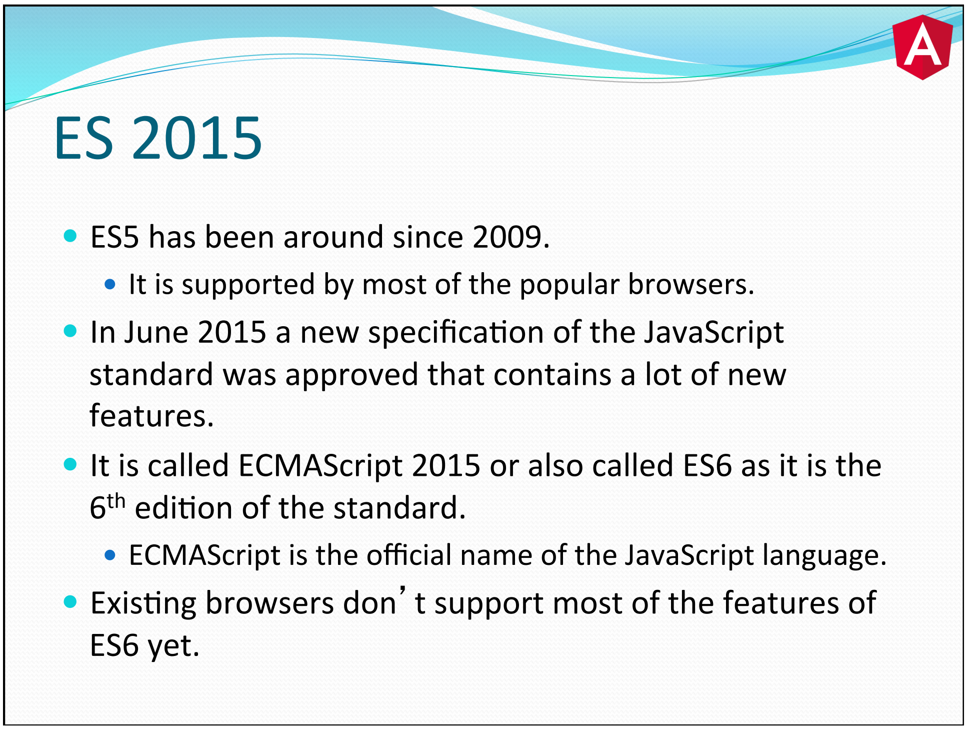
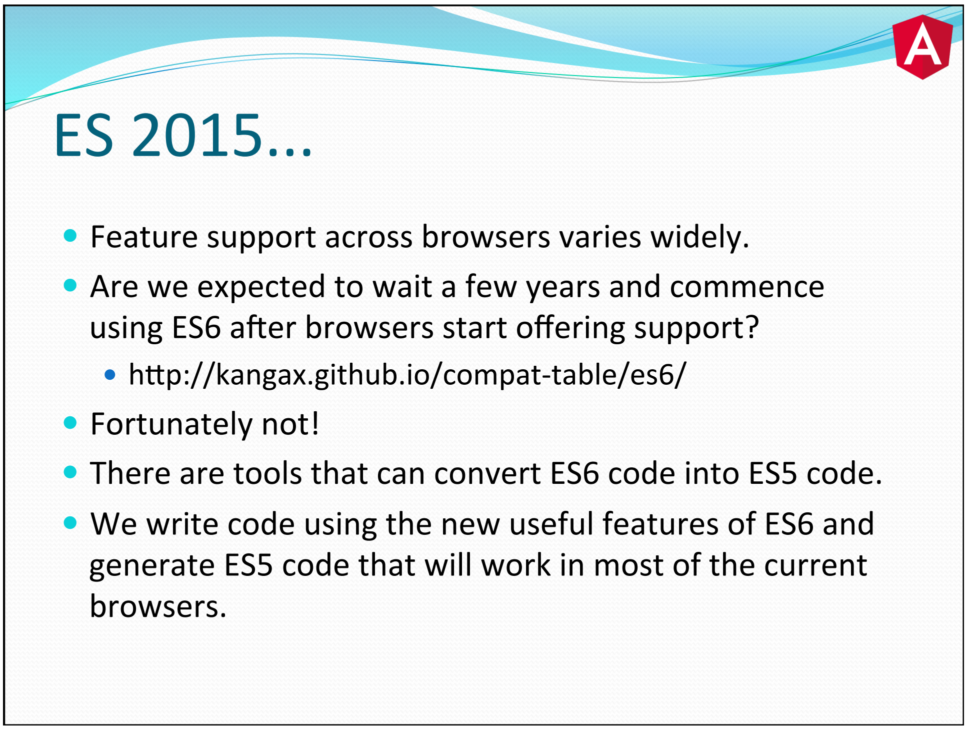


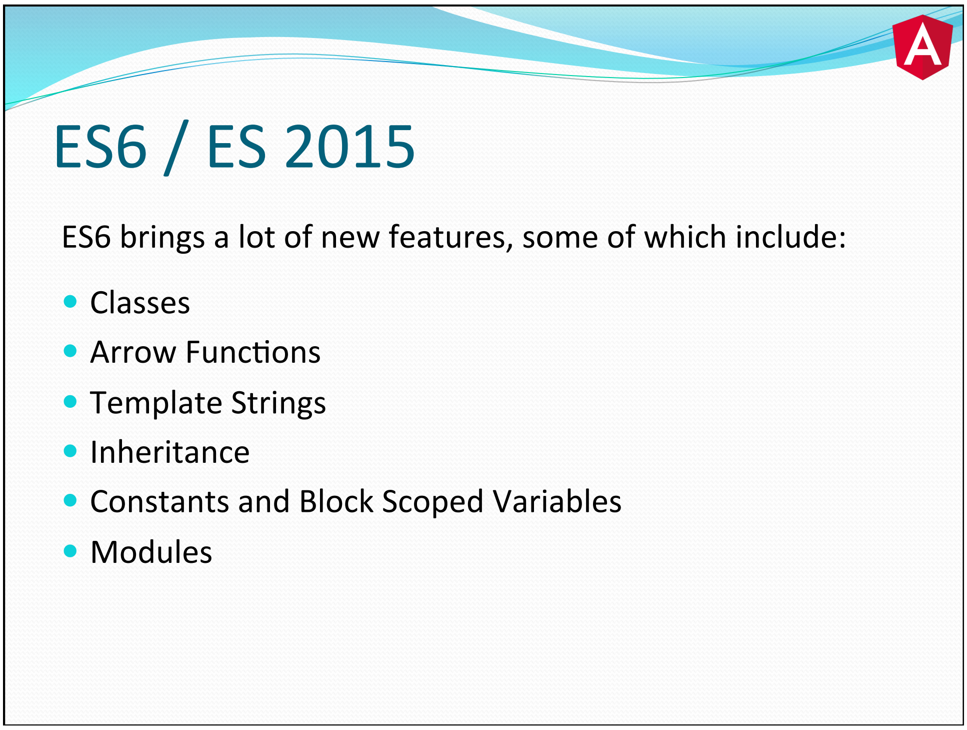
ES 2015

- ES5 has been around since 2009.
  - It is supported by most of the popular browsers.
- In June 2015 a new specification of the JavaScript standard was approved that contains a lot of new features.
- It is called ECMAScript 2015 or also called ES6 as it is the 6th edition of the standard.
  - ECMAScript is the official name of the JavaScript language.
- Existing browsers don't support most of the features of ES6 yet.



ES 2015...

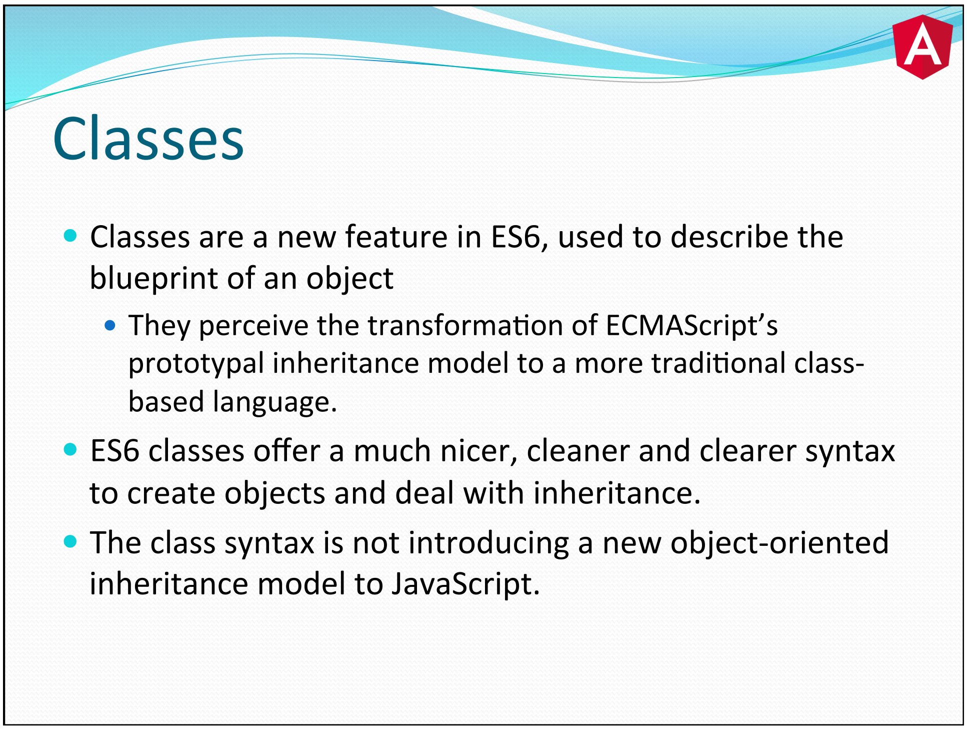
- Feature support across browsers varies widely.
- Are we expected to wait a few years and commence using ES6 after browsers start offering support?
  - http://kangax.github.io/compat-table/es6/
- Fortunately not!
- There are tools that can convert ES6 code into ES5 code.
- We write code using the new useful features of ES6 and generate ES5 code that will work in most of the current browsers.



ES6 / ES 2015

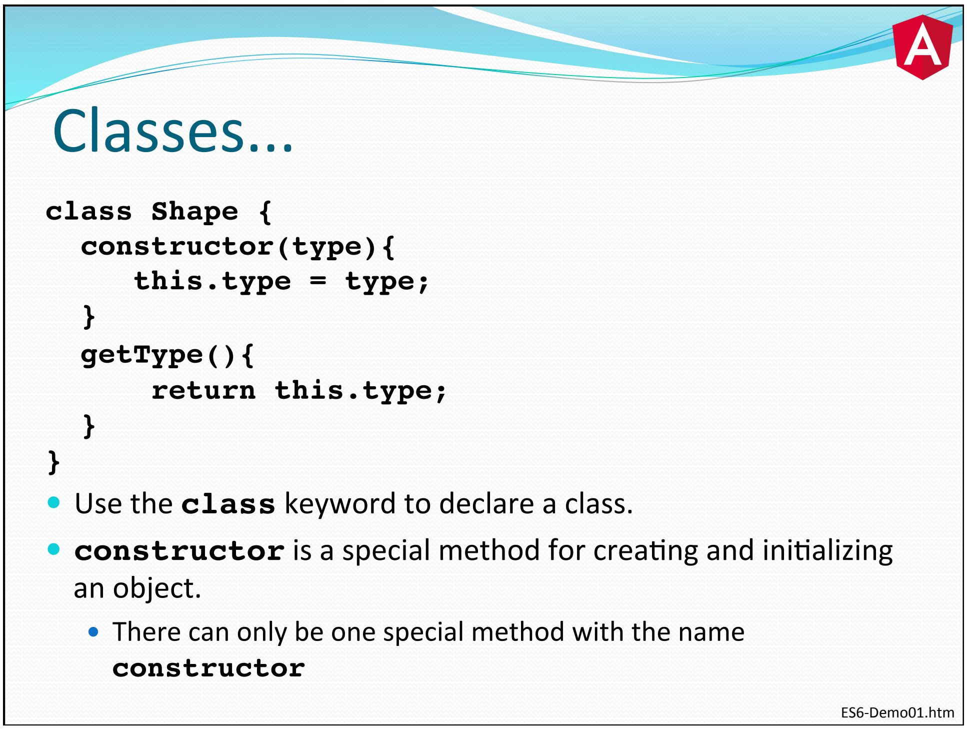
ES6 brings a lot of new features, some of which include:

- Classes
- Arrow Functions
- Template Strings
- Inheritance
- Constants and Block Scoped Variables
- Modules



Classes

- Classes are a new feature in ES6, used to describe the blueprint of an object
  - They perceive the transformation of ECMAScript’s prototypal inheritance model to a more traditional class-based language.
- ES6 classes offer a much nicer, cleaner and clearer syntax to create objects and deal with inheritance.
- The class syntax is not introducing a new object-oriented inheritance model to JavaScript.

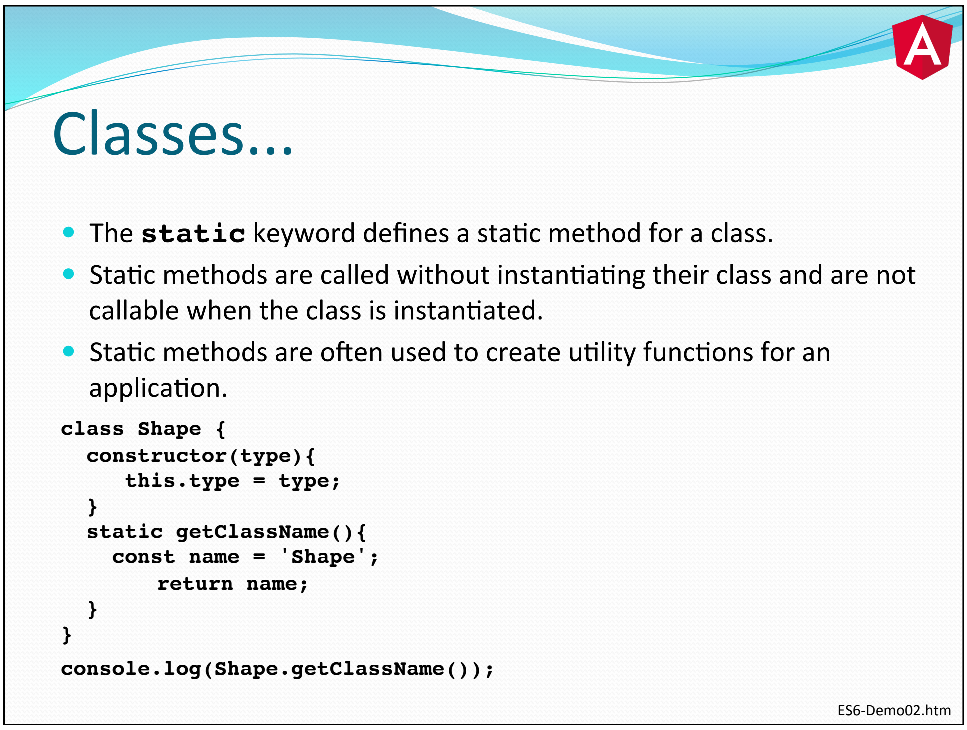


Classes...

```
class Shape {
  constructor(type){
     this.type = type;
  }
  getType(){
      return this.type;
  }
}
```

- Use the **`class`** keyword to declare a class.
- **`constructor`** is a special method for creating and initializing an object.
  - There can only be one special method with the name **`constructor`**

ES6-Demo01.htm

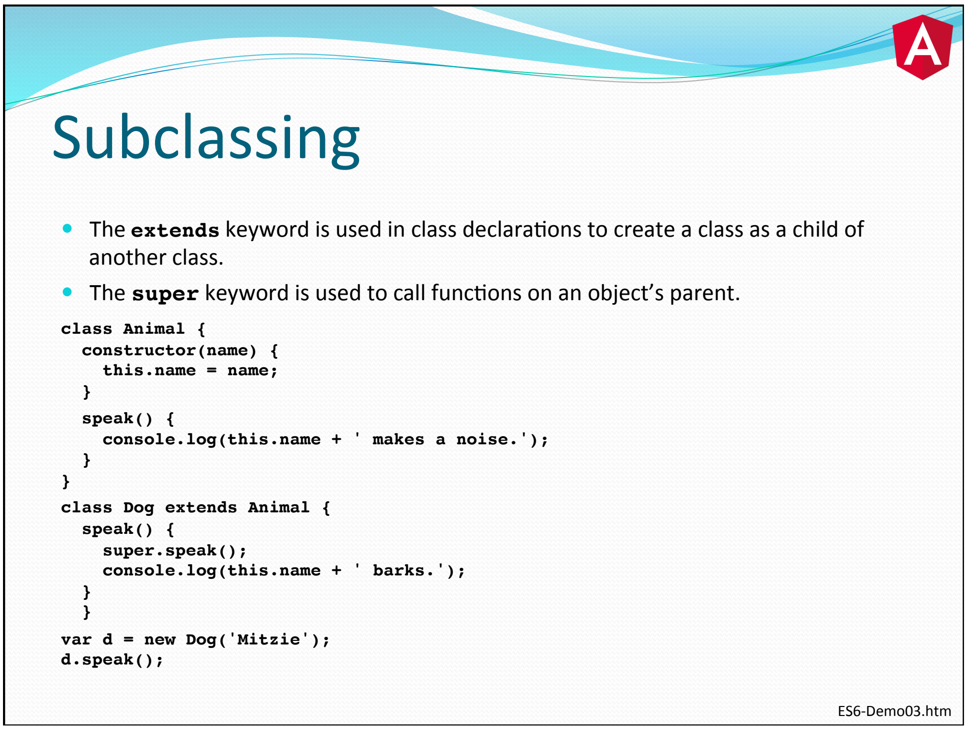


Classes...

- The **`static`** keyword defines a static method for a class.
- Static methods are called without instantiating their class and are not callable when the class is instantiated.
- Static methods are often used to create utility functions for an application.

```
class Shape {
  constructor(type){
     this.type = type;
  }
  static getClassName(){
    const name = 'Shape';
       return name;
  }
}
console.log(Shape.getClassName());
```

ES6-Demo02.htm

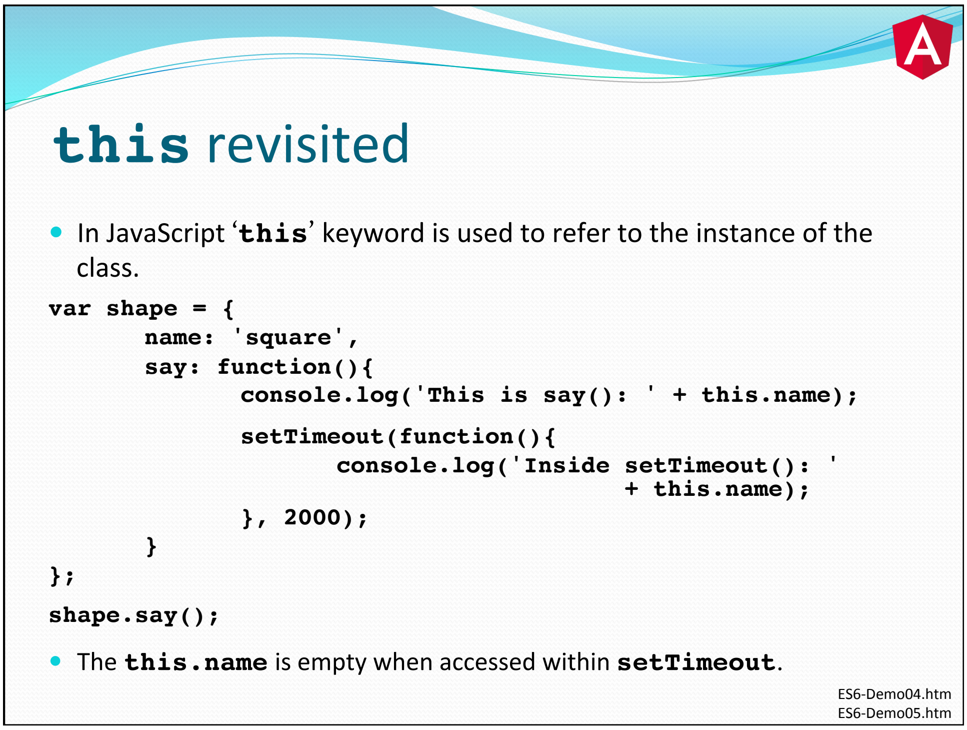


Subclassing

- The **extends** keyword is used in class declarations to create a class as a child of another class.
- The **super** keyword is used to call functions on an object’s parent.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' makes a noise.');
  }
}
class Dog extends Animal {
  speak() {
    super.speak();
    console.log(this.name + ' barks.');
  }
  }
var d = new Dog('Mitzie');
d.speak();
```

ES6-Demo03.htm



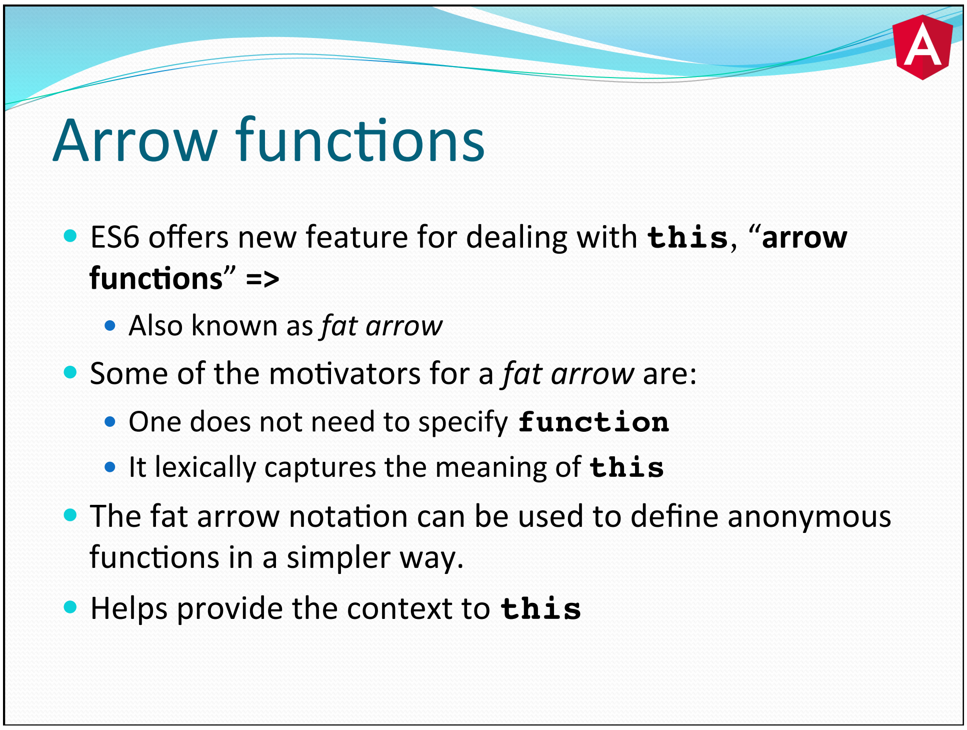
this revisited

- In JavaScript ‘**this**’ keyword is used to refer to the instance of the class.

```
var shape = {
       name: 'square',
       say: function(){
              console.log('This is say(): ' + this.name);
              setTimeout(function(){
                     console.log('Inside setTimeout(): '
                                         + this.name);
              }, 2000);
       }
};
shape.say();
```

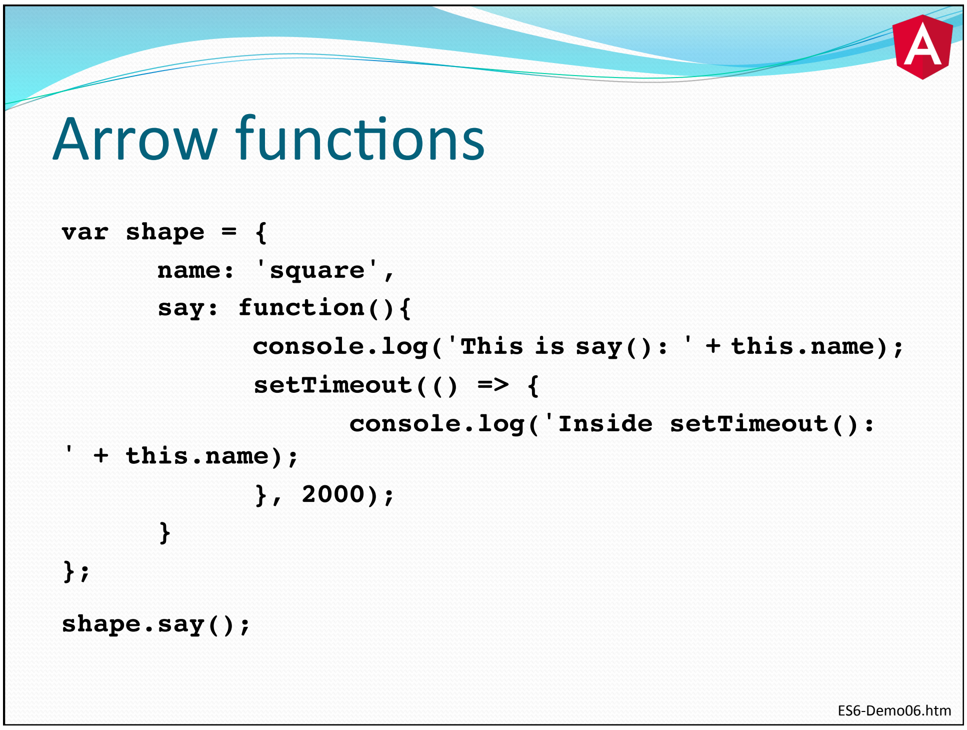
- The **this.name** is empty when accessed within **setTimeout**.

ES6-Demo04.htm
ES6-Demo05.htm



Arrow functions

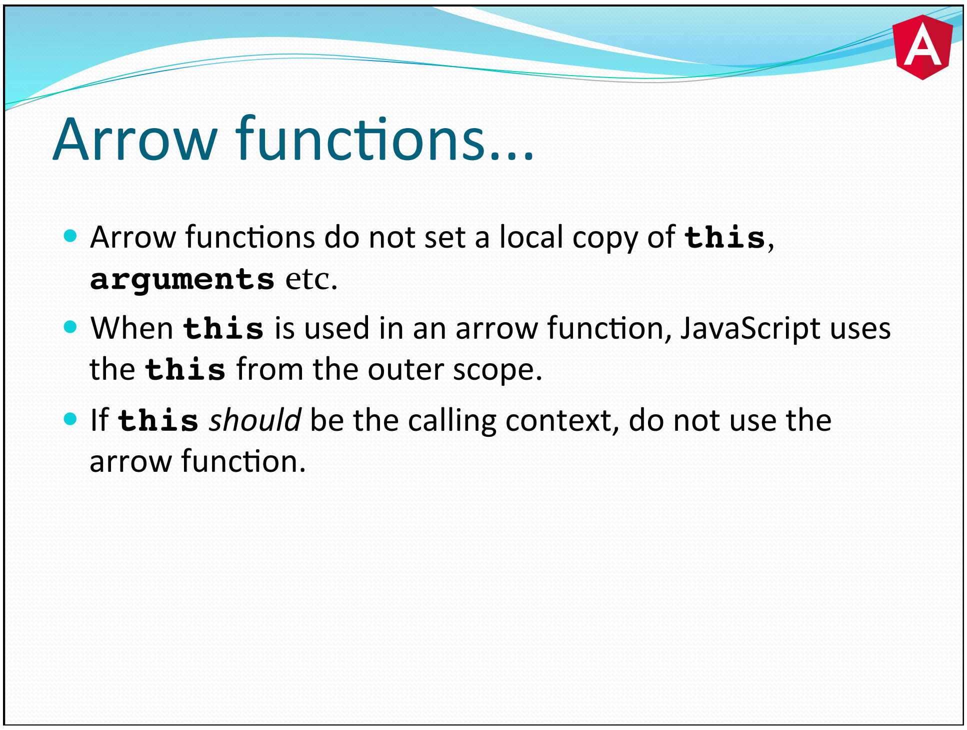
- ES6 offers new feature for dealing with **`this`**, "**arrow functions**" **=>**
  - Also known as *fat arrow*
- Some of the motivators for a *fat arrow* are:
  - One does not need to specify **`function`**
  - It lexically captures the meaning of **`this`**
- The fat arrow notation can be used to define anonymous functions in a simpler way.
- Helps provide the context to **`this`**



Arrow functions

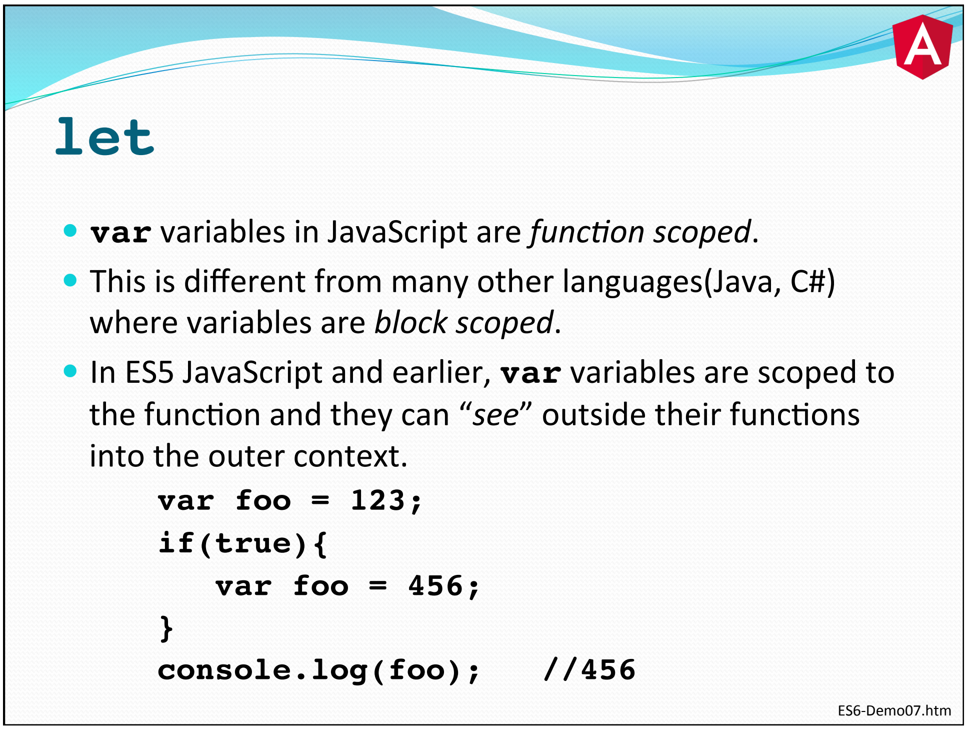
```
var shape = {
      name: 'square',
      say: function(){
            console.log('This is say(): ' + this.name);
            setTimeout(() => {
                  console.log('Inside setTimeout():
' + this.name);
            }, 2000);
      }
};
shape.say();
```

ES6-Demo06.htm



Arrow functions...

- Arrow functions do not set a local copy of **`this`**, **`arguments`** etc.
- When **`this`** is used in an arrow function, JavaScript uses the **`this`** from the outer scope.
- If **`this`** *should* be the calling context, do not use the arrow function.



`let`

- **`var`** variables in JavaScript are *function scoped*.
- This is different from many other languages(Java, C#) where variables are *block scoped*.
- In ES5 JavaScript and earlier, **`var`** variables are scoped to the function and they can "*see*" outside their functions into the outer context.

```
var foo = 123;
if(true){
    var foo = 456;
}
console.log(foo);   //456
```

ES6-Demo07.htm

let…

- ES6 introduces the **`let`** keyword to allow defining variables with true *block scope*.
- Use of the **`let`** instead of **`var`** gives a true unique element disconnected from what is defined outside the scope.

```
let foo = 123;
if(true){
 let foo = 456;
}
console.log(foo);  //123
```

ES6-Demo08.htm

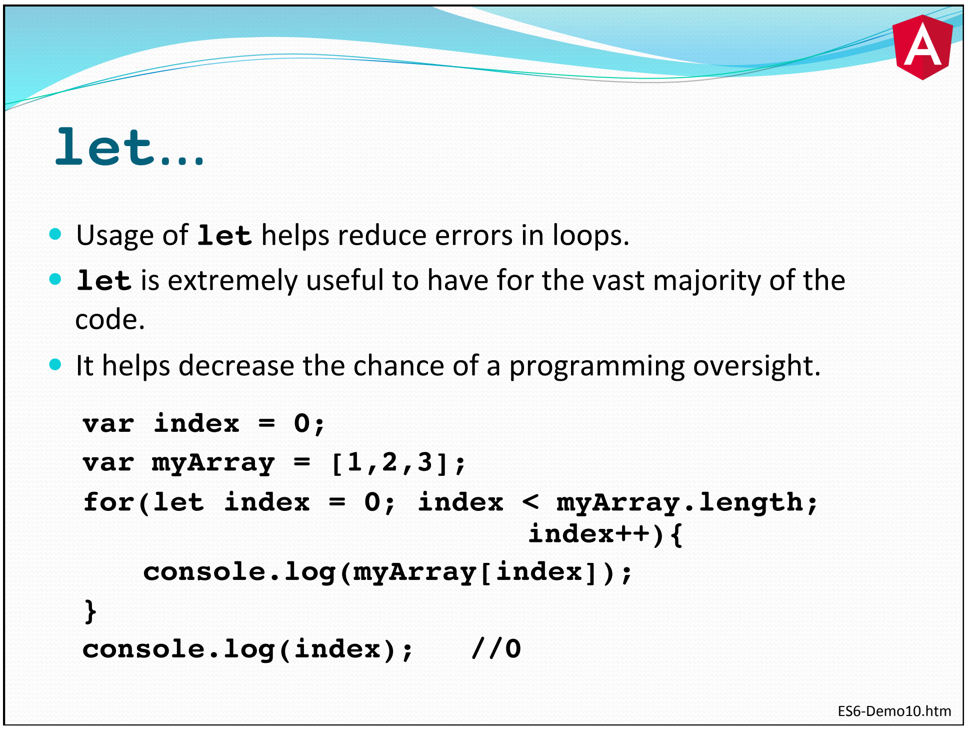
let…

- Functions create a new variable scope in JavaScript as expected.

```
var num = 123;
function numbers(){
   var num = 456;
}

numbers();
console.log(num);   //123
```

ES6-Demo09.htm

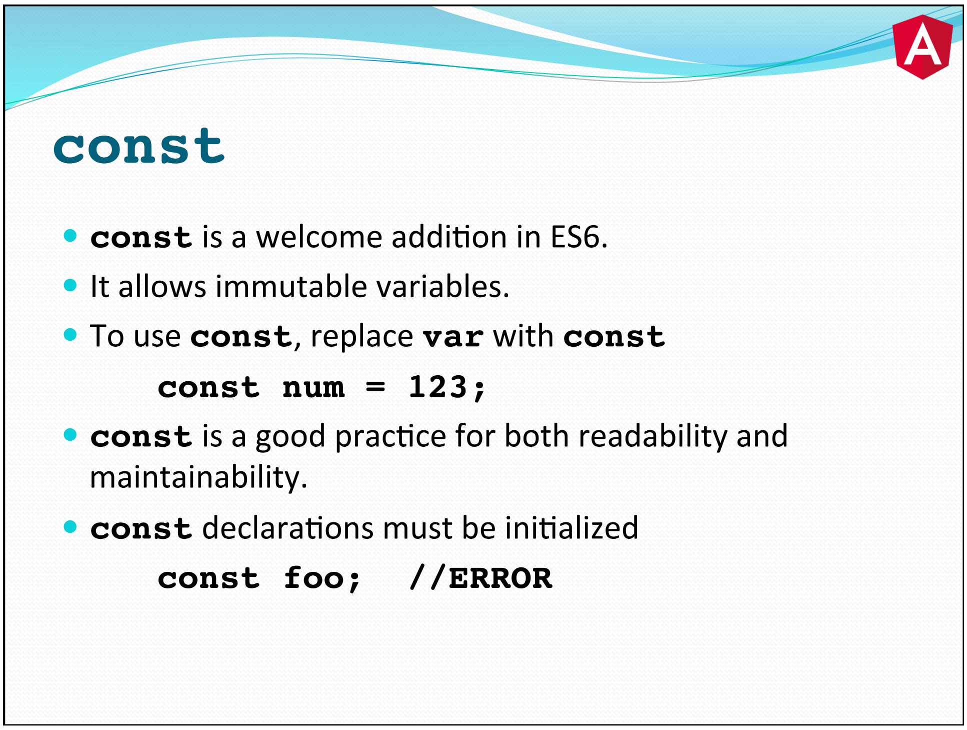


let…

- Usage of **let** helps reduce errors in loops.
- **let** is extremely useful to have for the vast majority of the code.
- It helps decrease the chance of a programming oversight.

```
var index = 0;
var myArray = [1,2,3];
for(let index = 0; index < myArray.length;
                             index++){
    console.log(myArray[index]);
}
console.log(index);   //0
```

ES6-Demo10.htm



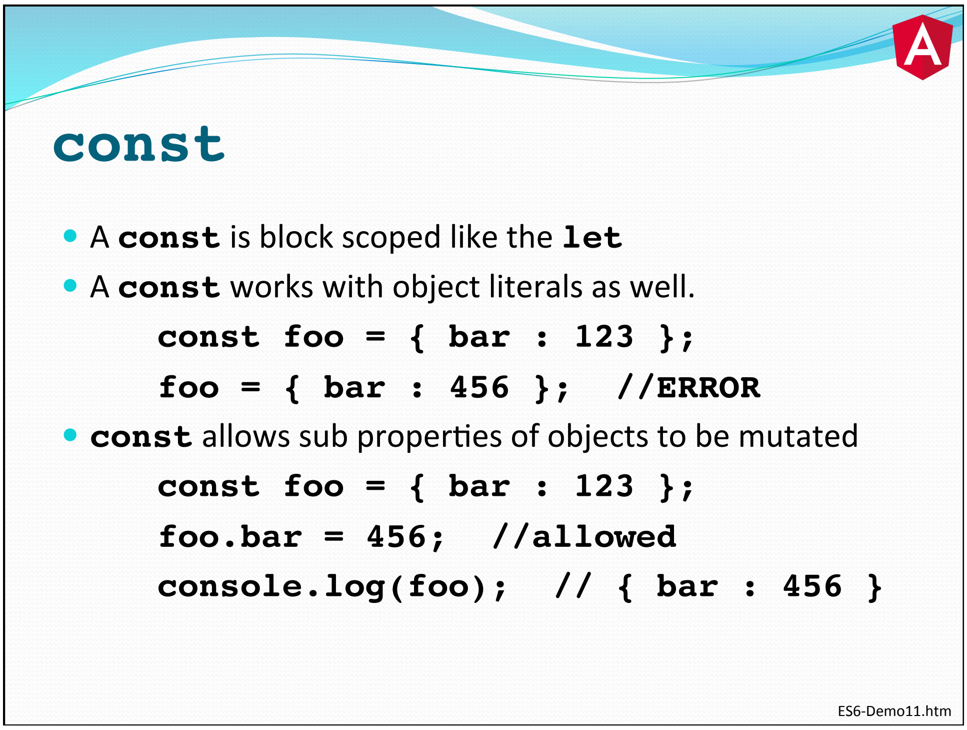
const

- **const** is a welcome addition in ES6.
- It allows immutable variables.
- To use **const**, replace **var** with **const**

```
const num = 123;
```

- **const** is a good practice for both readability and maintainability.
- **const** declarations must be initialized

```
const foo;   //ERROR
```



const

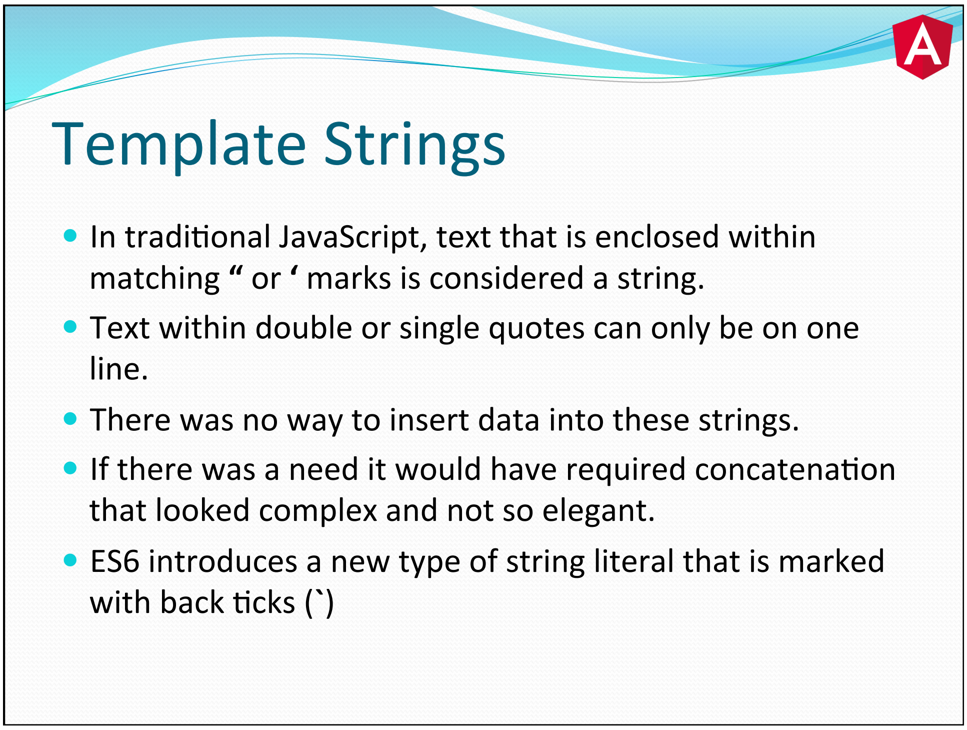
- A **const** is block scoped like the **let**
- A **const** works with object literals as well.

```
const foo = { bar : 123 };
foo = { bar : 456 };   //ERROR
```

- **const** allows sub properties of objects to be mutated

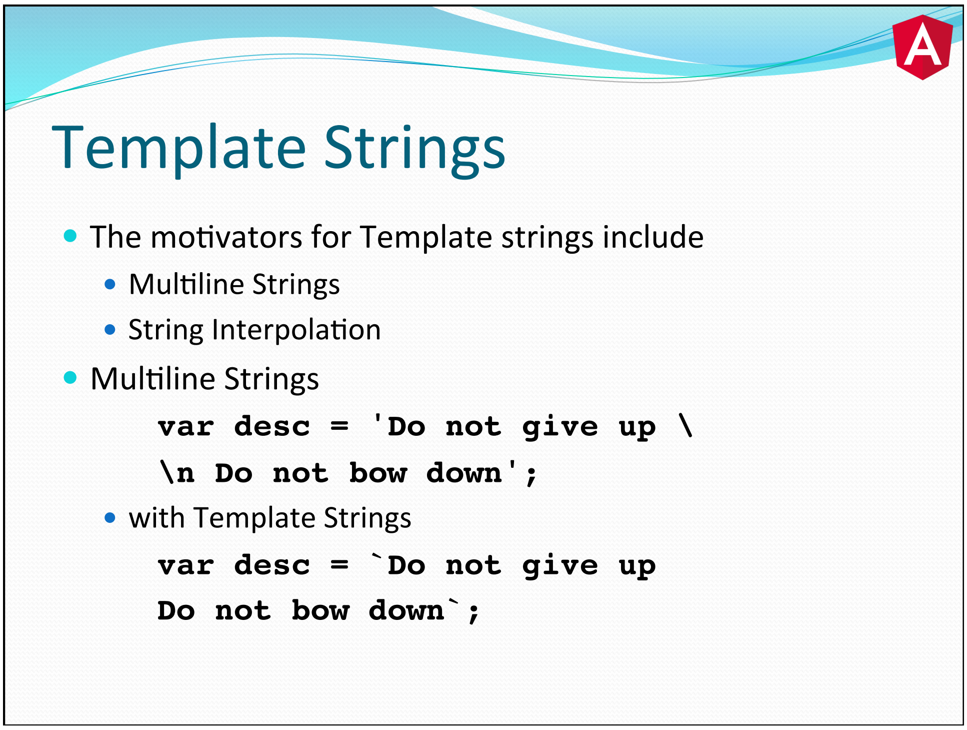
```
const foo = { bar : 123 };
foo.bar = 456;   //allowed
console.log(foo);   // { bar : 456 }
```

ES6-Demo11.htm



Template Strings

- In traditional JavaScript, text that is enclosed within matching " or ' marks is considered a string.
- Text within double or single quotes can only be on one line.
- There was no way to insert data into these strings.
- If there was a need it would have required concatenation that looked complex and not so elegant.
- ES6 introduces a new type of string literal that is marked with back ticks (`)



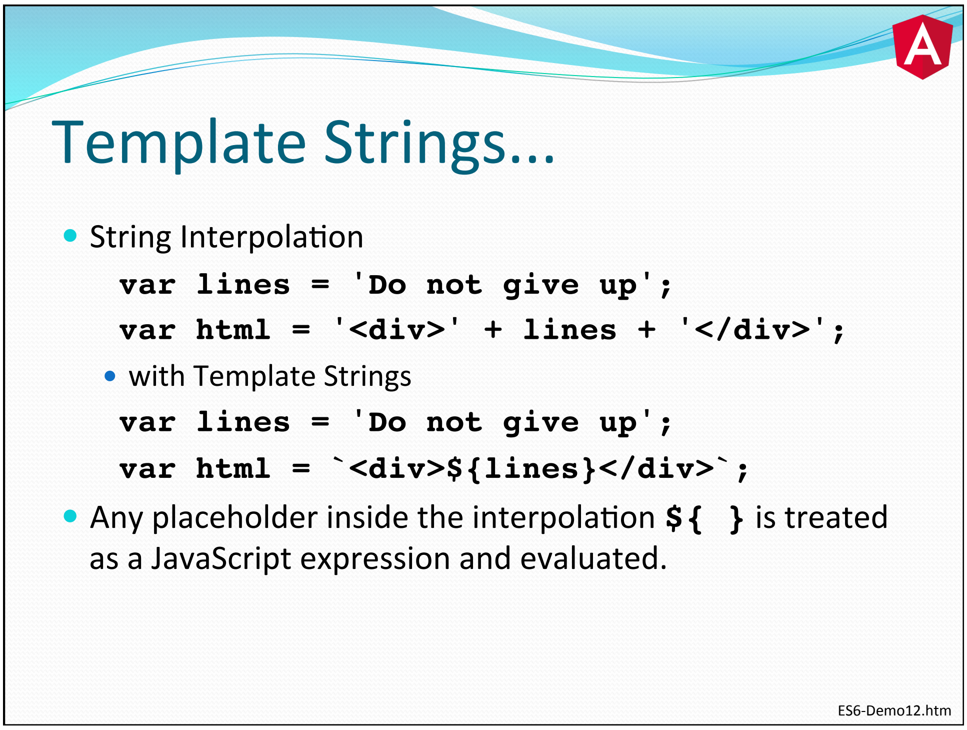
Template Strings

- The motivators for Template strings include
  - Multiline Strings
  - String Interpolation
- Multiline Strings

```
var desc = 'Do not give up \
\n Do not bow down';
```

  - with Template Strings

```
var desc = `Do not give up
Do not bow down`;
```



Template Strings...

- String Interpolation

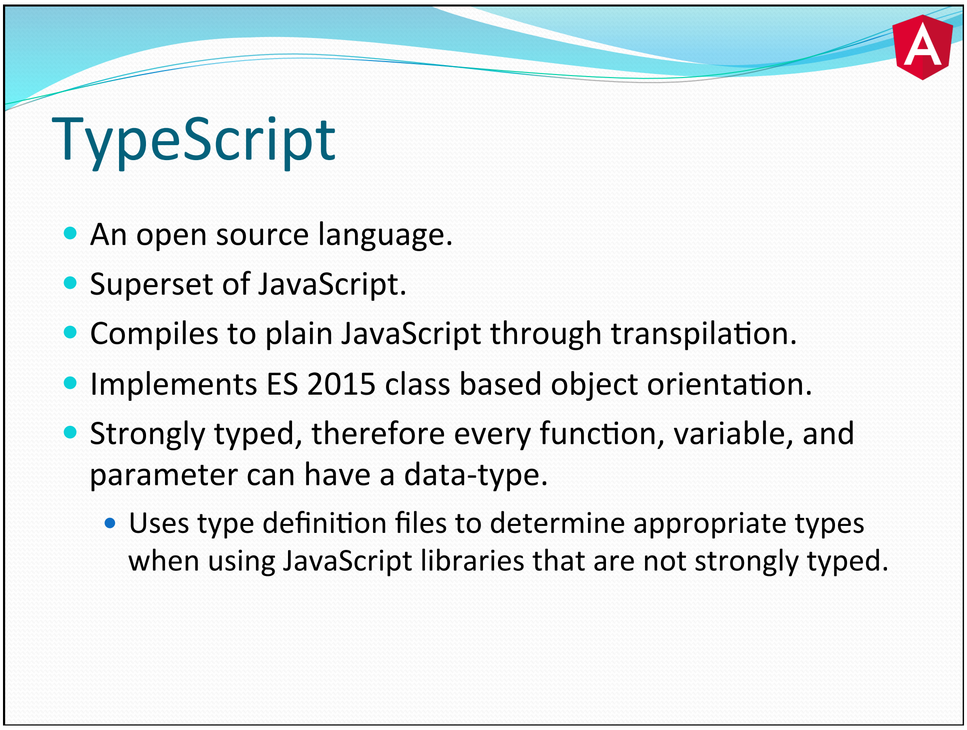
```
var lines = 'Do not give up';
var html = '<div>' + lines + '</div>';
```

  - with Template Strings

```
var lines = 'Do not give up';
var html = `<div>${lines}</div>`;
```

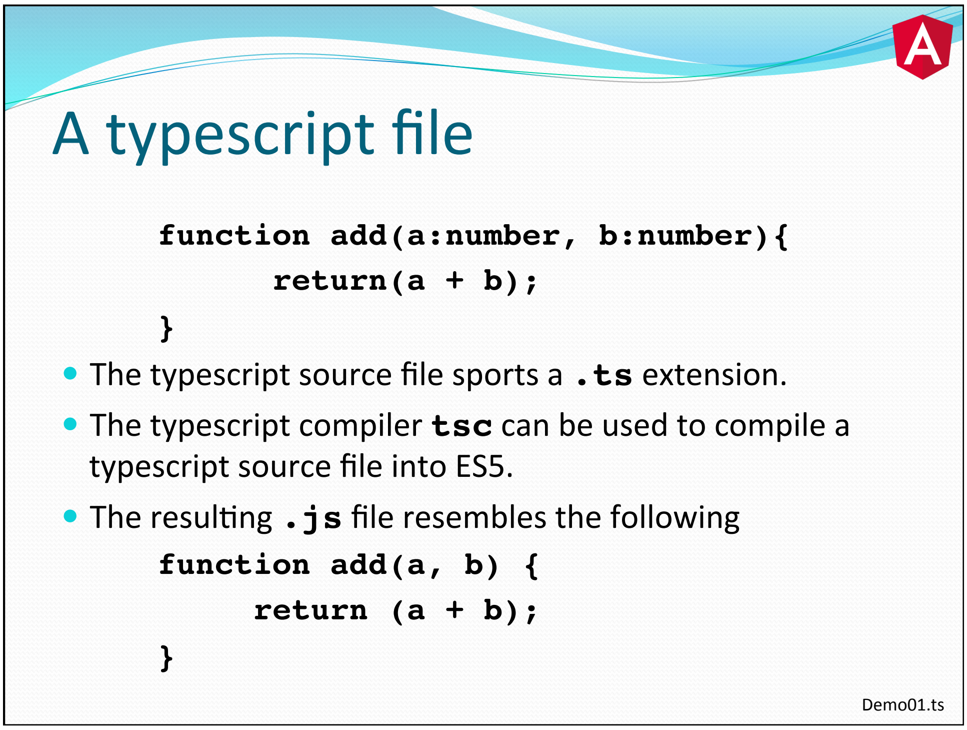
- Any placeholder inside the interpolation **`${  }`** is treated as a JavaScript expression and evaluated.

ES6-Demo12.htm



TypeScript

- An open source language.
- Superset of JavaScript.
- Compiles to plain JavaScript through transpilation.
- Implements ES 2015 class based object orientation.
- Strongly typed, therefore every function, variable, and parameter can have a data-type.
  - Uses type definition files to determine appropriate types when using JavaScript libraries that are not strongly typed.



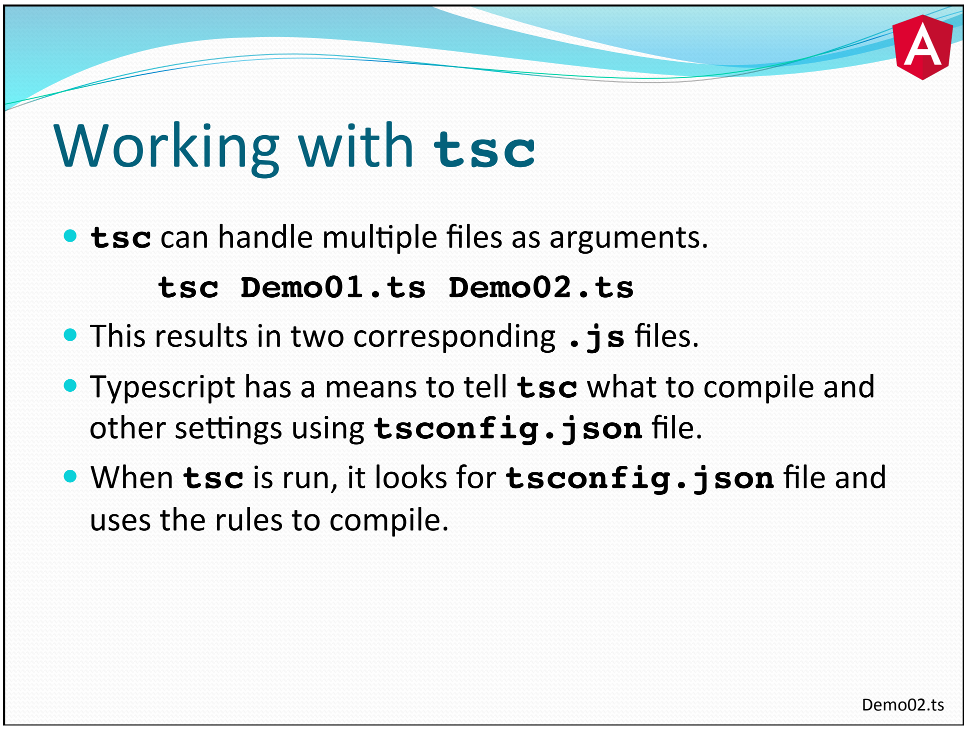
A typescript file

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

- The typescript source file sports a **.ts** extension.
- The typescript compiler **tsc** can be used to compile a typescript source file into ES5.
- The resulting **.js** file resembles the following

```
function add(a, b) {
      return (a + b);
}
```

Demo01.ts



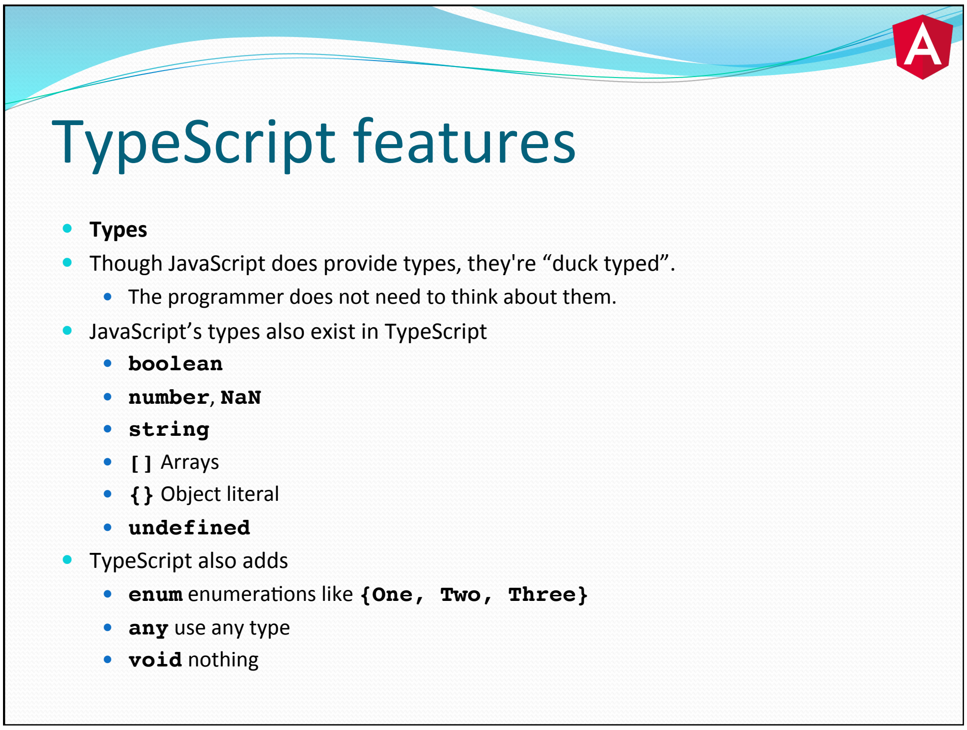
Working with `tsc`

- **`tsc`** can handle multiple files as arguments.

```
tsc Demo01.ts Demo02.ts
```

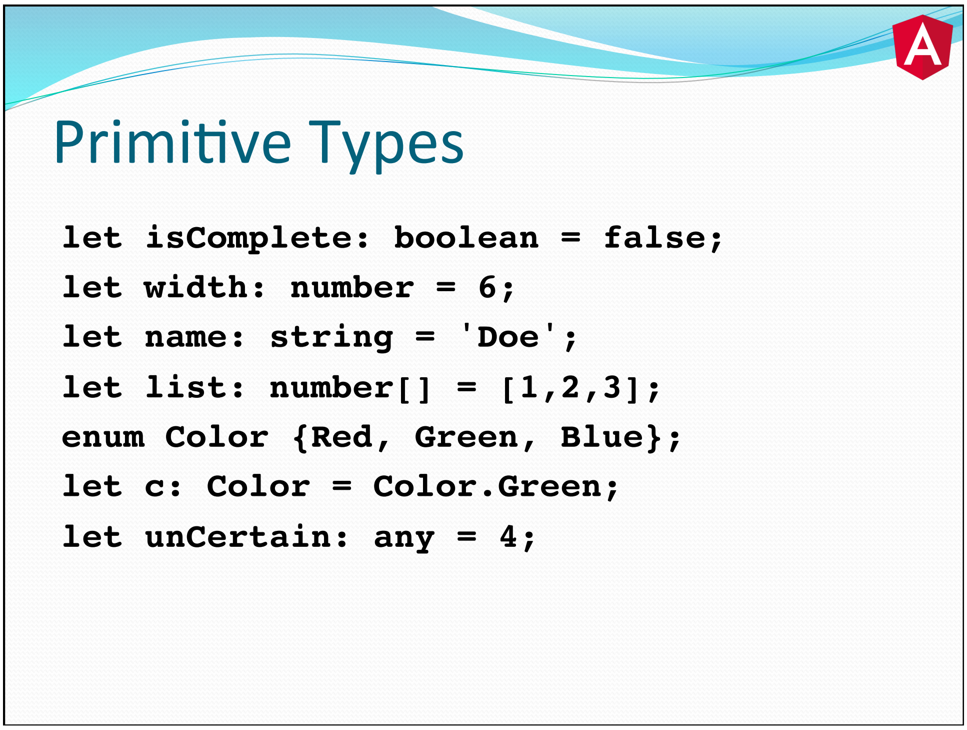
- This results in two corresponding **`.js`** files.
- Typescript has a means to tell **`tsc`** what to compile and other settings using **`tsconfig.json`** file.
- When **`tsc`** is run, it looks for **`tsconfig.json`** file and uses the rules to compile.

Demo02.ts



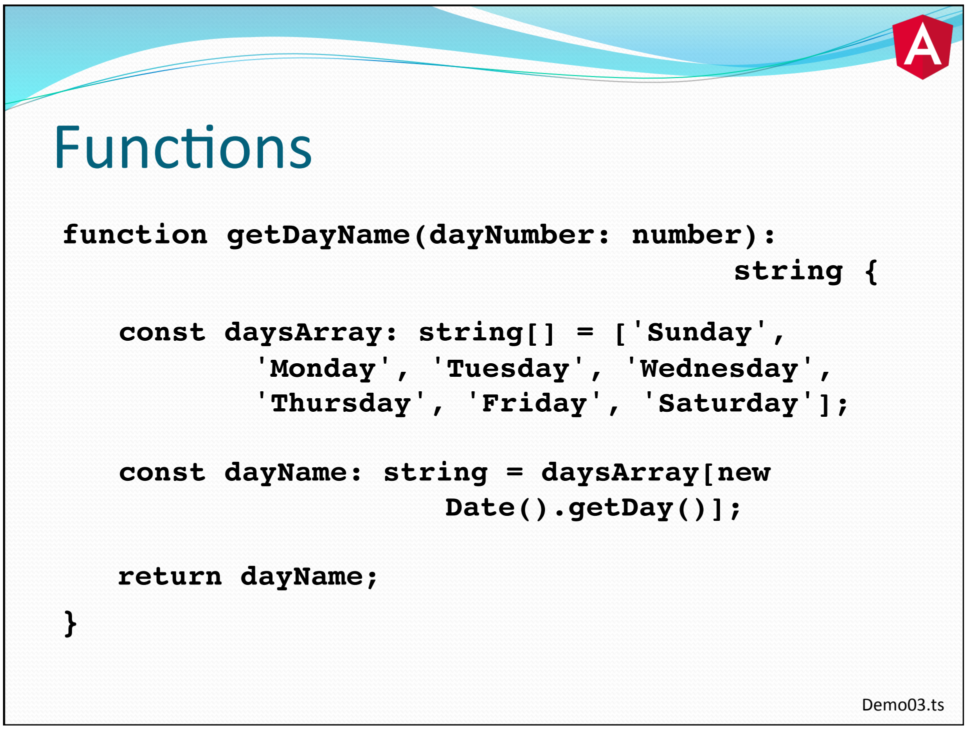
TypeScript features

- **Types**
- Though JavaScript does provide types, they're "duck typed".
  - The programmer does not need to think about them.
- JavaScript's types also exist in TypeScript
  - **`boolean`**
  - **`number`**, **`NaN`**
  - **`string`**
  - **`[]`** Arrays
  - **`{}`** Object literal
  - **`undefined`**
- TypeScript also adds
  - **`enum`** enumerations like **`{One, Two, Three}`**
  - **`any`** use any type
  - **`void`** nothing



Primitive Types

```
let isComplete: boolean = false;
let width: number = 6;
let name: string = 'Doe';
let list: number[] = [1,2,3];
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
let unCertain: any = 4;
```



Functions

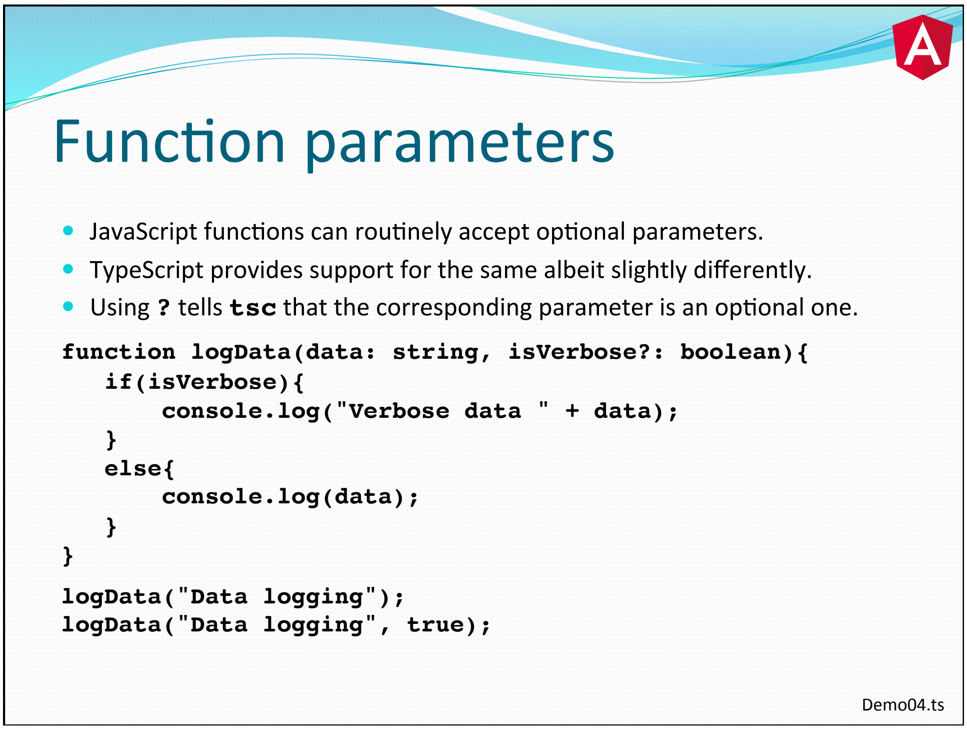
```
function getDayName(dayNumber: number):
                                  string {

   const daysArray: string[] = ['Sunday',
          'Monday', 'Tuesday', 'Wednesday',
          'Thursday', 'Friday', 'Saturday'];

   const dayName: string = daysArray[new
                    Date().getDay()];

   return dayName;
}
```

Demo03.ts



Function parameters

- JavaScript functions can routinely accept optional parameters.
- TypeScript provides support for the same albeit slightly differently.
- Using **?** tells **`tsc`** that the corresponding parameter is an optional one.

```
function logData(data: string, isVerbose?: boolean){
   if(isVerbose){
       console.log("Verbose data " + data);
   }
   else{
       console.log(data);
   }
}
logData("Data logging");
logData("Data logging", true);
```

Demo04.ts