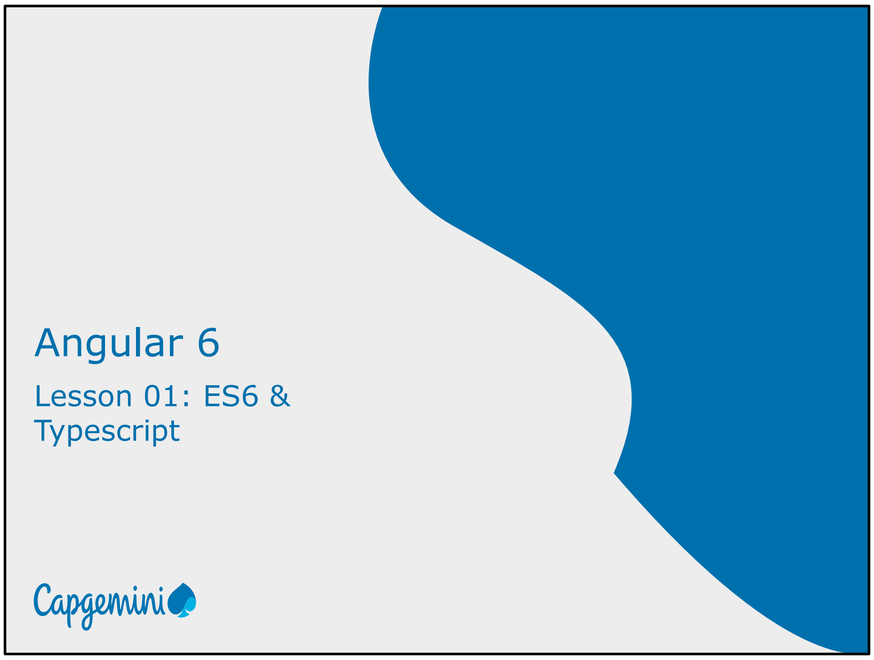


Angular 6

Lesson 01: ES6 & Typescript

Capgemini

Lesson Objectives

- ➢Var, Let and Const keyword
- ➢Arrow functions, default arguments
- ➢Spread operator / Rest Parameters
- ➢Template Strings, String methods
- ➢Object de-structuring
- ➢Typescript Fundamentals
- ➢Types & type assertions
- ➢Typescript OOPS - Classes, Interfaces, Constructor, etc
- ➢Creating custom types
- ➢Decorators

ES6

- ECMAScript (ES) is a scripting language specification standardized by ECMAScript International
- Languages like JavaScript, Jscript and ActionScript are governed by this specification.
- ECMASCRIPT was created to standardize **JavaScript**

ECMAScript History

Edition	Date published	Changes from prior edition
ES 1	June 1997	First edition
ES 2	June 1998	Editorial changes
ES 3	December 1999	Added regular expressions
ES 4	*Abandoned*	Fourth Edition was abandoned,
ES 5	December 2009	Adds "strict mode," Clarifies many ambiguities in the 3rd edition specification, Adds some new features, such as getters and setters,
ES 6	June 2015	Classes and modules, iterators and for/of loops, generators, arrow functions, typed arrays, collections (maps, sets and weak maps), promises,

ES 6 Features

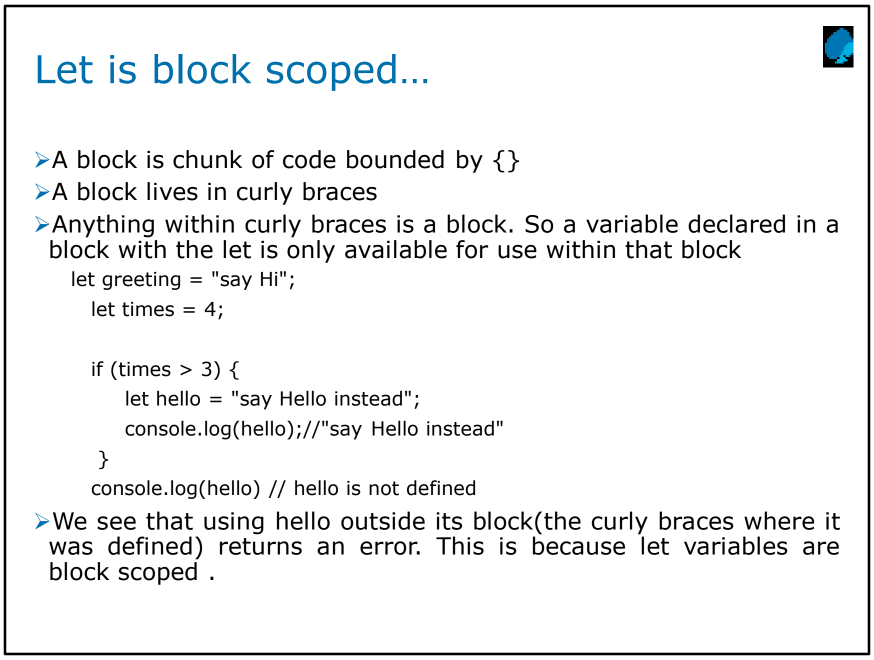
- Var, Let and Const keyword
- Arrow functions, default arguments
- Template Strings, String methods
- Object Destructuring
- Spread and Rest operator
- … many more

Block Scoping with let

- Before the advent of ES6, var declarations ruled as King.
- Var declarations are globally scoped or function/locally scoped

```
var greeter = "hey hi";
function newFunction() {
        var hello = "hello";
}
```

- Here, greeter is globally scoped because it exists outside a function while hello is function scoped. So we cannot access the variable hello outside of a function.
- let is preferred for variable declaration now



Let is block scoped…

- A block is chunk of code bounded by {}
- A block lives in curly braces
- Anything within curly braces is a block. So a variable declared in a block with the let is only available for use within that block

```
let greeting = "say Hi";
   let times = 4;

   if (times > 3) {
        let hello = "say Hello instead";
        console.log(hello);//"say Hello instead"
    }
   console.log(hello) // hello is not defined
```

- We see that using hello outside its block(the curly braces where it was defined) returns an error. This is because let variables are block scoped .

Const keyword

- Variables declared with the const maintain constant values. const declarations share some similarities with let declarations.
- Like let declarations, const declarations can only be accessed within the block it was declared.

```
const greeting = "say Hi";
greeting = "say Hello instead"; //error : Assignment to constant variable.
```

Summary…

- var declarations are globally scoped or function scoped while let and const are block scoped.
- var variables can be updated and re-declared within its scope; let variables can be updated but not re-declared; const variables can neither be updated nor re-declared.
- They are all hoisted to the top of their scope but while varvariables are initialized with undefined, let and const variables are not initialized.
- While var and let can be declared without being initialized, const must be initialized during declaration.

Arrow Functions…

- An **arrow function expression** has a shorter syntax than a function expression and does not have its own this, arguments, super, or new.target.
- These function expressions are best suited for non-method functions, and they cannot be used as constructors.
- It is an anonymous function expression that points to a single line of code. Following is the syntax for the same.

```
([param1, parma2,…param n] )=>statement;
```

Syntactic Variations in Arrow Functions…

- The syntax for arrow functions comes in many flavours depending upon what you're trying to accomplish. All variations begin with function arguments, followed by the arrow, followed by the body of the function. Both the arguments and the body can take different forms depending on usage. For example, the following arrow function takes a single argument and simply returns it:

```
var reflect = value => value;
// effectively equivalent to:

var reflect = function(value) {
    return value;
};
```

Syntactic Variations in Arrow Functions…

- If you are passing in more than one argument, then you must include parentheses around those arguments, like this:

```
var sum = (num1, num2) => num1 + num2;

// effectively equivalent to:

var sum = function(num1, num2) {
    return num1 + num2;
};
```

Syntactic Variations in Arrow Functions…

- If there are no arguments to the function, then you must include an empty set of parentheses in the declaration, as follows:

```
var getName = () => "Nicholas";

// effectively equivalent to:

var getName = function() {
    return "Nicholas";
};
```

Syntactic Variations in Arrow Functions…

- When you want to provide a more traditional function body, perhaps consisting of more than one expression, then you need to wrap the function body in braces and explicitly define a return value, as in this version of sum():

```
var sum = (num1, num2) => {
    return num1 + num2;
};

// effectively equivalent to:
 var sum = function(num1, num2) {
    return num1 + num2;
};
```

Default function parameters

- In ES6, a function allows the parameters to be initialized with default values, if no values are passed to it or it is undefined. The same is illustrated in the following code.

```
function add(a, b = 1) {
   return a+b;
}
console.log(add(4))
```

- The above function, sets the value of b to 1 by default. The function will always consider the parameter b to bear the value 1 unless a value has been explicitly passed.

Spread operator / Rest Parameters

- ES6 introduced "…" operator which is also called as spread operator. When "…" operator is applied on an array it expands the array into multiple variables in syntax wise.
- When its applied to a function argument it makes the function argument behave like array of arguments.
- To declare a rest parameter, the parameter name is prefixed with three periods, known as the spread operator.

```
function fun1(...params) {
  console.log(params.length);
}
fun1();
fun1(5);
fun1(5, 6, 7);
```

- **Note – Rest parameters should be the last in a function's parameter list.**

Template Strings, String methods

- Syntactically these are strings that use backticks (i.e. `) instead of single (') or double (") quotes. The motivation of Template Strings is three fold:
  - String Interpolation
  - Multiline Strings
  - Tagged Templates

String interpolation…

- A common use case is when you want to generate some string out of some static strings + some variables. For this you would need some templating logic and this is where template strings get their name from. Here's how you would potentially generate an html string previously:

```
var lyrics = 'Never gonna give you up';
var html = '<div>' + lyrics + '</div>';
```

Now with template strings you can just do:

```
var lyrics = 'Never gonna give you up';
var html = `<div>${lyrics}</div>`;
```

Multiple Strings…

- Ever wanted to put a newline in a JavaScript string? Perhaps you wanted to embed some lyrics? You would have needed to escape the literal newline using our favorite escape character \, and then put a new line into the string manually \n at the next line. This is shown below:

```
var lyrics = "Never gonna give you up \\n Never gonna let you down";
```

With TypeScript you can just use a template string:

```
var lyrics = `Never gonna give you up
            Never gonna let you down`;
```

Tagged Templates…

- You can place a function (called a tag) before the template string and it gets the opportunity to pre process the template string literals plus the values of all the placeholder expressions and return a result. A few notes:
- All the static literals are passed in as an array for the first argument.

Tagged Templates…

- Here is an example where we have a tag function (named htmlEscape) that escapes the html from all the placeholders:

  var say = "a bird in hand > two in the bush";

  var html = htmlEscape `<div> I would just like to say : ${say}</div>`;

```
// a sample tag function
function htmlEscape(literals, ...placeholders) {
    let result = "";

    // interleave the literals with the placeholders
    for (let i = 0; i < placeholders.length; i++) {
        result += literals[i];
        result += placeholders[i]
            .replace(/&/g, '&amp;')
            .replace(/"/g, '&quot;')
            .replace(/'/g, '&#39;')
            .replace(/</g, '&lt;')
            .replace(/>/g, '&gt;');
    }
    // add the last literal
    result += literals[literals.length - 1];
    return result;
}
```

Object de-structuring

- Destructuring is a way of extracting values into variables from data stored in objects and arrays.
- Let’s imagine we have an object like so:

```
const obj = {first: 'Asim', last: 'Hussain', age: 39 };
```

- We want to extract the first and last properties into local variables, prior to ES6 we would have to write something like this:

```
const f = obj.first;
const l = obj.last;
console.log(f); // Asim
console.log(l); // Hussain
```

- With destructing we can do so in one line, like so:

```
const {first: f, last: l} = obj;
console.log(f); // Asim
console.log(l); // Hussain
```

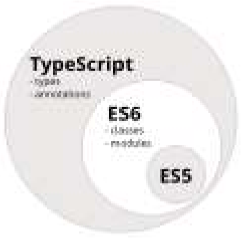
Add instructor notes here.

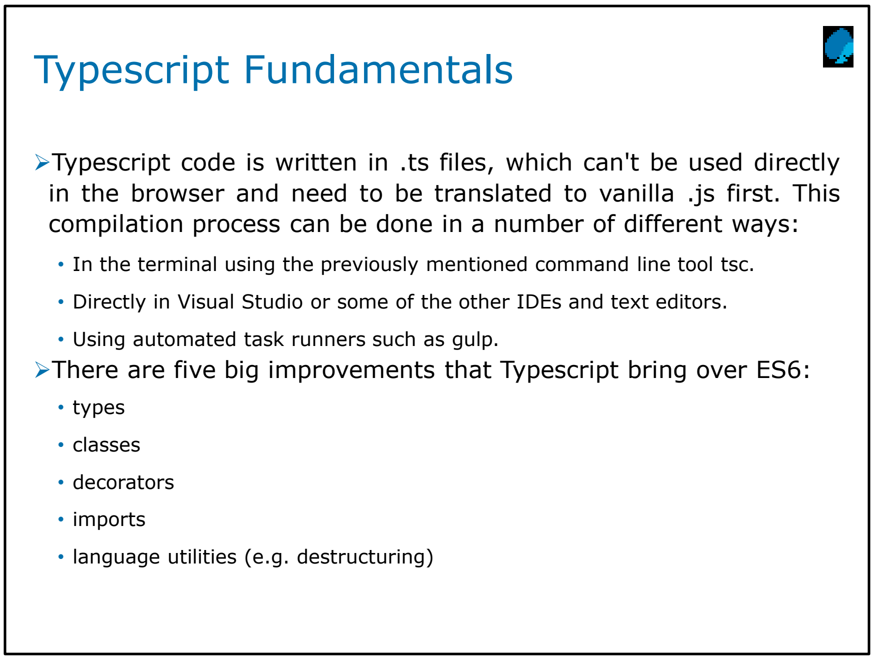
References

- http://es6-features.org
- https://developer.mozilla.org/bm/docs/Web/JavaScript

Typescript…

- Typescript is a Strongly typed, object oriented, compiled language.
- Typescript is a typed superset of JavaScript compiled to JavaScript. In other words, Typescript is JavaScript plus some additional features.
- Angular is built in a JavaScript-like language called Typescript.
- Typescript isn’t a completely new language, it’s a superset of ES6. If we write ES6 code, it’s perfectly valid and compliable Typescript code



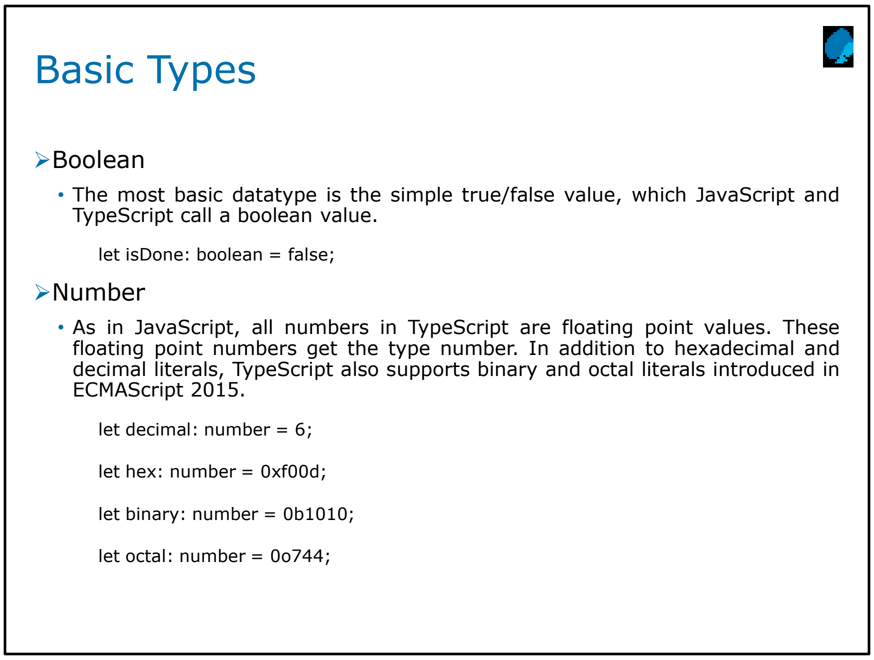


Typescript Fundamentals

- ➢Typescript code is written in .ts files, which can't be used directly in the browser and need to be translated to vanilla .js first. This compilation process can be done in a number of different ways:
  - In the terminal using the previously mentioned command line tool tsc.
  - Directly in Visual Studio or some of the other IDEs and text editors.
  - Using automated task runners such as gulp.
- ➢There are five big improvements that Typescript bring over ES6:
  - types
  - classes
  - decorators
  - imports
  - language utilities (e.g. destructuring)

Types

- ➢The major improvement of TypeScript over ES6, that gives the language its name, is the typing system. For some people the lack of type checking is considered one of the benefits of using a language like JavaScript.
  - It Helps When Writing Code Because It Can Prevent Bugs at Compile Time And
  - It helps when reading code because it clarifies your intentions



Basic Types

- Boolean
  - The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.

```
let isDone: boolean = false;
```

- Number
  - As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type number. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

```
let decimal: number = 6;

let hex: number = 0xf00d;

let binary: number = 0b1010;

let octal: number = 0o744;
```

Basic Types

- String
  - Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type string to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data.

```
let color: string = "blue";
color = 'red';
```

- You can also use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

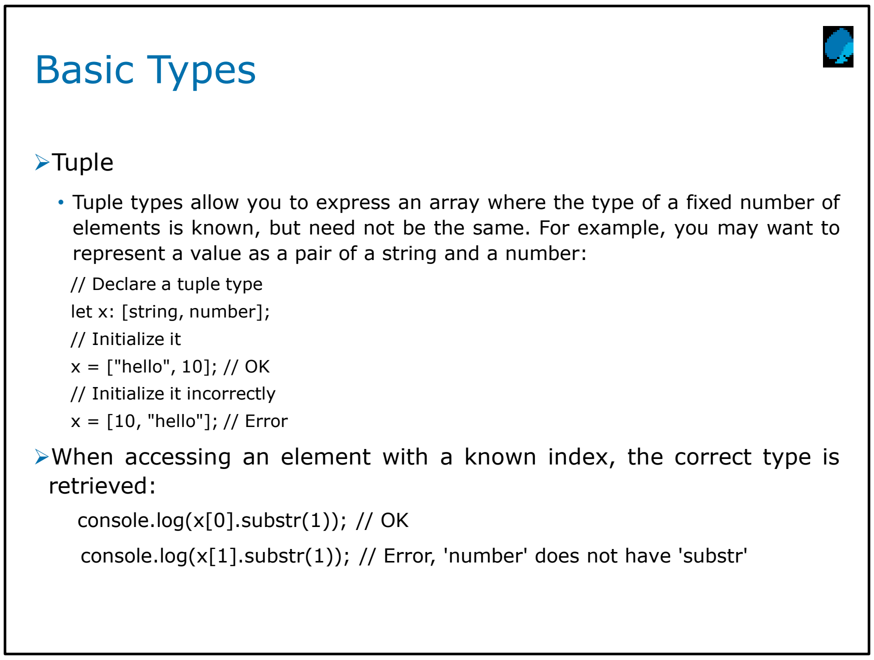
Basic Types

- Array
  - TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type:

```
let list: number[] = [1, 2, 3];
```

- The second way uses a generic array type, Array<elemType>:

```
let list: Array<number> = [1, 2, 3];
```



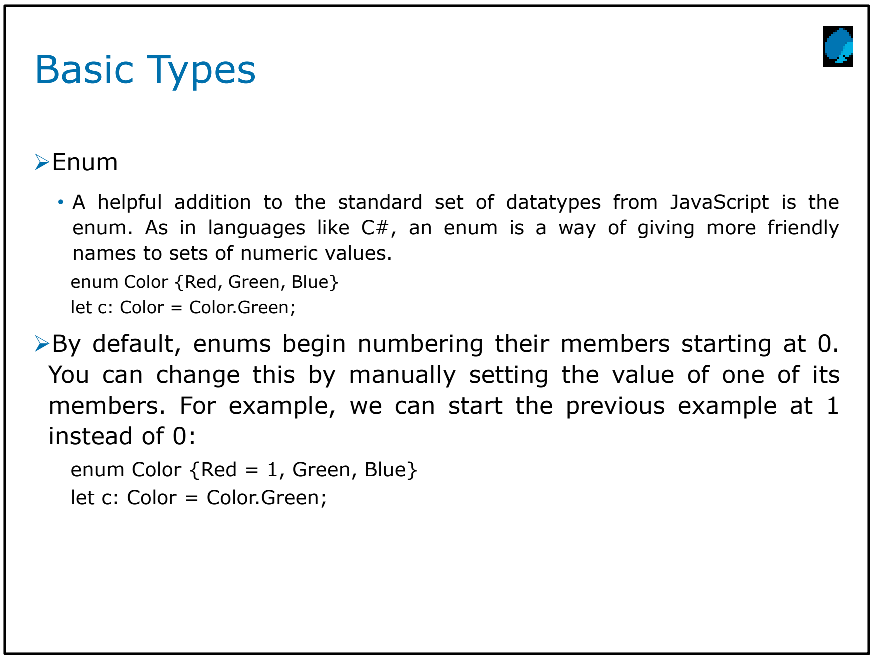
Basic Types

- Tuple
  - Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same. For example, you may want to represent a value as a pair of a string and a number:

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

- When accessing an element with a known index, the correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```



Basic Types

- Enum
  - A helpful addition to the standard set of datatypes from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

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

- By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

Basic Types

- Any
  - We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the any type:

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

  - The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. You might expect Object to play a similar role, as it does in other languages. But variables of type Object only allow you to assign any value to them - you can’t call arbitrary methods on them, even ones that actually exist:

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)
let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

  - The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

```
let list: any[] = [1, true, "free"];
list[1] = 100;
```

Basic Types

- Void
  - void is a little like the opposite of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
    console.log("This is my warning message");
}
```

- Declaring variables of type void is not useful because you can only assign undefined or null to them:

```
let unusable: void = undefined;
```

Basic Types

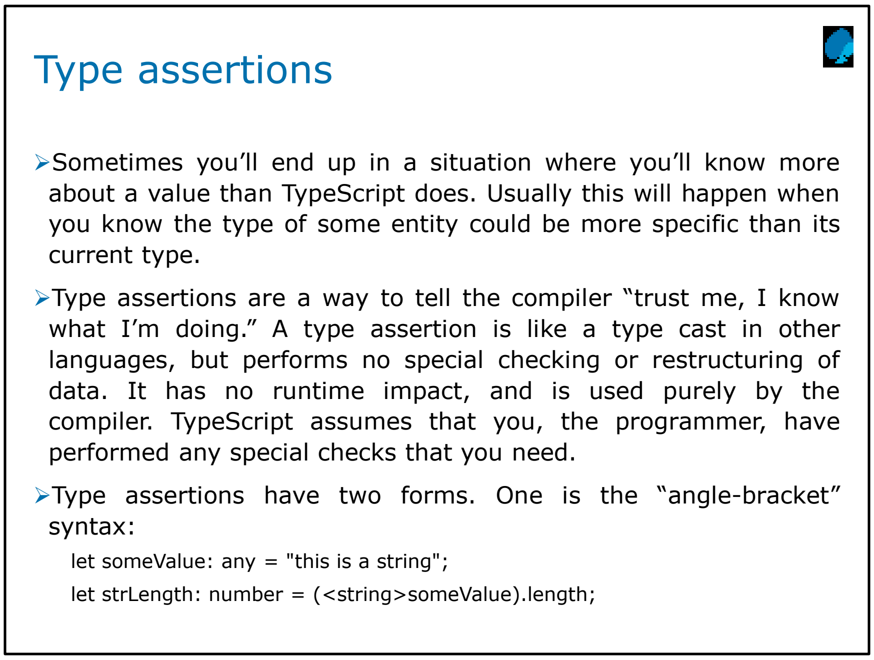
- Null and Undefined
  - In TypeScript, both undefined and null actually have their own types named undefined and null respectively. Much like void, they're not extremely useful on their own:
  - // Not much else we can assign to these variables!

```
let u: undefined = undefined;
let n: null = null;
```

- By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number.

Basic Types

- Never
  - The never type represents the type of values that never occur. For instance, never is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type never when narrowed by any type guards that can never be true.
  - The never type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, never (except never itself). Even any isn’t assignable to never.



Type assertions

- Sometimes you'll end up in a situation where you'll know more about a value than TypeScript does. Usually this will happen when you know the type of some entity could be more specific than its current type.
- Type assertions are a way to tell the compiler "trust me, I know what I'm doing." A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.
- Type assertions have two forms. One is the "angle-bracket" syntax:

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
```

Classes

- Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components,
- Classes inherit functionality and objects are built from these classes.
- In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript

Classes

- Let’s take a look at a simple class-based example:

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

let greeter = new Greeter("world");
```

Inheritance

- In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}
class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}
const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

Public, private, and protected modifiers

- Public by default
  - In our examples, we've been able to freely access the members that we declared throughout our programs
  - In TypeScript, each member is public by default.
- You may still mark a member public explicitly. We could have written the Animal class from the previous section in the following way:

```
class Animal {
    public name: string;
    public constructor(theName: string) { this.name = theName; }
    public move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

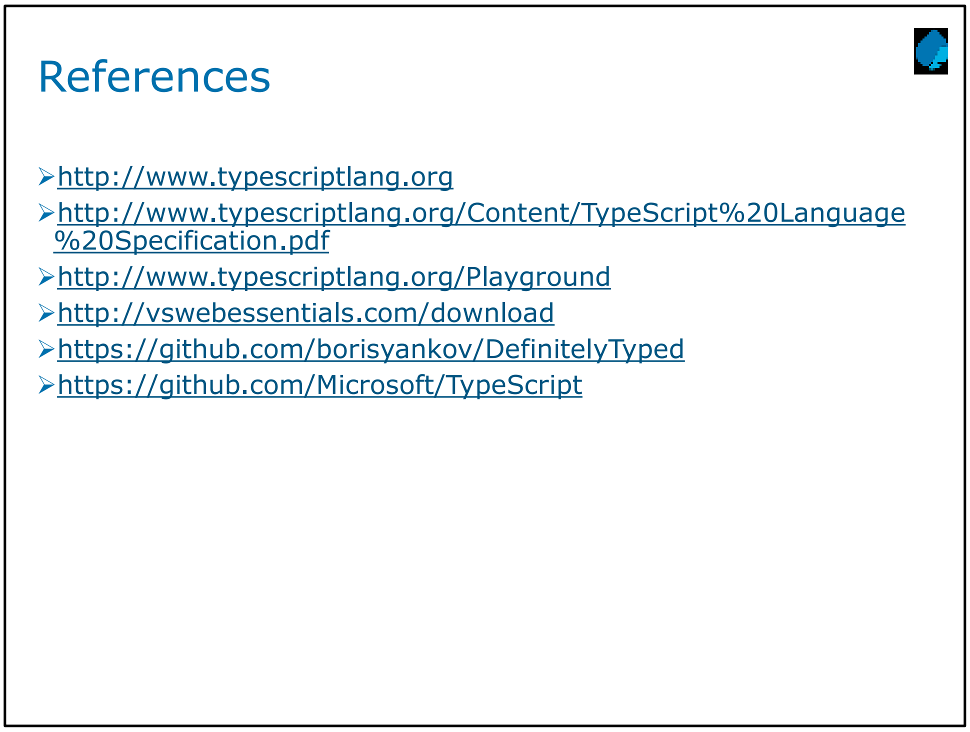
Readonly modifier

- You can make properties readonly by using the readonly keyword. Readonly properties must be initialized at their declaration or in the constructor.

```
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error! name is readonly.
```

Summary

- Types & type assertions
- Typescript OOPS - Classes, Interfaces, Constructor, etc
- Creating custom types
- Decorators



References

- http://www.typescriptlang.org
- http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf
- http://www.typescriptlang.org/Playground
- http://vswebessentials.com/download
- https://github.com/borisyankov/DefinitelyTyped
- https://github.com/Microsoft/TypeScript

Add instructor notes here.