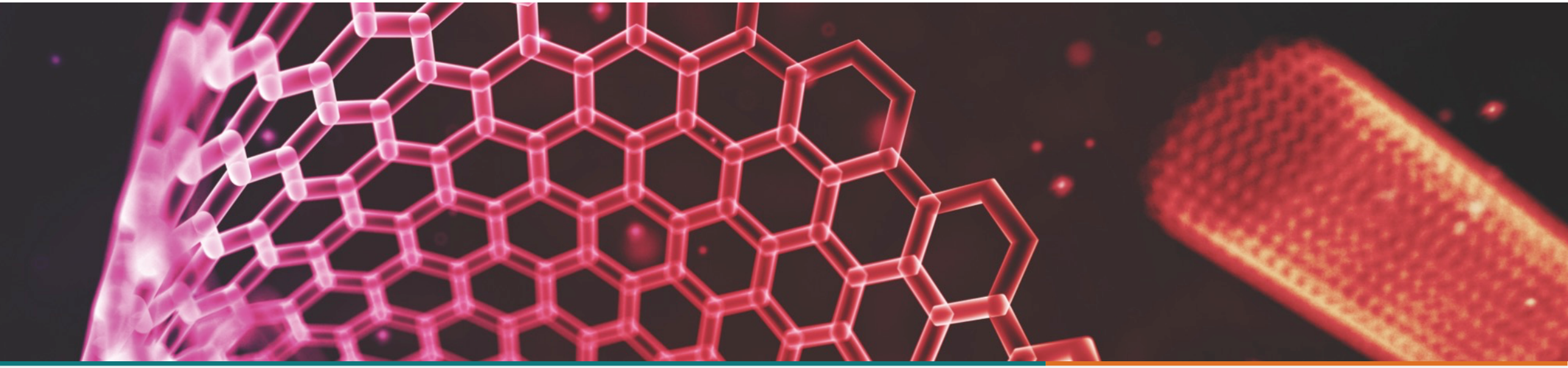
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 554 – Web Programming II

TypeScript



What is TypeScript?

TypeScript is an open source programming language developed by Microsoft that is a superset of JavaScript in order to provide a great deal of syntactical sugar.

- https://www.typescriptlang.org/
- https://www.typescriptlang.org/play/index.html

What is the Problem We Are Trying to Solve?

Many programmers believe that adding structure and tooling are paramount to project success.

- Tools allow us to know something can fail before it does
- Structure provides patterns for us to follow to repeat duplicate work

In the wild-west that is known as JavaScript, there was historically a serious lack of structure and tooling until the last 5 years (approximately)

- One of the things that JS Devs most missed has been static type checking

How Does TypeScript Solve That Problem?

TypeScript forces a build process step into your JavaScript, and therefore provides:

- Static type checking; never have to worry about passing a number when you should be passing a string again!
- Intellisense! By the addition of a type system, your IDEs can often more easily suggest what you are trying to write.
- Next-generation JS features today such as async / await support, transpiled down to today's JavaScript; similar to Babel

Features of TypeScript

TypeScript is just JavaScript. TypeScript starts with JavaScript and ends with JavaScript. Typescript adopts the basic building blocks of your program from JavaScript. Hence, you only need to know JavaScript to use TypeScript. All TypeScript code is converted into its JavaScript equivalent for the purpose of execution.

TypeScript supports other JS libraries. Compiled TypeScript can be consumed from any JavaScript code. TypeScript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.

JavaScript is TypeScript. This means that any valid **.js** file can be renamed to **.ts** and compiled with other TypeScript files.

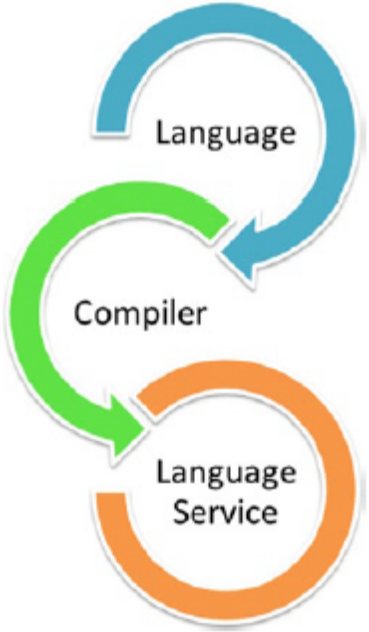
TypeScript is portable. TypeScript is portable across browsers, devices, and operating systems. It can run on any environment that JavaScript runs on. Unlike its counterparts, TypeScript doesn't need a dedicated VM or a specific runtime environment to execute.

Components of TypeScript

Language – It comprises of the syntax, keywords, and type annotations.

The TypeScript Compiler – The TypeScript compiler (tsc) converts the instructions written in TypeScript to its JavaScript equivalent.

The TypeScript Language Service – The "Language Service" exposes an additional layer around the core compiler pipeline that are editor-like applications. The language service supports the common set of a typical editor operations like statement completions, signature help, code formatting and outlining, colorization, etc.



Why Use TypeScript?

- The benefits of TypeScript include:
- **Compilation** – JavaScript is an interpreted language. Hence, it needs to be run to test that it is valid. It means you write all the codes just to find no output, in case there is an error. Hence, you have to spend hours trying to find bugs in the code. The TypeScript transpiler provides the error-checking feature. TypeScript will compile the code and generate compilation errors, if it finds some sort of syntax errors. This helps to highlight errors before the script is run.

- **Strong Static Typing** – JavaScript is not strongly typed. TypeScript comes with an optional static typing and type inference system through the TLS (TypeScript Language Service). The type of a variable, declared with no type, may be inferred by the TLS based on its value.
- TypeScript **supports type definitions** for existing JavaScript libraries. TypeScript Definition file (with **.d.ts** extension) provides definition for external JavaScript libraries. Hence, TypeScript code can contain these libraries.

- TypeScript **supports Object Oriented Programming** concepts like classes, interfaces, inheritance, etc.

What Are the Downsides?

Additional build step

Another layer in your dev process that limits developer hiring pool. Have to download typings for existing libraries to fully benefit from TS

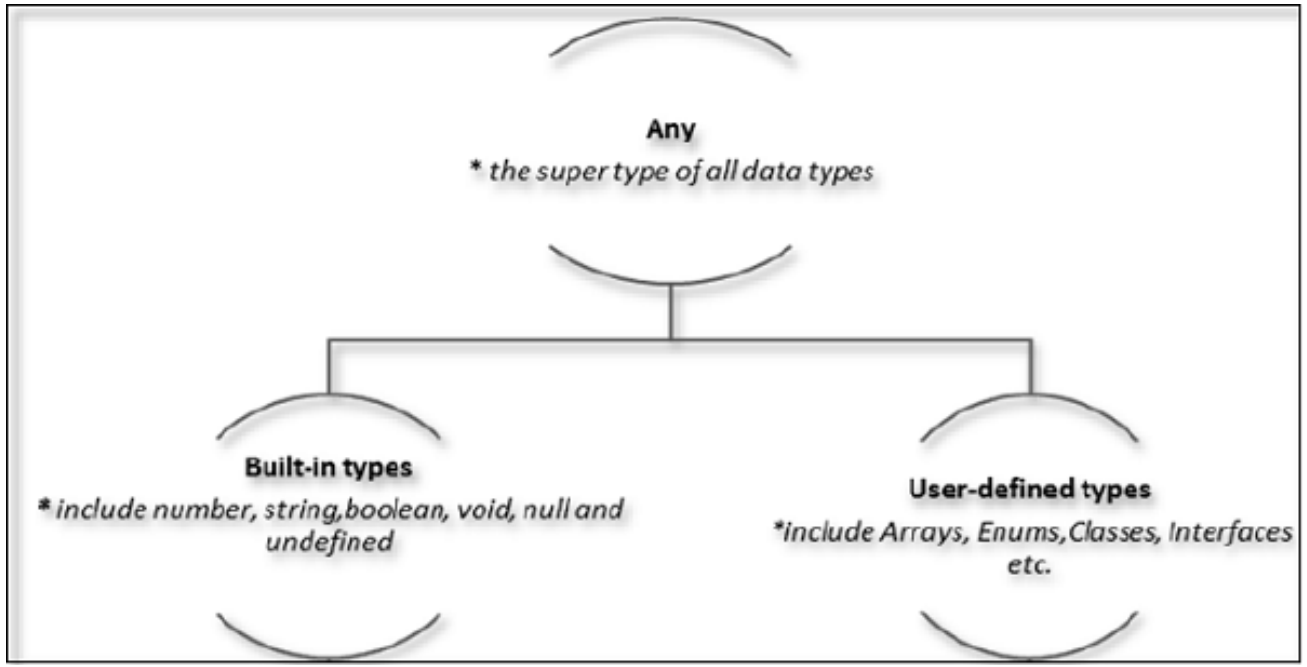
- More dependencies, yay

TypeScript Types

The Type System represents the different types of values supported by the language.

The Type System checks the validity of the supplied values, before they are stored or manipulated by the program. This ensures that the code behaves as expected.

The Type System further allows for richer code hinting and automated documentation too.
TypeScript provides data types as a part of its optional Type System.
The data type classification is as given below –



TypeScript Types

1. The Any type:
The **any** data type is the super type of all types in TypeScript. It denotes a dynamic type.

Using the **any** type is equivalent to opting out of type checking for a variable.

2. Built-in types
The following table illustrates all the built-in types in TypeScript

Data type	Keyword	Description
Number	number	Double precision 64-bit floating point values. It can be used to represent both, integers and fractions.
String	string	Represents a sequence of Unicode characters
Boolean	boolean	Represents logical values, true and false
Void	void	Used on function return types to represent non-returning functions
Null	null	Represents an intentional absence of an object value.
Undefined	undefined	Denotes value given to all uninitialized variables

Note – There is no integer type in TypeScript and JavaScript.

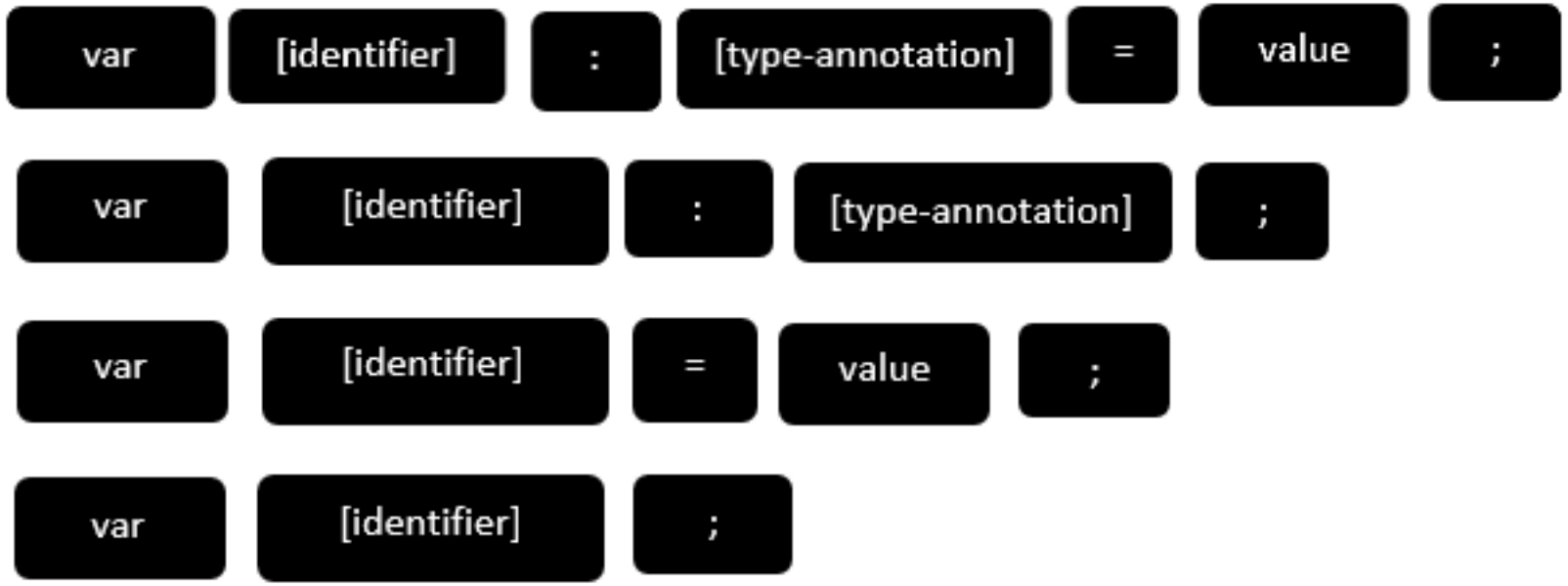
3. User-defined Types
User-defined types include Enumerations (enums), classes, interfaces, arrays, and tuple.

TypeScript Defining Variables

Variable Declaration in TypeScript

The type syntax for declaring a variable in TypeScript is to include a colon (:) after the variable name, followed by its type. Just as in JavaScript, we use the **var** keyword to declare a variable.

When you declare a variable, you have four options



Variable Examples

```
var name:string = "John";
var score1:number = 50;
var score2:number = 42.50
var sum = score1 + score2
console.log("name"+name)
console.log("first score: "+score1)
console.log("second score: "+score2)
console.log("sum of the scores: "+sum)
```

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var name = "John";
var score1 = 50;
var score2 = 42.50;
var sum = score1 + score2;
console.log("name" + name);
console.log("first score: " + score1);
console.log("second score : " + score2);
console.log("sum of the scores: " + sum);
```

How Do We Compile TypeScript?

TypeScript is compiled by installing the *typescript* module globally

- **npm install -g typescript**

This will install the TypeScript compiler which can run for the command line and be used to compile any project.

- Projects are often configured using a ***tsconfig*** file, which sets up basic rules

Local projects can also use a version of TypeScript defined locally

- This is allows you to set which version of TS to use; there are many!

How Does It All Work?

Like most other build systems, you simply setup which files you want to target and run the compiler.

It will follow links to other TS files you reference, which will force those files to be built as well.

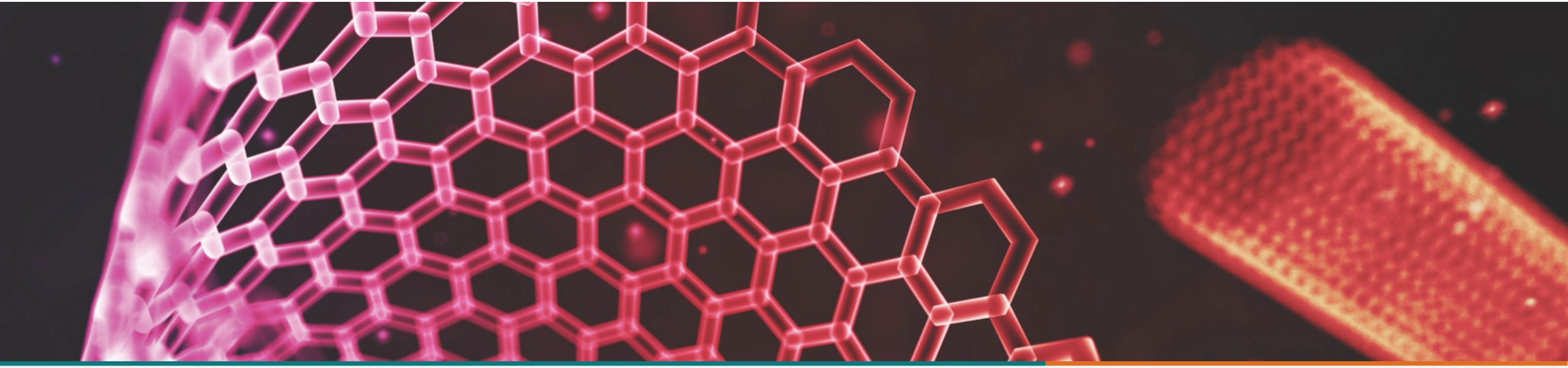
To resolve references to projects that were **not** created in TypeScript, you will need to download a *Type Definition* as well. This will allow your compiler to make sure you are using external libraries properly, as well.

- **https://www.typescriptlang.org/docs/handbook/declaration-files/consumption.html**

There are many definitions available for both server and frontend code, meaning you can easily write your backend and frontend TypeScript

- **http://microsoft.github.io/TypeSearch/**

TypeScript Usage



The Majority of TypeScript is JavaScript

The majority of TypeScript is plain old vanilla JavaScript.

Any syntax that is valid in JavaScript will be valid in TypeScript, however you can write more advanced concepts in TypeScript that will be transpiled down into a particular version of JavaScript

- You can set this version in your tsconfig file
- Similar to babel in concept

Experimenting With TypeScript

To experiment with TypeScript, you can use the online TypeScript Playground maintained by Microsoft

https://www.typescriptlang.org/play/index.html

This is fairly useful for a syntax checking and experimentation.

Integrating TypeScript

Integrating TypeScript into existing projects is extraordinarily easily

You can effectively start dropping in TypeScript one piece at a time and compiling smaller portions of your codebase.

Adding Types to Methods

We can begin by adding TypeScript to existing methods in order to add static type checking to methods.

On the left is an example of a TypeScript file, while the right is the compiled JavaScript.

```typescript
function Greeter(greeting: string) {
    this.greeting = greeting;
}

Greeter.prototype.greet = function() {
    return "Hello, " + this.greeting;
}

let greeter = new Greeter("world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
};

document.body.appendChild(button);
```

```javascript
function Greeter(greeting) {
    this.greeting = greeting;
}
Greeter.prototype.greet = function () {
    return "Hello, " + this.greeting;
};
var greeter = new Greeter("world");
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
    alert(greeter.greet());
};
document.body.appendChild(button);
```

Classes

Unlike vanilla JavaScript, we can easily build out our own classes in TypeScript that will get compiled down into objects.

This allows us to have strong type-checking for our custom data, as well as embrace more Object-Oriented Style Concepts.

In JavaScript, OOP is not necessarily always the answer, but composing logical data is generally a good thing.

```typescript
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
}

document.body.appendChild(button);
```

```javascript
var Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
}());
var greeter = new Greeter("world");
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
    alert(greeter.greet());
};
document.body.appendChild(button);

```

Interfaces

Because of duck-typing, we can define and use interfaces we do not explicitly inherit on our classes. if it looks like an interface, it acts like an interface.

```
class Student {
    fullName: string;
    constructor(public firstName: string, public middleInitial: string, public lastName
: string) {
        this.fullName = firstName + " " + middleInitial + " " + lastName;
    }
}

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

function greeter(person : Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

let user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);
```

Getters/Setters

It is easy to override getters and setters for your convenience, so that you do not have to continually call methods on your own. It allows for conveniently abstracting logic away.

```
1
2  class Employee {
3      private _fullName: string;
4
5      get fullName(): string {
6          return this._fullName;
7      }
8
9      set fullName(newName: string) {
10         if (newName) {
11             this._fullName = newName;
12         }
13         else {
14             throw ("Error: No name provided!");
15         }
16     }
17 }
18
19 let employee = new Employee();
20 employee.fullName = "Bob Smith";
21 if (employee.fullName) {
22     console.log(employee.fullName);
23 }
```

```
1  var Employee = /** @class */ (function () {
2      function Employee() {
3      }
4      Object.defineProperty(Employee.prototype, "fullName", {
5          get: function () {
6              return this._fullName;
7          },
8          set: function (newName) {
9              if (newName) {
10                 this._fullName = newName;
11             }
12             else {
13                 throw ("Error: No name provided!");
14             }
15         },
16         enumerable: true,
17         configurable: true
18     });
19     return Employee;
20 }());
21 var employee = new Employee();
22 employee.fullName = "Bob Smith";
23 if (employee.fullName) {
24     console.log(employee.fullName);
25 }
26
```

Enums

It is very strange that enums have not been added to JavaScript by this point but we can use them with TypeScript!

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2];

alert(colorName); // Displays 'Green' as its value is 2 above
```

Generic Types

Generics are a way of allowing you to set up a type for a function to return, or a class to use, across your application. For example, **Array<string>** is actually a Generic Class, **Array**, with a type parameter of **string**. As a result, all array methods will expect strings to be pushed, for filter to return a new array of strings, etc.

```typescript
class Greeter<T> {
    greeting: T;
    constructor(message: T) {
        this.greeting = message;
    }
    greet() {
        return this.greeting;
    }
}

let greeter = new Greeter<string>("Hello, world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
}

document.body.appendChild(button);
```

```javascript
var Greeter = /** @class */ (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return this.greeting;
    };
    return Greeter;
}());
var greeter = new Greeter("Hello, world");
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
    alert(greeter.greet());
};
document.body.appendChild(button);
```

String Literal Types

Similar to enums, you can also use strings that are bound to certain values.

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
        if (easing === "ease-in") {
            // ...
        }
        else if (easing === "ease-out") {
        }
        else if (easing === "ease-in-out") {
        }
        else {
            // error! should not pass null or undefined.
        }
    }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed here
```

Union Types

Union Types allow you to dictate that a parameter may be one of many types; such as expecting a **string** or a **number**.

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation
```

TypeScript With JSX

It is quite simple to have Typescript compile JSX with type checking.

- https://www.typescriptlang.org/docs/handbook/jsx.html
- https://www.typescriptlang.org/docs/handbook/react-&-webpack.html
- https://github.com/Microsoft/TypeScript-React-Starter#typescript-react-starter

Questions?