- **public** - a property/method can be used anywhere (this is the default)
- **private** - a property/method cannot be used outside of this class
- **readonly** - a property cannot be changed



Public

```
class Player {  
    public name: string; // public is the default  
    constructor(name: string) {  
        this.name = name; // VALID inside  
    }  
}  
  
let mike: Player = new Player("Michael Jordan");  
console.log(mike.name); // also VALID outside
```

Private

```
class Player {  
  private name: string;  
  constructor(name: string) {  
    this.name = name; // VALID inside  
  }  
}  
  
let mike: Player = new Player("Michael Jordan");  
console.log(mike.name); // INVALID outside
```

Readonly

```
class Player {  
  readonly name: string;  
  constructor( name: string) {  
    this.name = name; // set only in the constructor  
  }  
}  
  
let mike: Player = new Player("Michael Jordan");  
console.log(mike.name); // reading is VALID  
mike.name = "Air Jordan"; // writing INVALID
```

Parameter Properties

Add a modifier to a constructor parameter to indicate that it is a property. This allows us to define and set the property all in one place. It is essentially a shortcut.



Parameter Properties

For example, these two classes are equivalent.

```
class Player {  
    constructor(  
        public name: string) {}  
}
```

Adding a modifier makes this a parameter property.

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



TypeScript Classes Reference

Read more at typescriptlang.org.



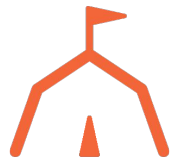
Array Methods



Array Methods

The following slides cover some important methods for working with arrays. All of these apply to both TypeScript and JavaScript.

- Adding to an array (**push**)
- Removing from an array (**splice**)
- Replacing or modify an item in an array ([] syntax)
- Search an array for a single match (**find**)
- Search an array to get the index (**findIndex**)
- Search an array for all matches (**filter**)



Add to an array

```
let colors = [ "orange", "yellow", "green", "blue" ];  
  
// add to end  
colors.push("violet");  
// add to beginning  
colors.unshift("red");
```



Remove from an array

```
let colors = [ "red", "green", "cobalt", "blue" ];  
  
// remove cobalt  
// (index = 2, number of elements to remove = 1)  
colors.splice(2, 1);
```



Replace an item

```
let colors = [ "red", "green", "blue" ];  
  
// replace green with white  
colors[1] = "white";
```



Search for single match (find)

`.find()` takes a callback function. It runs that function for every element and finds the first one that returns true.

```
let meals: Meal[] = [ { name: "spaghetti", price: 6 },  
                      { name: "lasagnia", price: 12 },  
                      { name: "pizza", price: 10 }      ];  
  
// returns { name: "pizza", price: 10 }  
meals.find(meal => meal.name === "pizza");  
// returns undefined  
meals.find(meal => meal.name === "curry");
```

Search for index (findIndex)

`.findIndex()` is the same as `.find()` but it returns the index of the element rather than the element itself.

```
let meals: Meal[] = [ { name: "spaghetti", price: 6 },  
                      { name: "lasagnia", price: 12 },  
                      { name: "pizza", price: 10 }      ];  
  
// returns 2  
meals.findIndex(meal => meal.name === "pizza");  
// returns -1  
meals.findIndex(meal => meal.name === "curry");
```

Search for all matches (filter)

`.filter()` returns an array of all elements for which the callback function returns true.

```
let meals: Meal[] = [ { name: "spaghetti", price: 6 },  
                      { name: "lasagnia", price: 12 },  
                      { name: "pizza", price: 10 }      ];  
  
// returns [ { name: "spaghetti", price: 6 },  
            { name: "pizza", price: 10 } ]  
meals.filter(meal => meal.price <= 10 );  
// returns []  
meals.filter(meal => meal.name === "curry");
```

Recap

- Write code in TypeScript.
- Annotate variables and function with types.
- Create and use interfaces to define object structures.
- Integrate multiple TypeScript files using export and import.
- Create and use classes with properties, constructor, and methods.
- Manipulate arrays using array methods.

