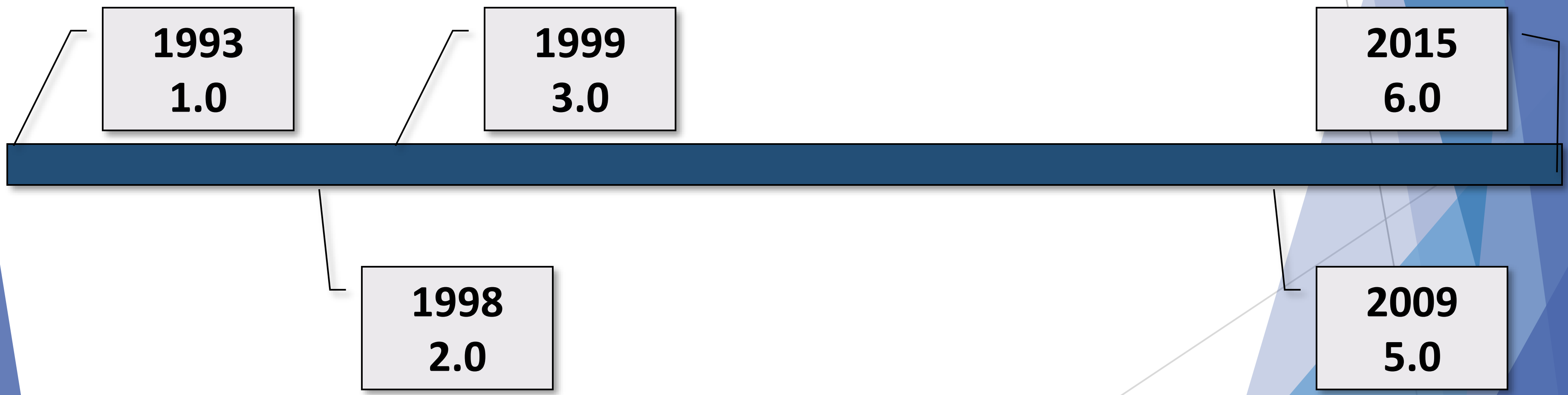
Quick Intro to Modern JavaScript and TypeScript

Who am I?

- Tiberiu ’Tibi’ Covaci
- Software engineer, 25 years experience
- CTO for DevMasters
- Microsoft Regional Director
- MCT since 2004, teaching web technologies
- ASP.NET Insider
- Aurelia core team member
- Father & Geek
- @tibor19 @DevEurope #DEVIntersectionEU

Why The Excitement?

- First substantial addition to JavaScript since inception



1993
1.0

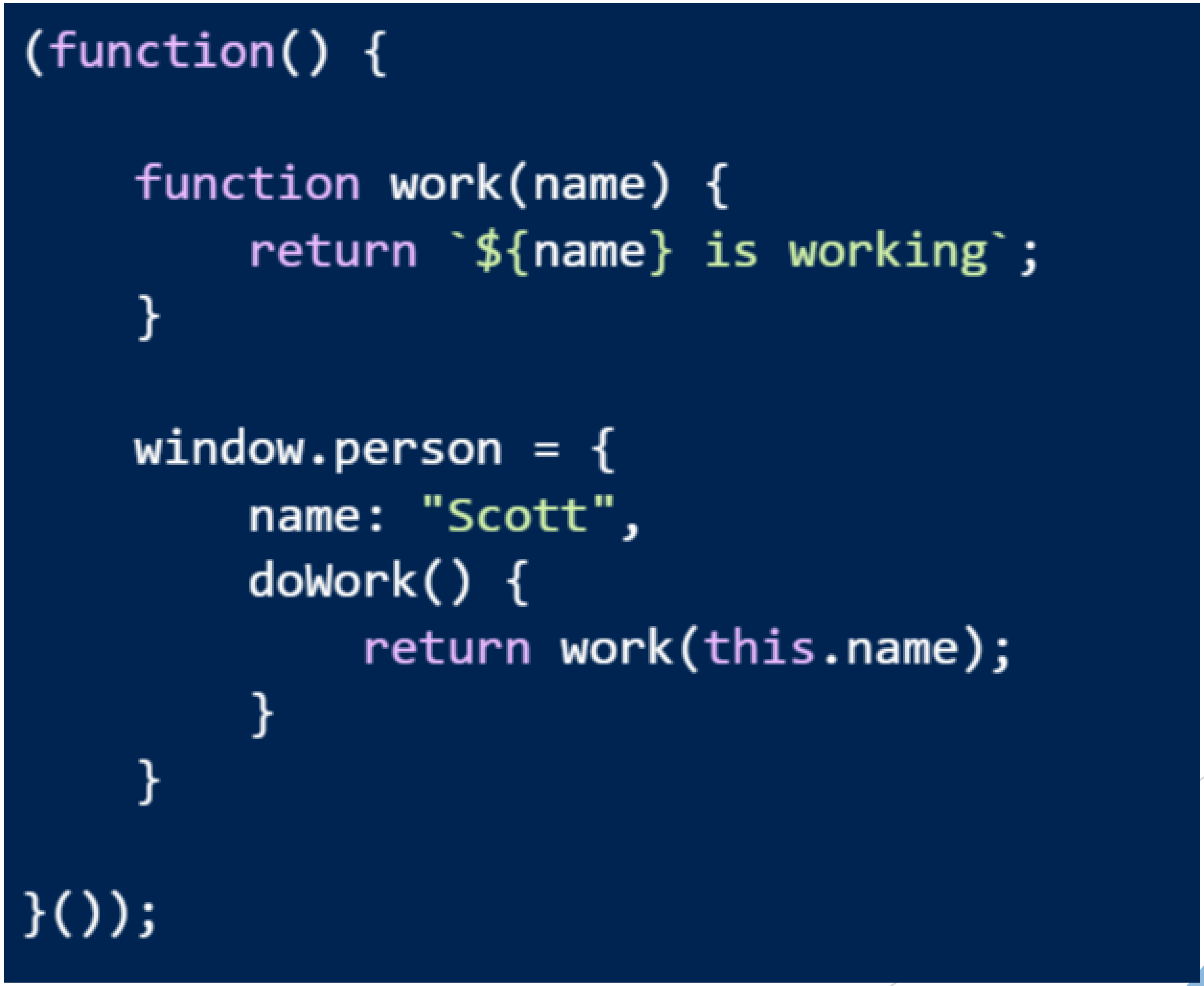
1998
2.0

1999
3.0

2009
5.0

2015
6.0

Problem: Modularity & Scope



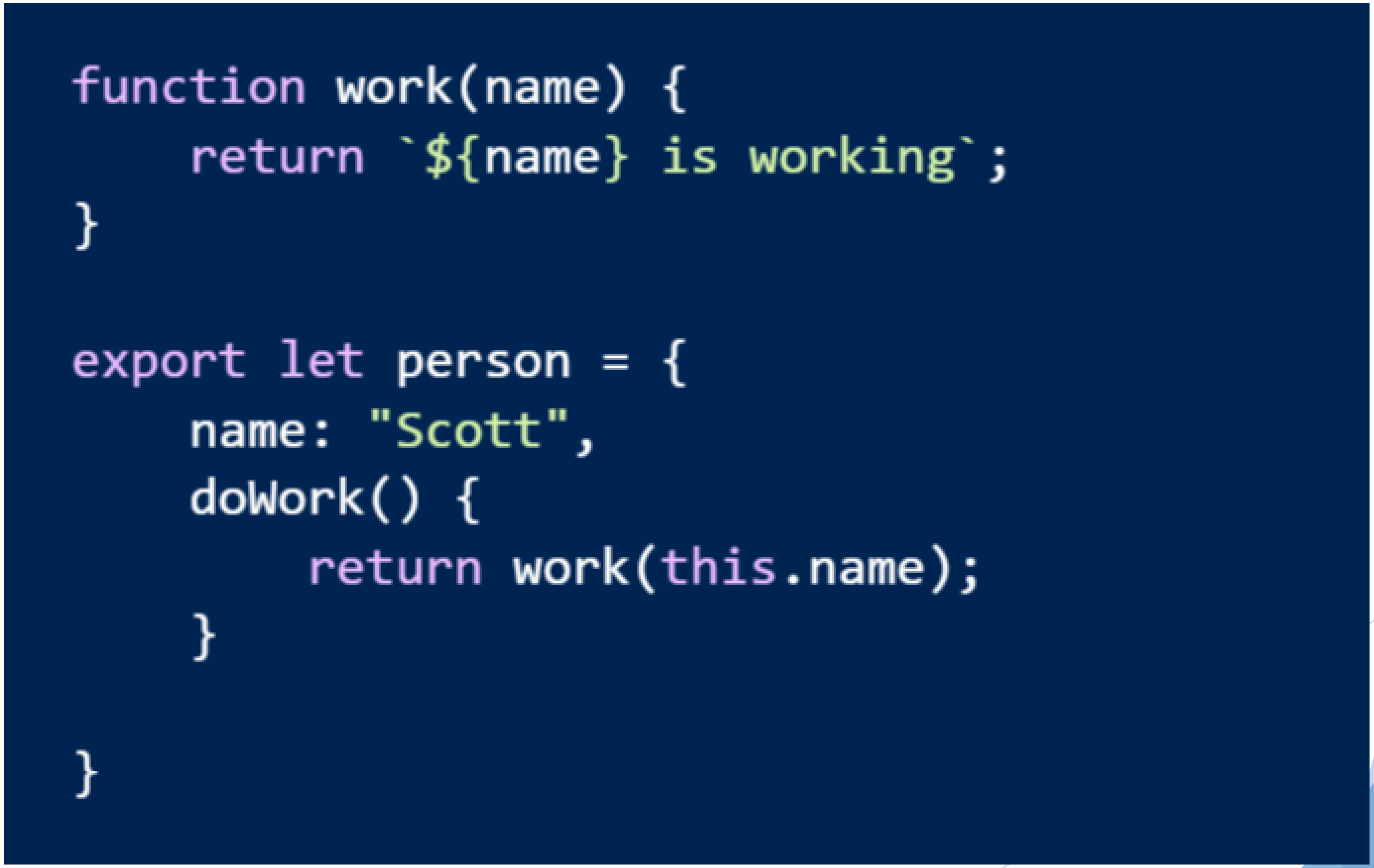
```
(function() {

    function work(name) {
        return `${name} is working`;
    }

    window.person = {
        name: "Scott",
        doWork() {
            return work(this.name);
        }
    }

}());
```

Problem: Modularity & Scope

- Possible solutions
  - Common JS
  - Asynchronous Module Definitions
  - IFFE and Globals
- Think about how current libraries from 2014 are designed
  - jQuery -> $
  - Angular -> angular
  - Lodash -> _

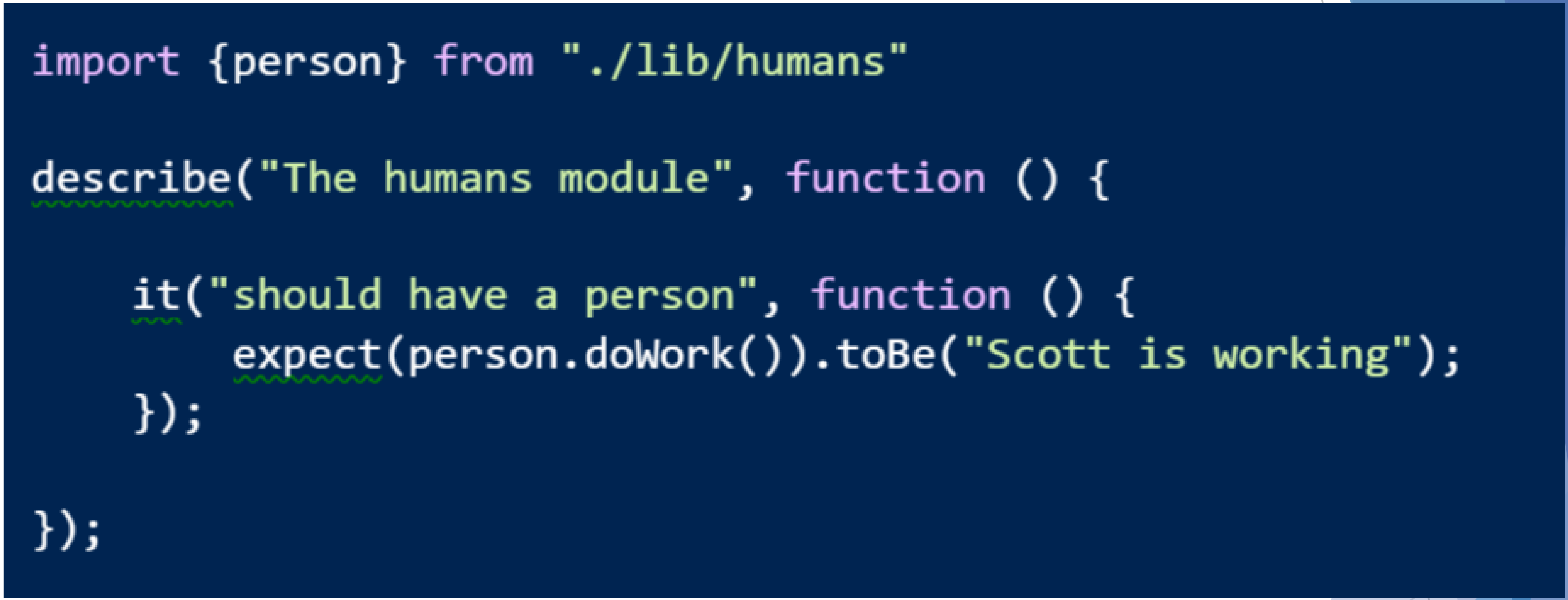
Solution: Real Modules!

- Think “module” not “file”



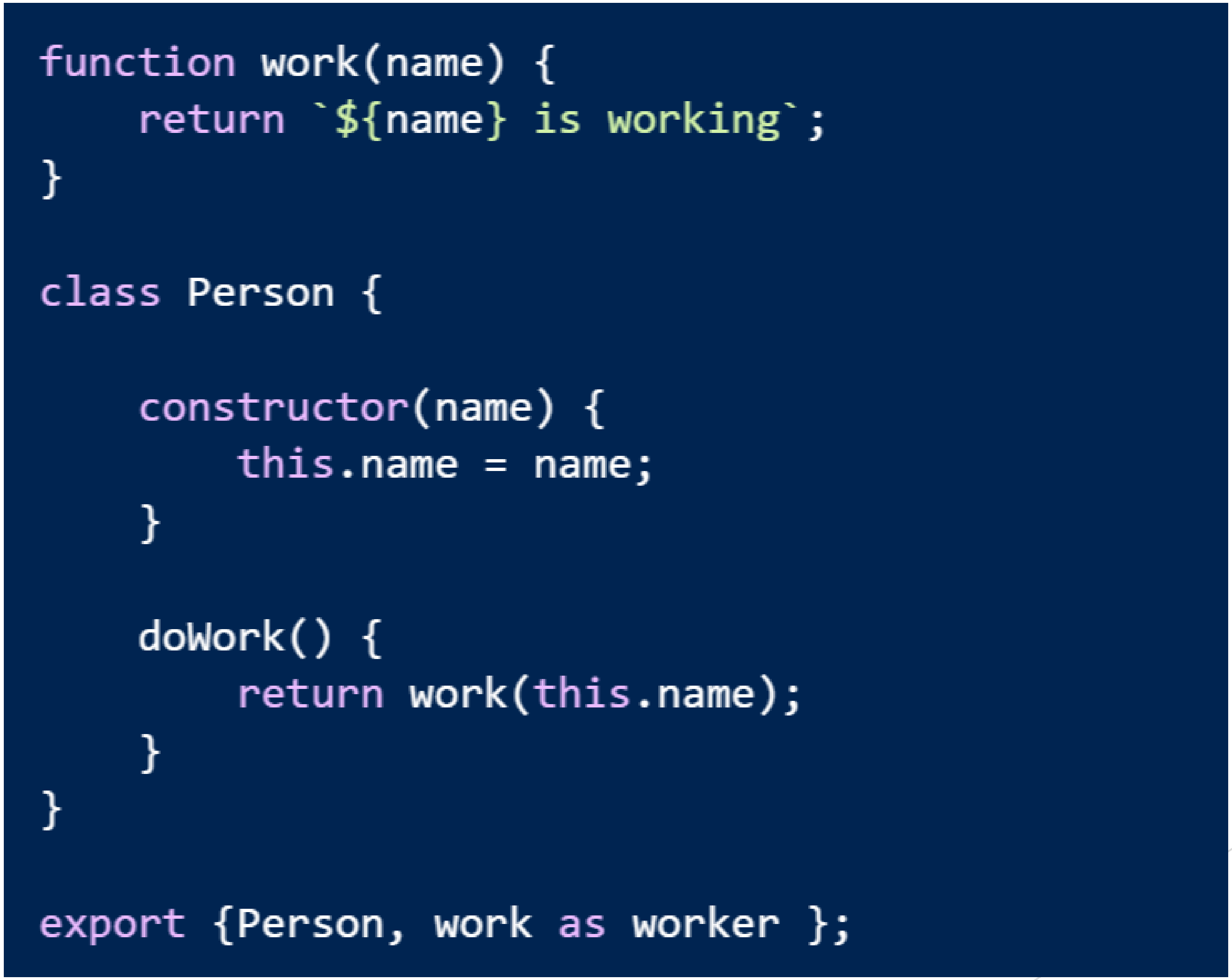
```
function work(name) {
    return `${name} is working`;
}

export let person = {
    name: "Scott",
    doWork() {
        return work(this.name);
    }

}
```

Imports



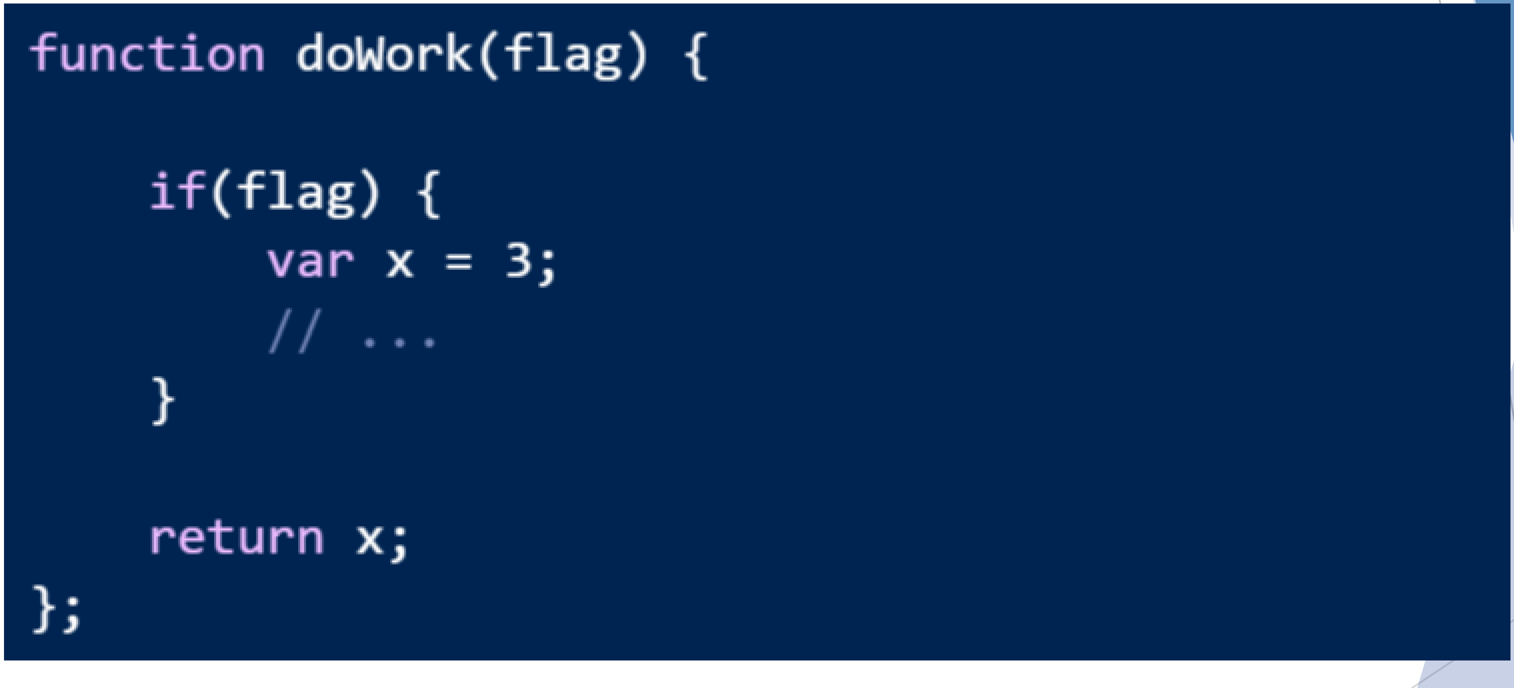
```
import {person} from "./lib/humans"

describe("The humans module", function () {

    it("should have a person", function () {
        expect(person.doWork()).toBe("Scott is working");
    });

});
```

Multiple Exports



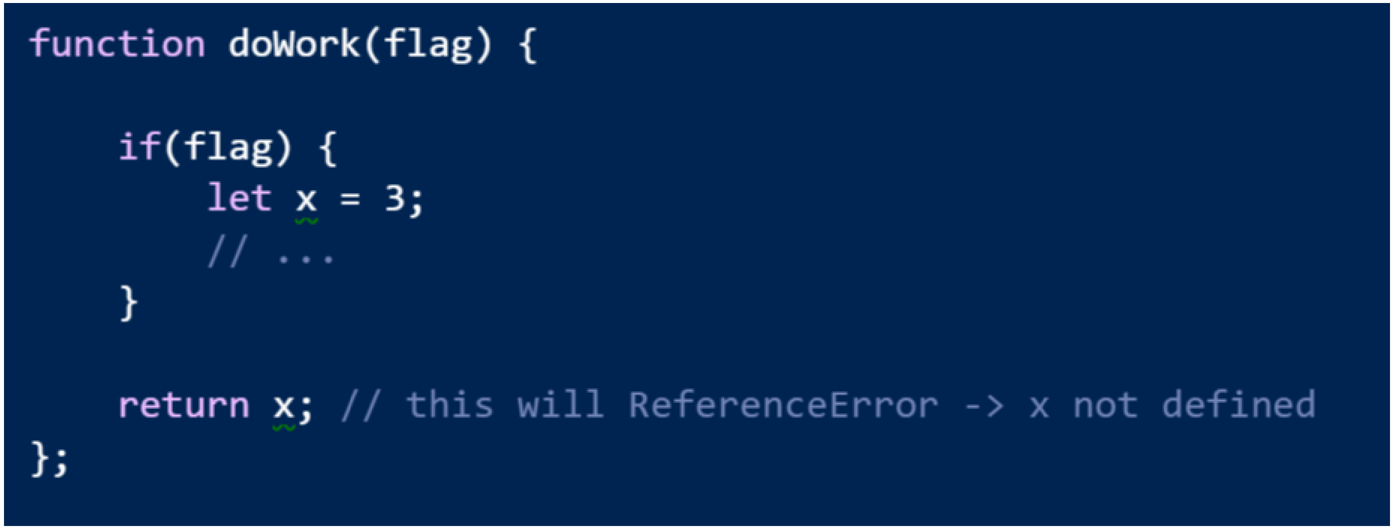
```
function work(name) {
    return `${name} is working`;
}

class Person {

    constructor(name) {
        this.name = name;
    }

    doWork() {
        return work(this.name);
    }
}

export {Person, work as worker };
```

Problem: JavaScript has no block scope

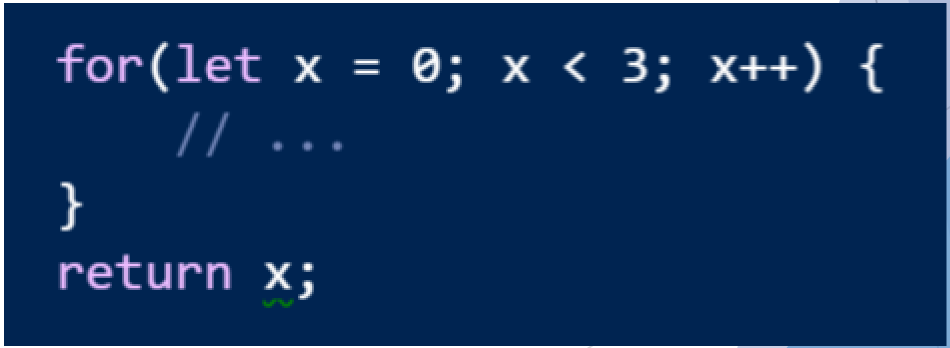


```
function doWork(flag) {

    if(flag) {
        var x = 3;
        // ...
    }

    return x;
};
```

Solution: let



```
function doWork(flag) {

    if(flag) {
        let x = 3;
        // ...
    }

    return x; // this will ReferenceError -> x not defined
};
```

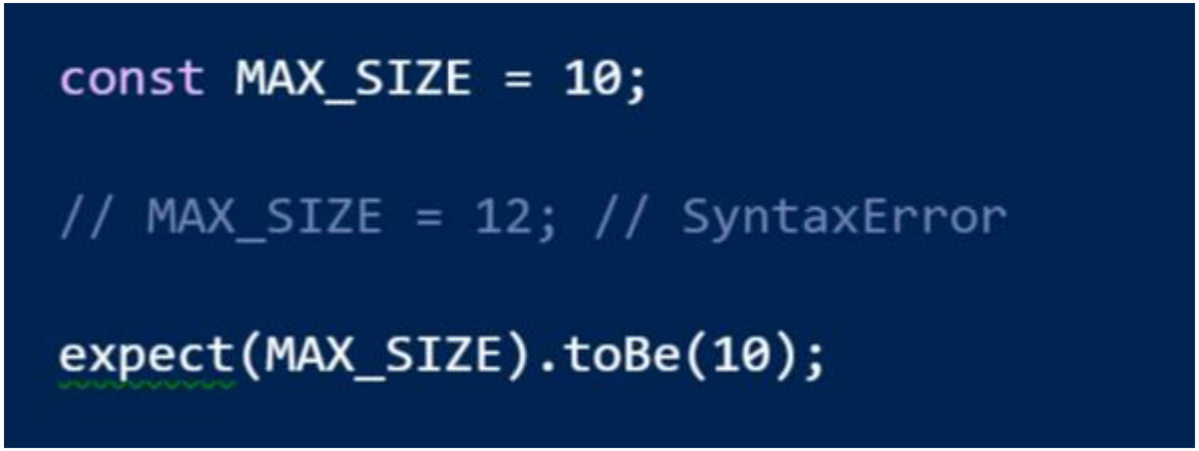


```
for(let x = 0; x < 3; x++) {
    // ...
}
return x;
```

Problem: All variables are mutable

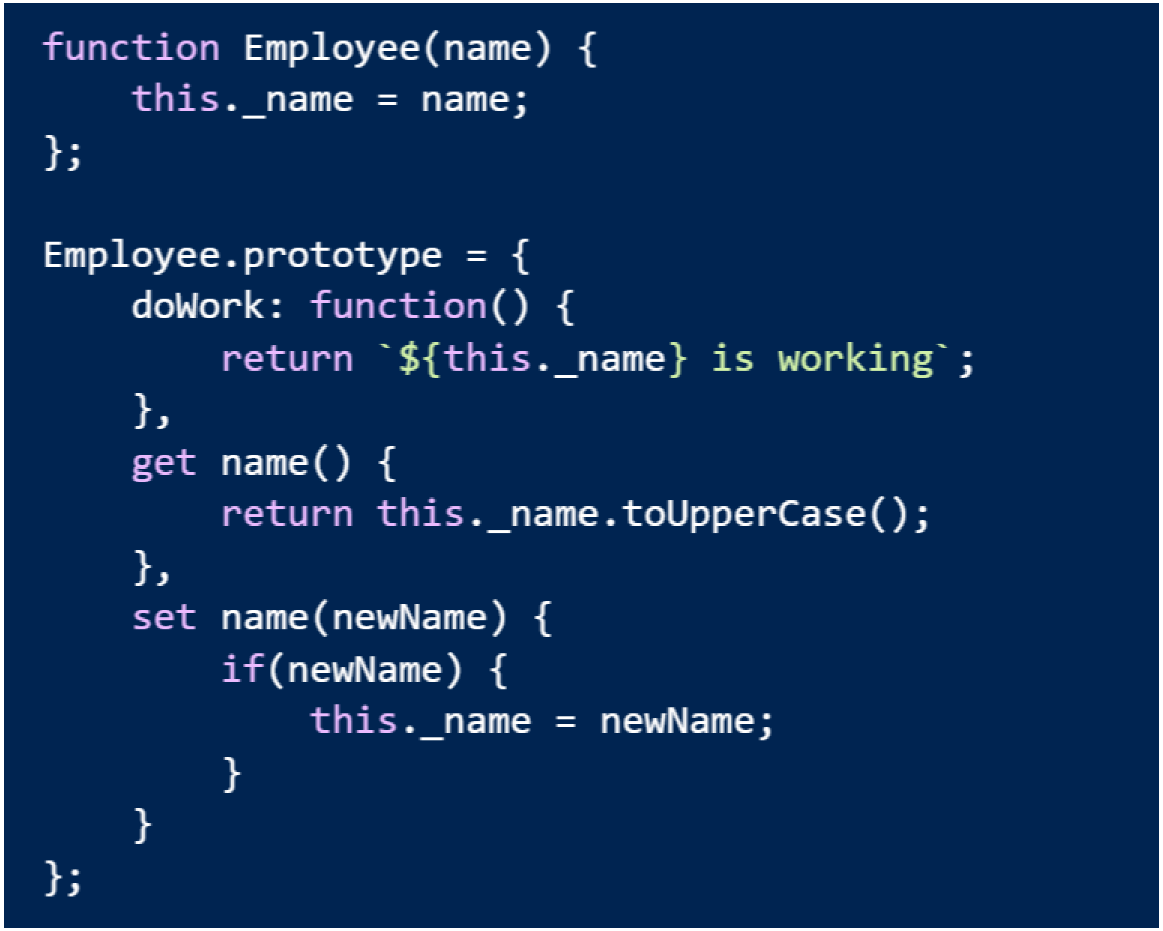
$$\pi=4$$

Solution: const



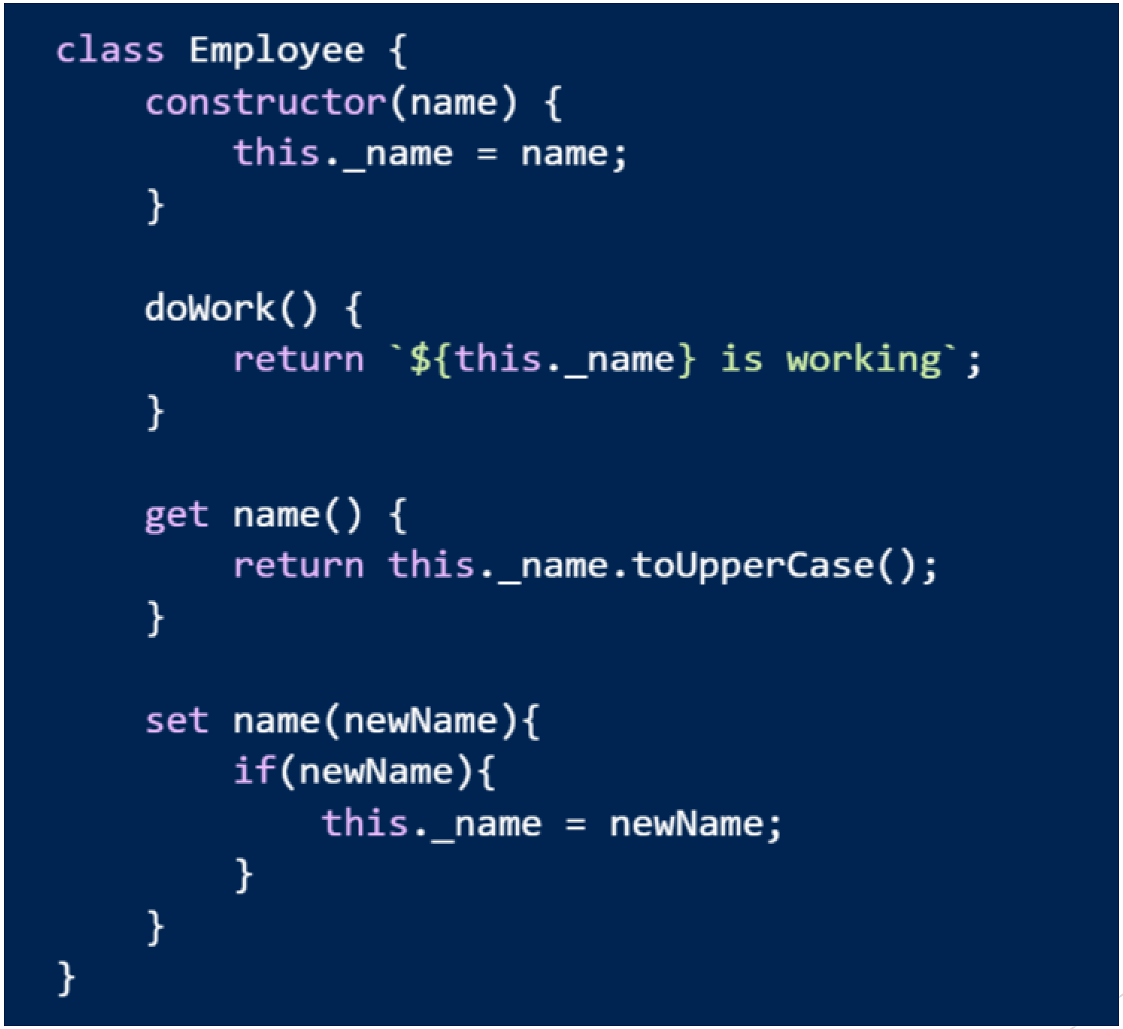
```
const MAX_SIZE = 10;

// MAX_SIZE = 12; // SyntaxError

expect(MAX_SIZE).toBe(10);
```

Problem: Simulating OOP



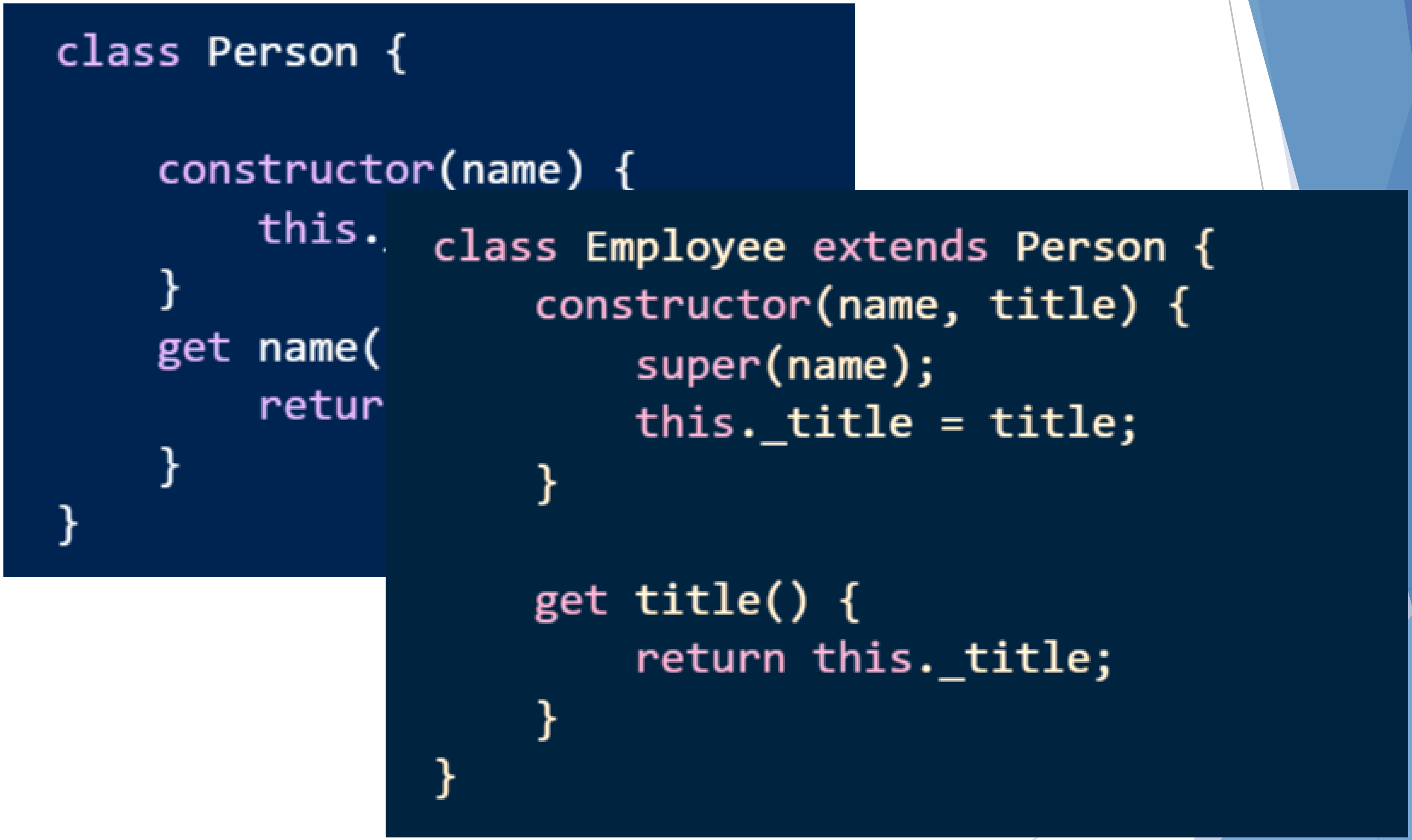
```
function Employee(name) {
    this._name = name;
};

Employee.prototype = {
    doWork: function() {
        return `${this._name} is working`;
    },
    get name() {
        return this._name.toUpperCase();
    },
    set name(newName) {
        if(newName) {
            this._name = newName;
        }
    }
};
```

Solution: class Keyword



```
class Employee {
    constructor(name) {
        this._name = name;
    }

    doWork() {
        return `${this._name} is working`;
    }

    get name() {
        return this._name.toUpperCase();
    }

    set name(newName){
        if(newName){
            this._name = newName;
        }
    }
}
```

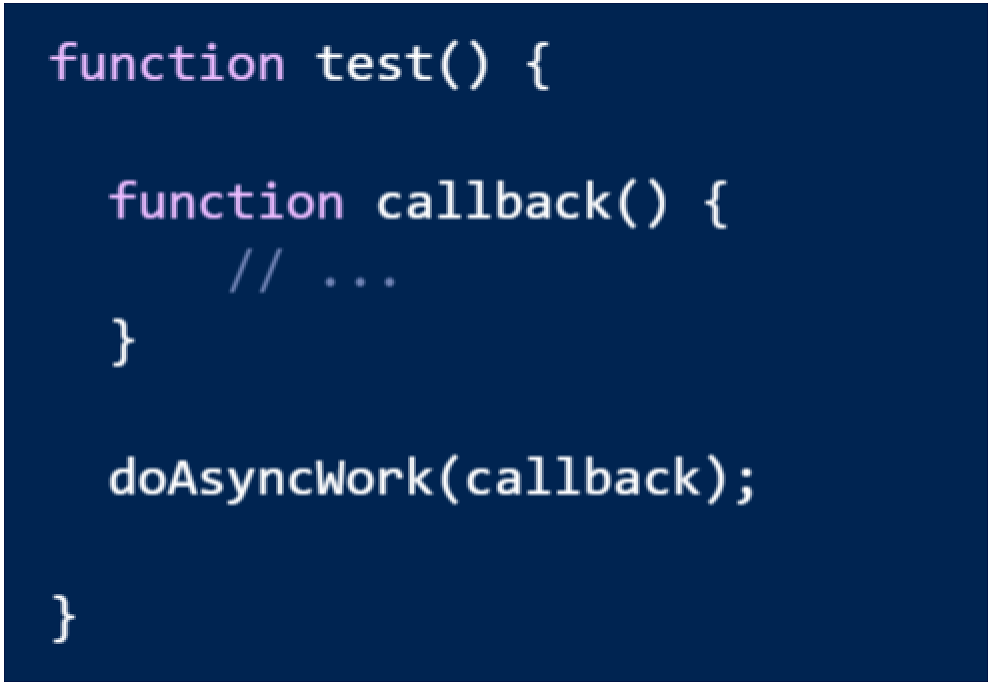
Inheritance



```
class Person {

    constructor(name) {
        this.
    }
    get name(
        retur
    }
}
```

```
class Employee extends Person {
    constructor(name, title) {
        super(name);
        this._title = title;
    }

    get title() {
        return this._title;
    }
}
```

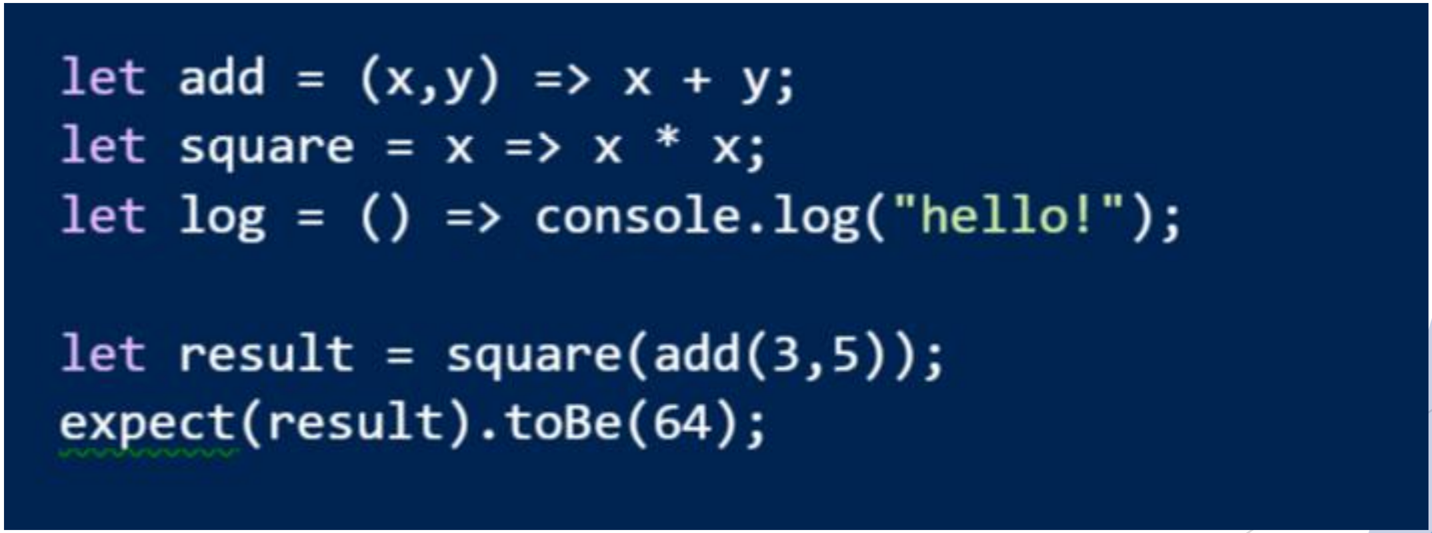
Problem: function is an 8 character word



```
function test() {

  function callback() {
      // ...
  }

  doAsyncWork(callback);

}
```

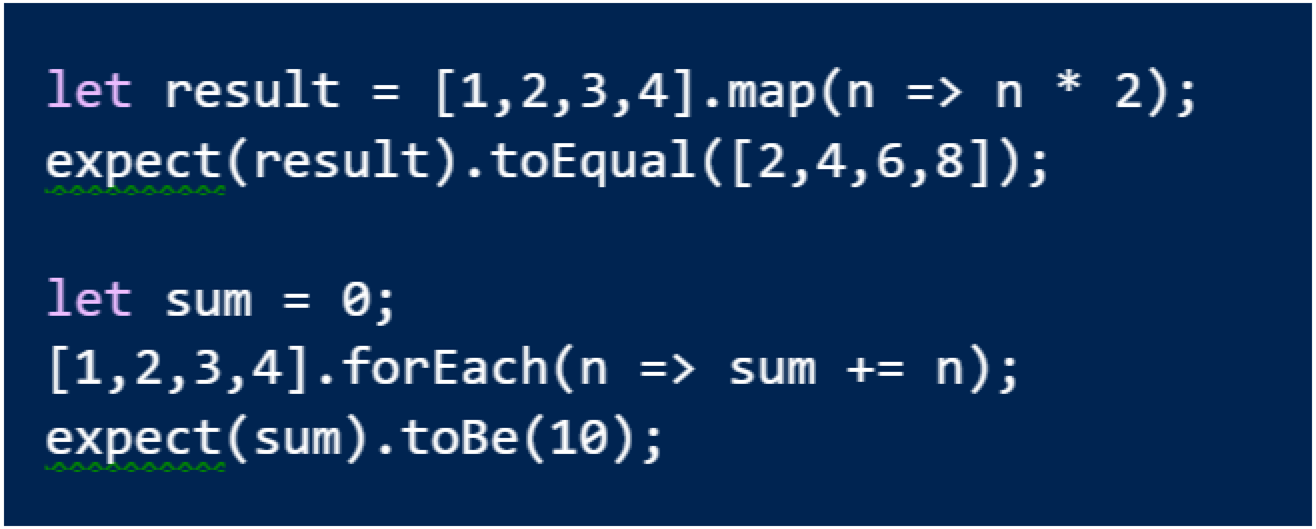
Arrow Functions

- Expressive syntax
- Familiar to C# developers



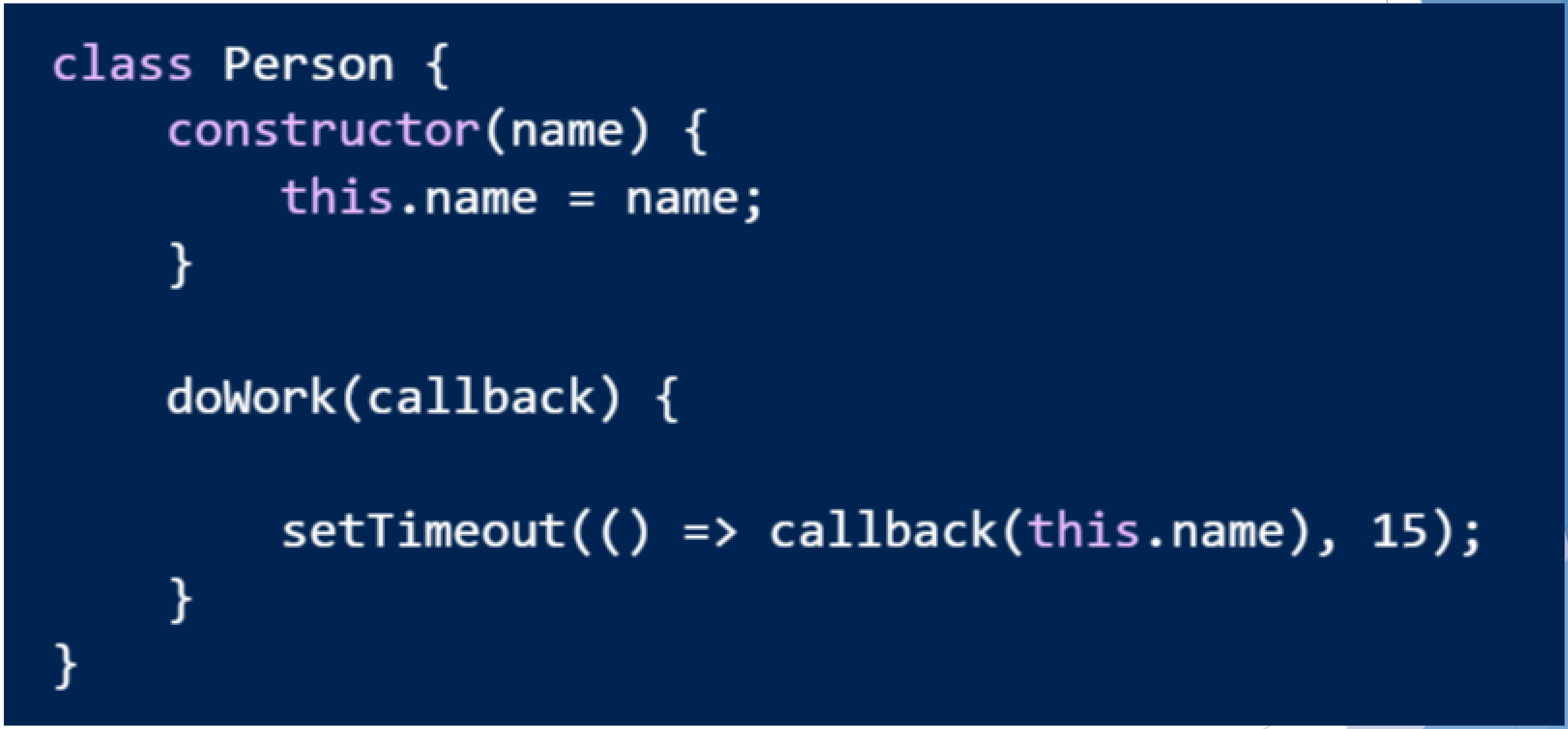
```
let add = (x,y) => x + y;
let square = x => x * x;
let log = () => console.log("hello!");

let result = square(add(3,5));
expect(result).toBe(64);
```

LINQish Arrows



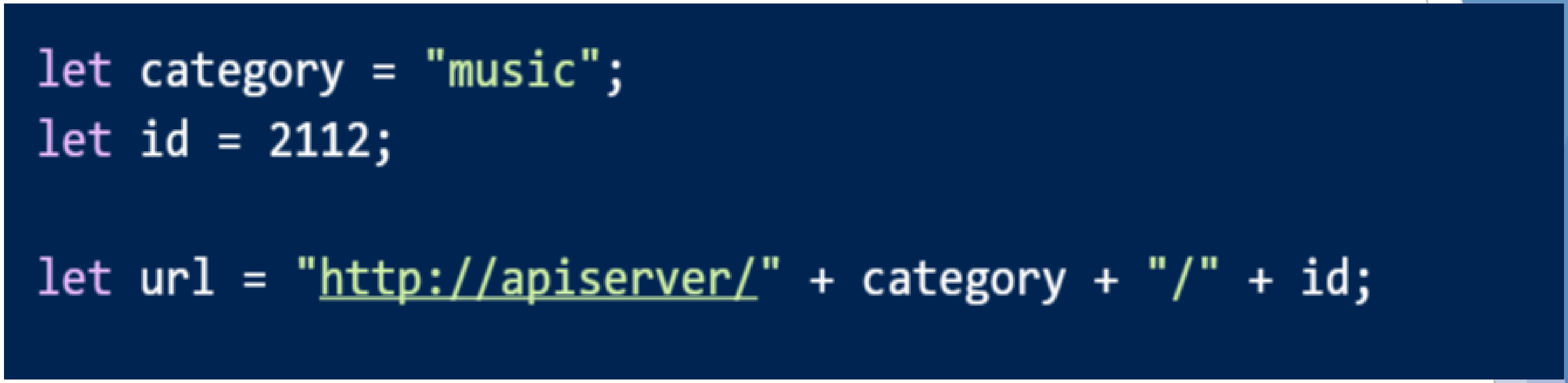
```
let result = [1,2,3,4].map(n => n * 2);
expect(result).toEqual([2,4,6,8]);

let sum = 0;
[1,2,3,4].forEach(n => sum += n);
expect(sum).toBe(10);
```

Arrow Functions Lexically Bind this



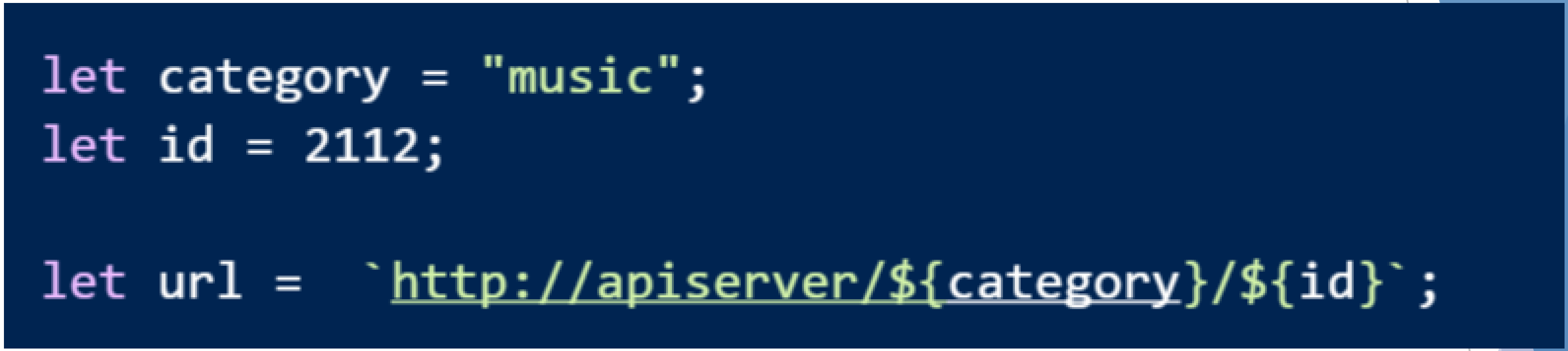
```
class Person {
    constructor(name) {
        this.name = name;
    }

    doWork(callback) {

        setTimeout(() => callback(this.name), 15);
    }
}
```

Problem: String Concatenation Is Unpleasant



```
let category = "music";
let id = 2112;

let url = "http://apiserver/" + category + "/" + id;
```

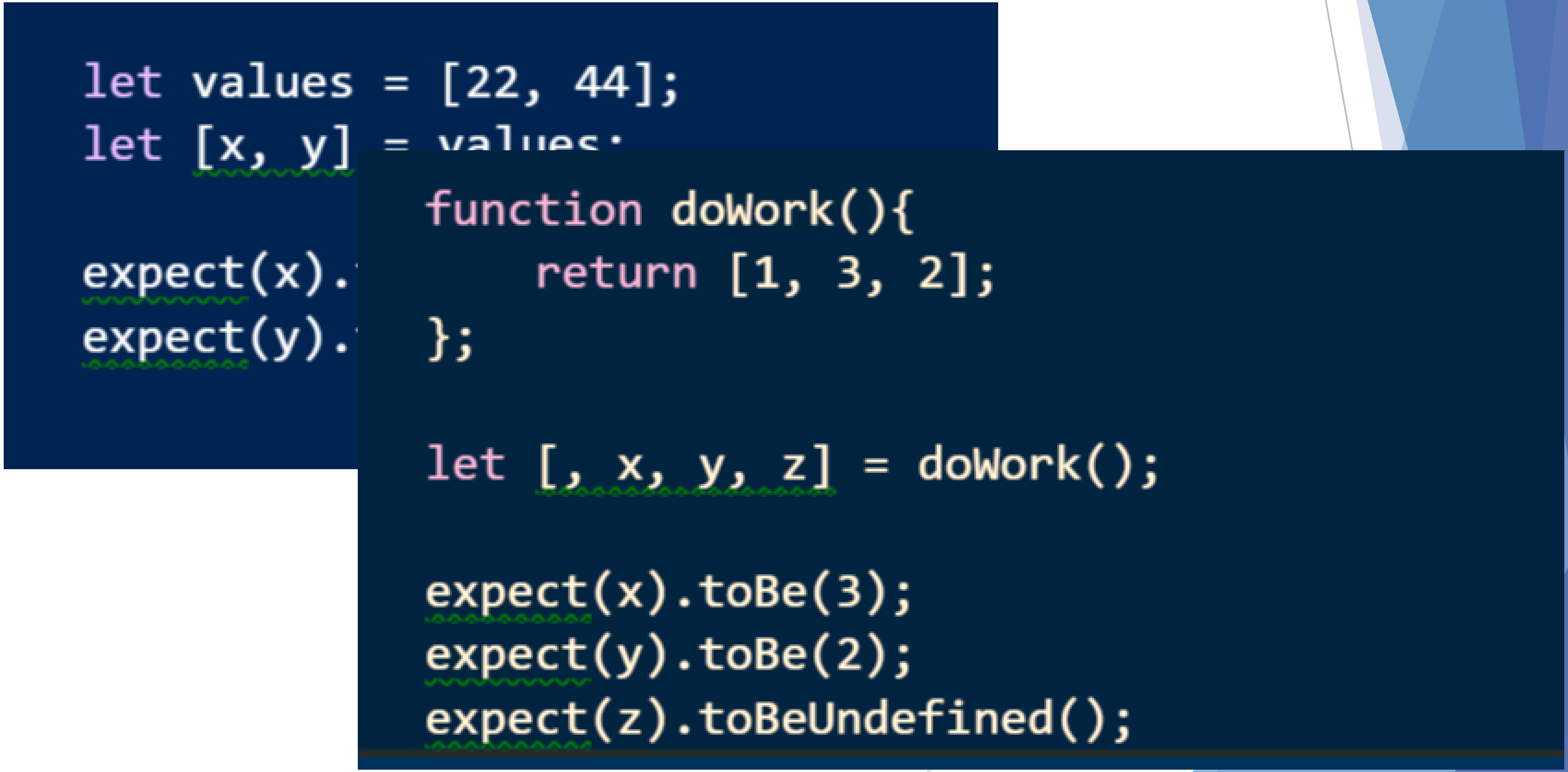
Solution: String Templates



```
let category = "music";
let id = 2112;

let url =  `http://apiserver/${category}/${id}`;
```

Destructuring

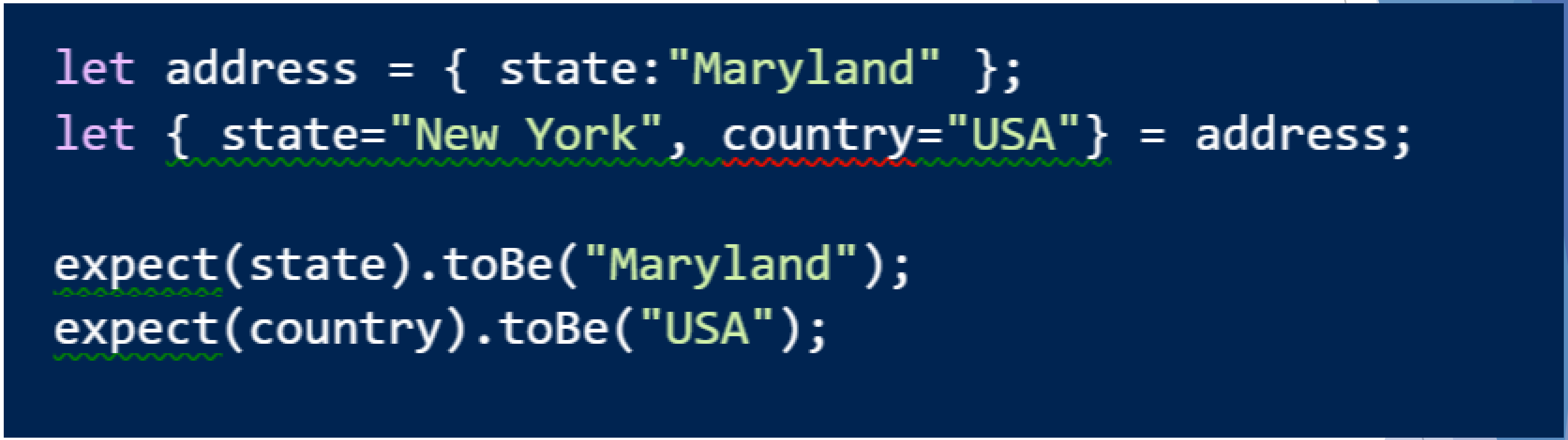
- The opposite of constructing is destructing



```
let values = [22, 44];
let [x, y] = values;

expect(x).
expect(y).
```

```
function doWork(){
    return [1, 3, 2];
};

let [, x, y, z] = doWork();

expect(x).toBe(3);
expect(y).toBe(2);
expect(z).toBeUndefined();
```

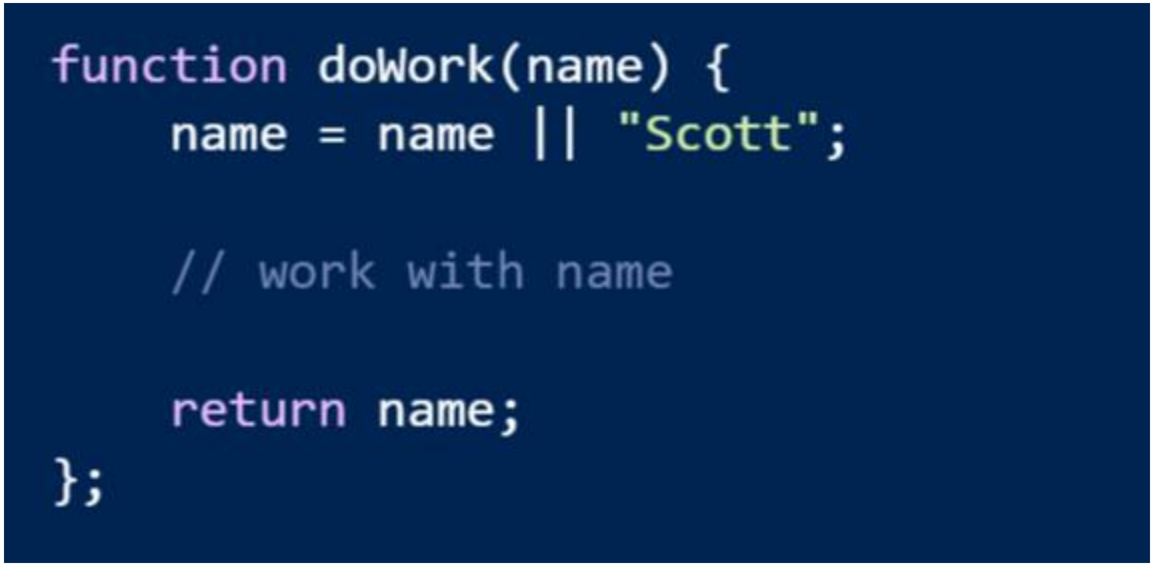
Destructuring Objects



```
let address = { state:"Maryland" };
let { state="New York", country="USA"} = address;

expect(state).toBe("Maryland");
expect(country).toBe("USA");
```

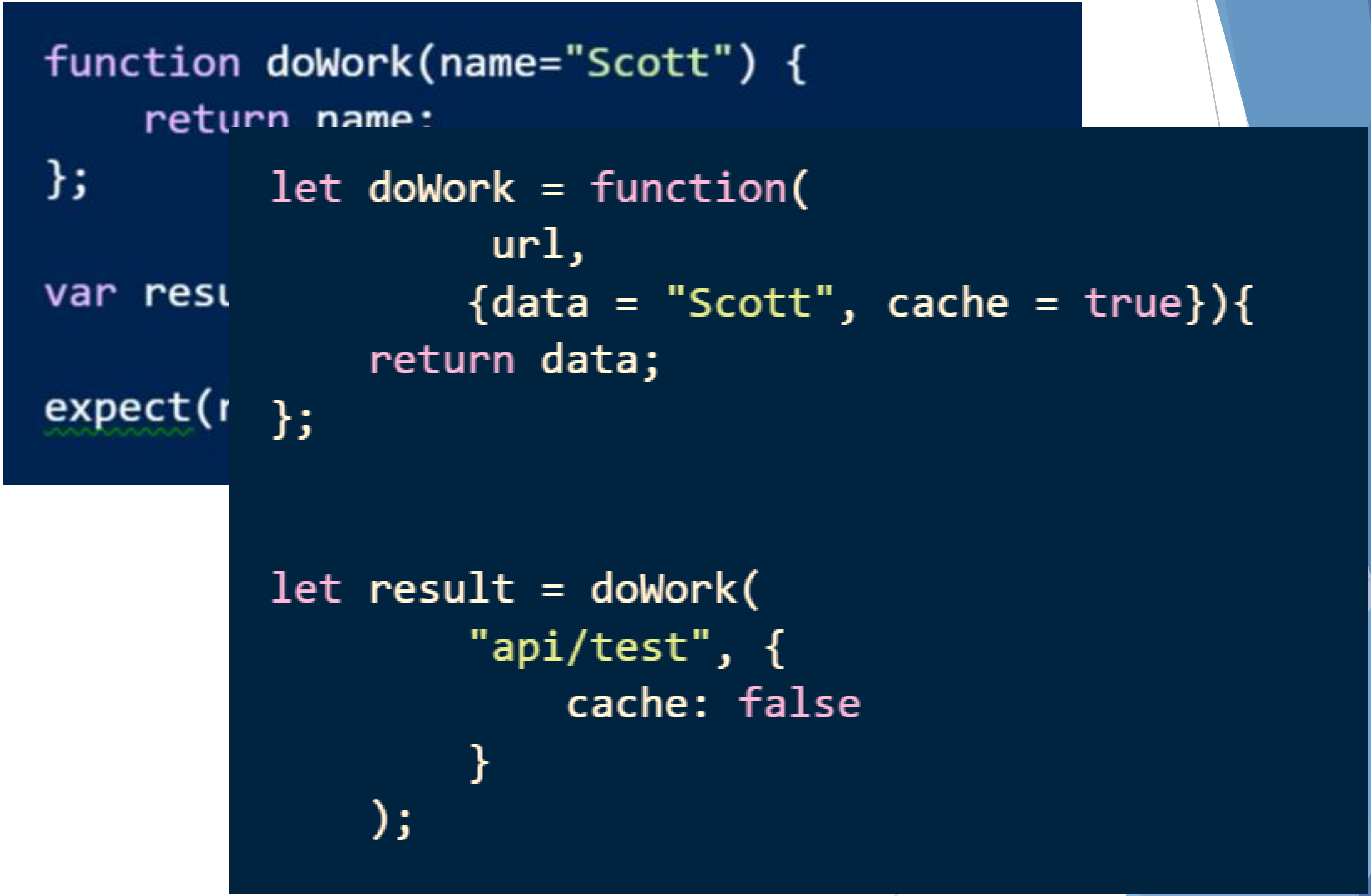
Problem: Default Values

- Default are buried in the function code



```
function doWork(name) {
    name = name || "Scott";

    // work with name

    return name;
};
```

Solution: Default Parameter Values

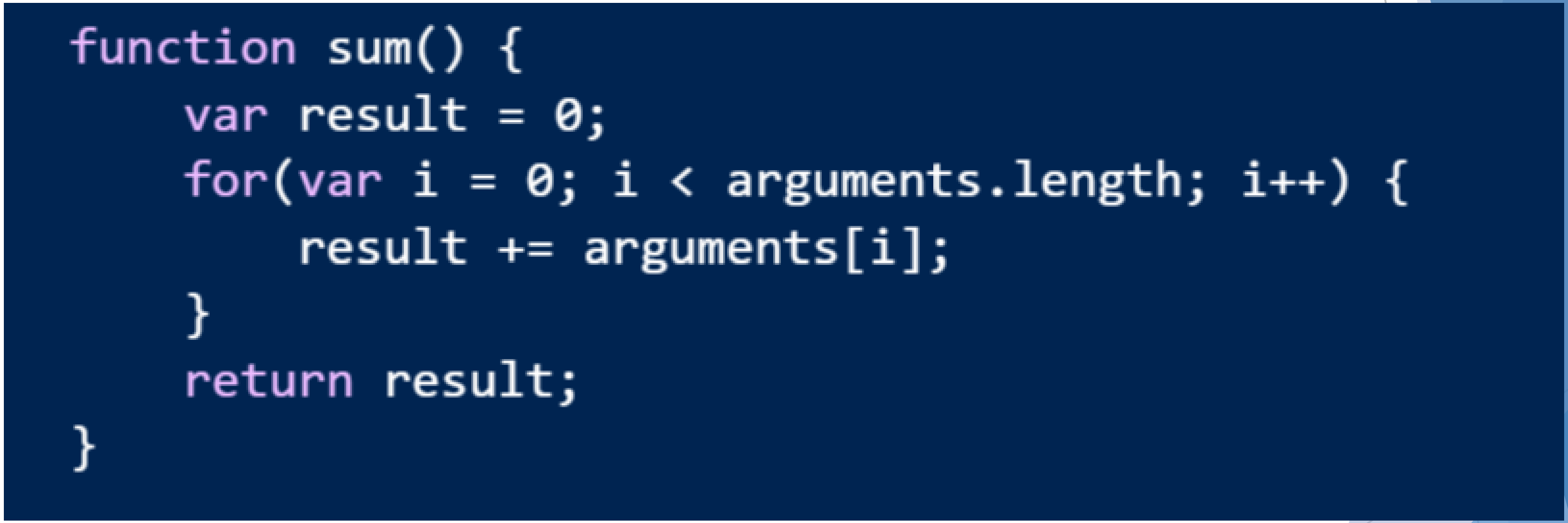


```
function doWork(name="Scott") {
    return name;
};

var resu

expect(r
```

```
let doWork = function(
         url,
        {data = "Scott", cache = true}){
    return data;
};


let result = doWork(
        "api/test", {
            cache: false
        }
    );
```

Problem: Variable Number of Arguments

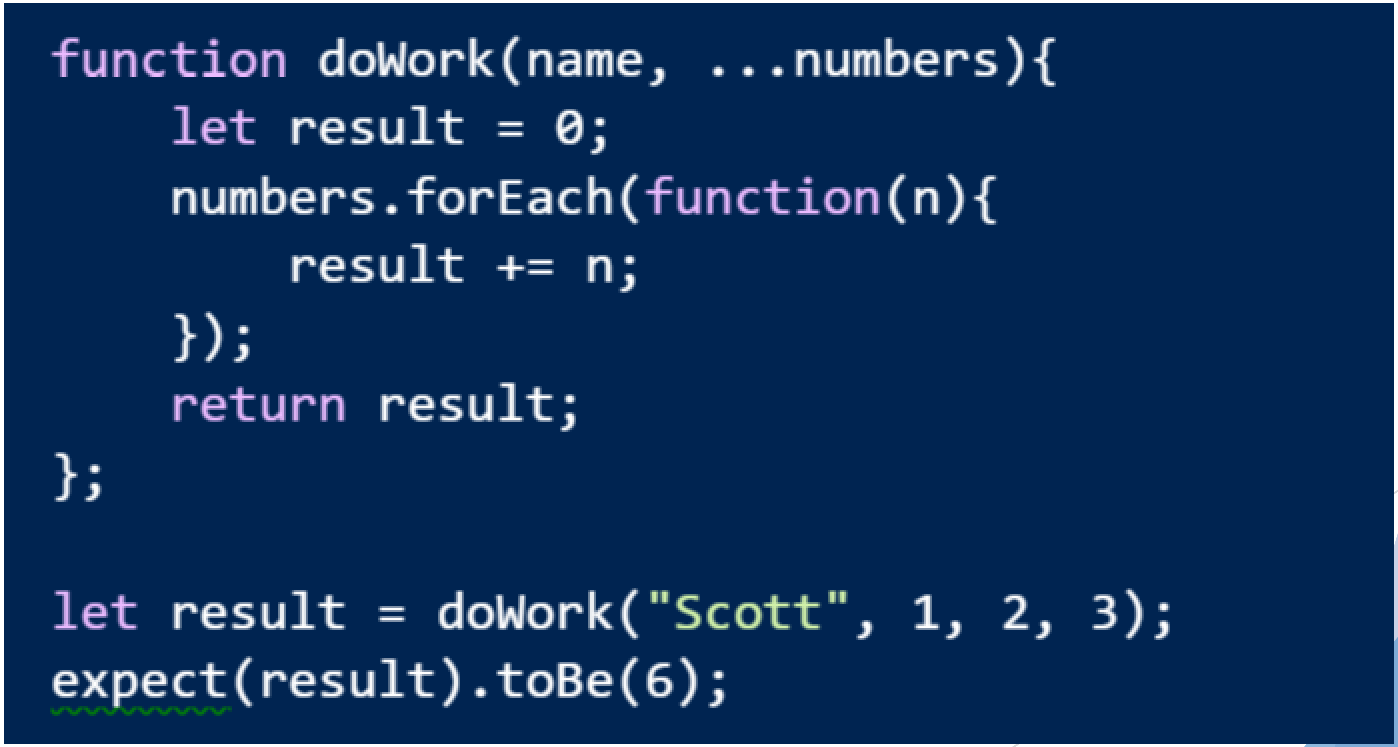
- Syntax is not obvious to the consumer



```
function sum() {
    var result = 0;
    for(var i = 0; i < arguments.length; i++) {
        result += arguments[i];
    }
    return result;
}
```

Solution: Rest Parameters

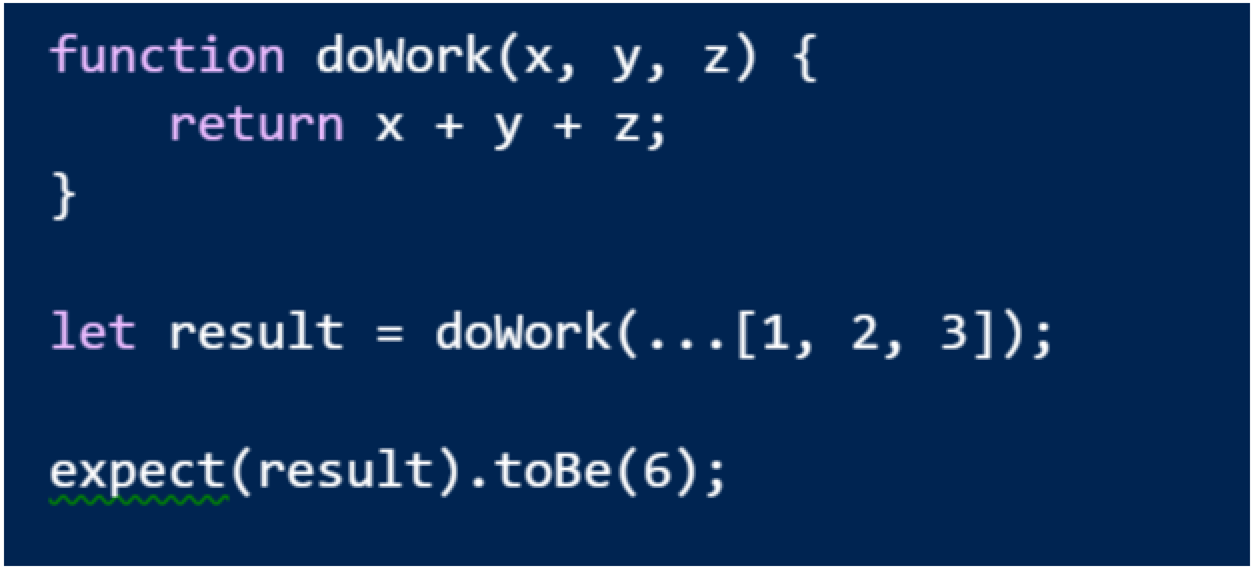
- Let the last parameter take the rest of the arguments



```
function doWork(name, ...numbers){
    let result = 0;
    numbers.forEach(function(n){
        result += n;
    });
    return result;
};

let result = doWork("Scott", 1, 2, 3);
expect(result).toBe(6);
```

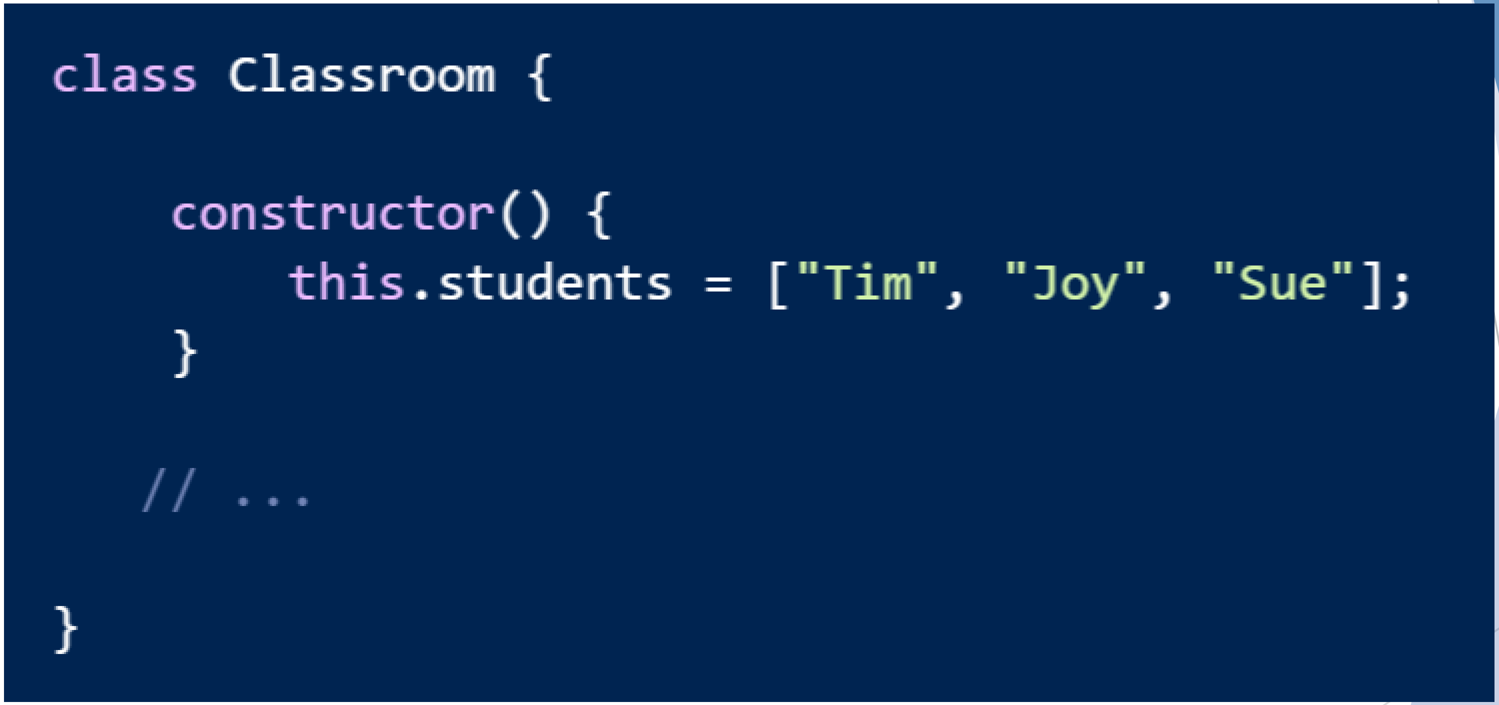
Spread Operator

- Spread an array across the parameters



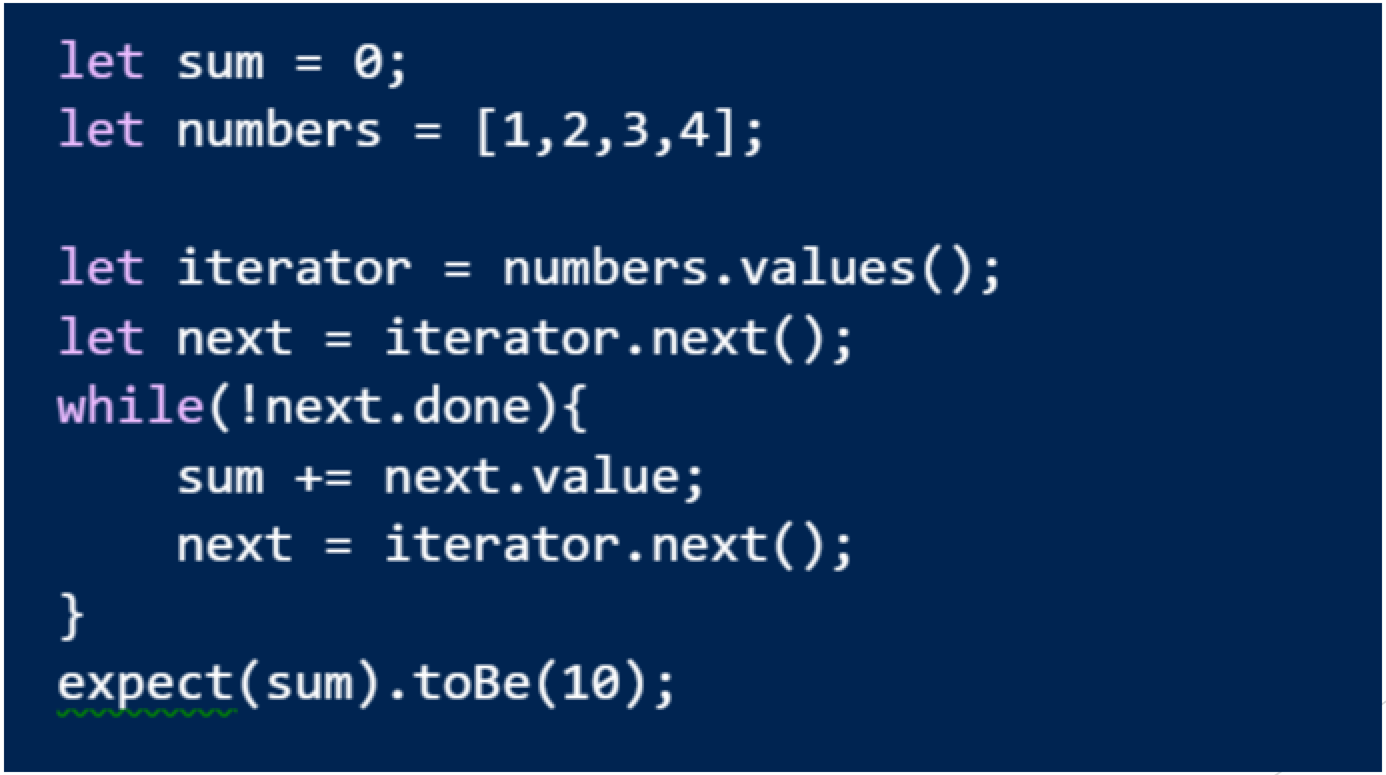
```
function doWork(x, y, z) {
    return x + y + z;
}

let result = doWork(...[1, 2, 3]);

expect(result).toBe(6);
```

Problem: Encapsulating Collections



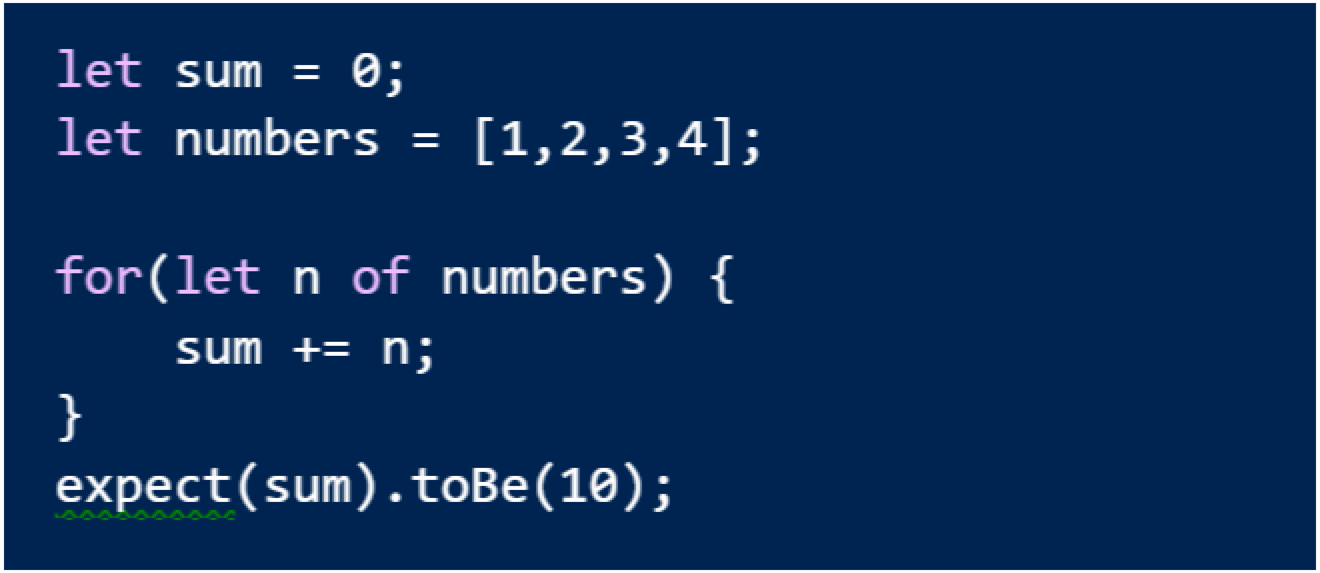
```
class Classroom {

    constructor() {
        this.students = ["Tim", "Joy", "Sue"];
    }

   // ...

}
```

Solution: Iterators and Iterables



```
let sum = 0;
let numbers = [1,2,3,4];

let iterator = numbers.values();
let next = iterator.next();
while(!next.done){
    sum += next.value;
    next = iterator.next();
}
expect(sum).toBe(10);
```

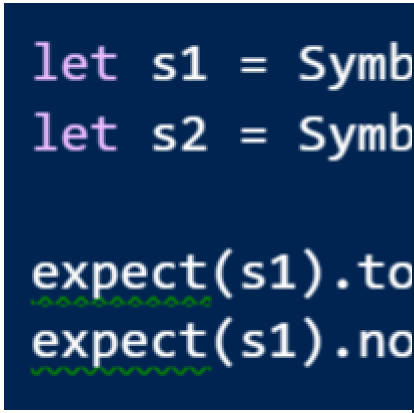
for of



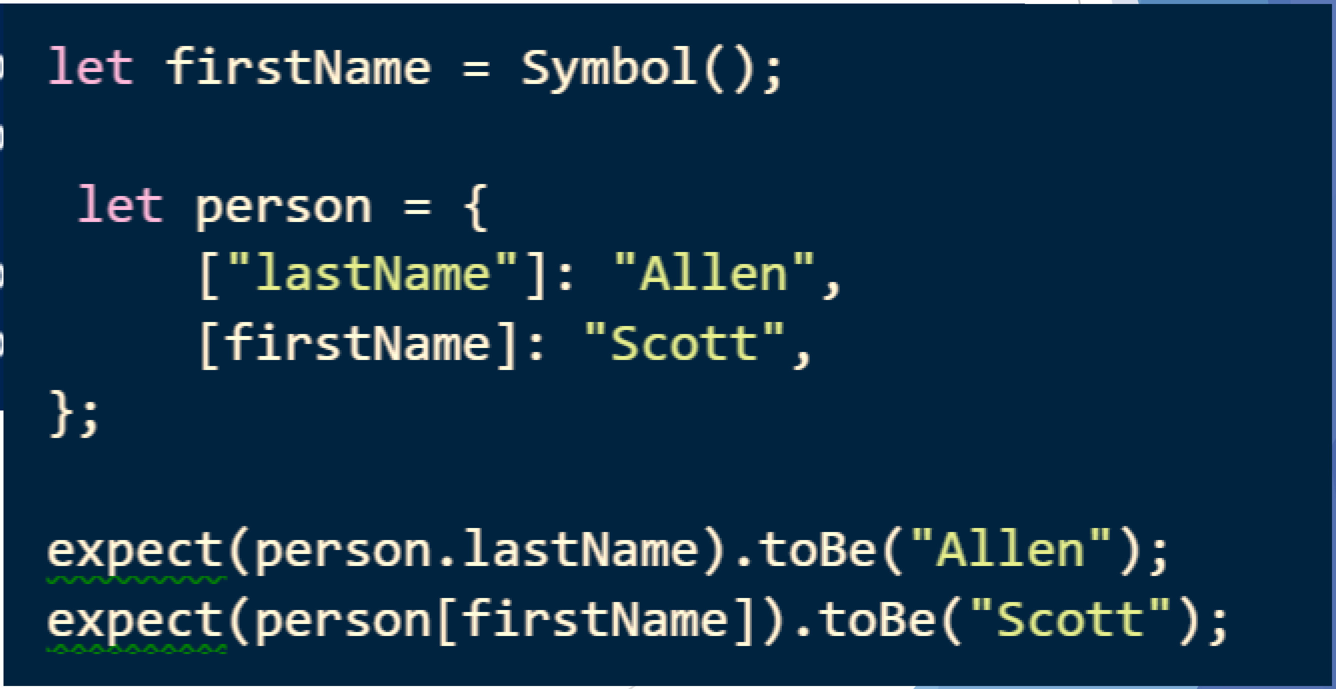
```
let sum = 0;
let numbers = [1,2,3,4];

for(let n of numbers) {
    sum += n;
}
expect(sum).toBe(10);
```

Symbol

- A new type where every value is unique and immutable
- Can use a symbol as a key into an object



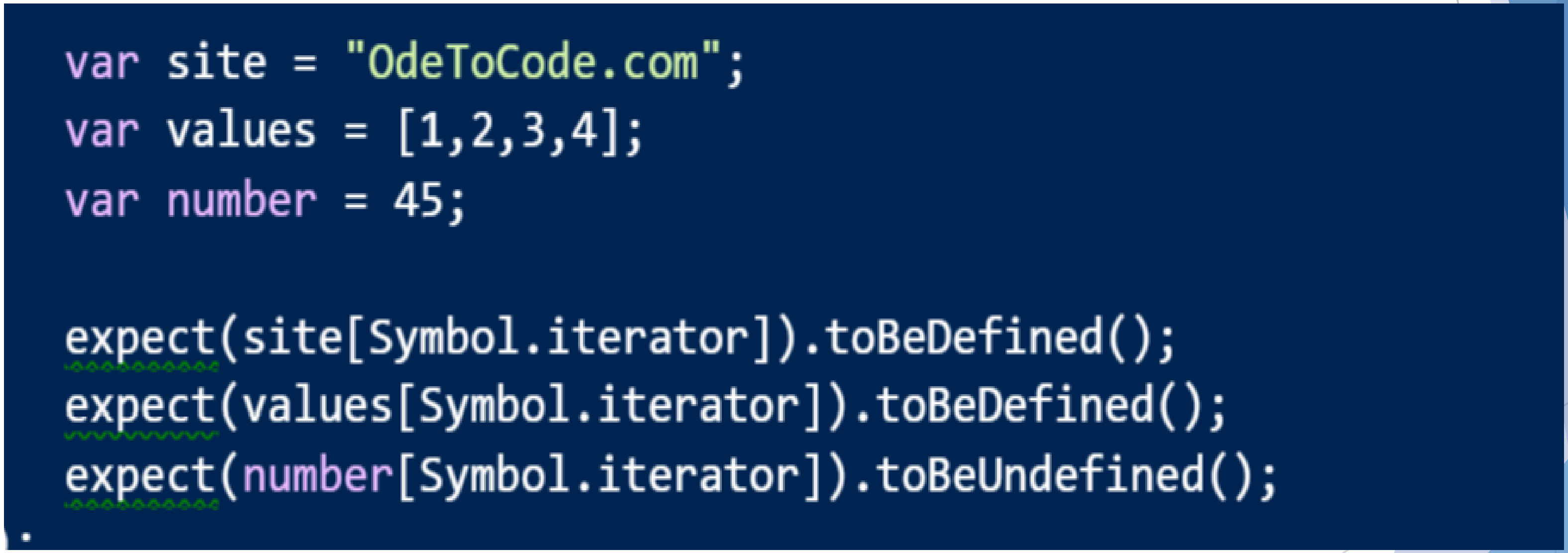
```
let s1 = Symb
let s2 = Symb

expect(s1).to
expect(s1).no
```



```
let firstName = Symbol();

 let person = {
     ["lastName"]: "Allen",
     [firstName]: "Scott",
};

expect(person.lastName).toBe("Allen");
expect(person[firstName]).toBe("Scott");
```

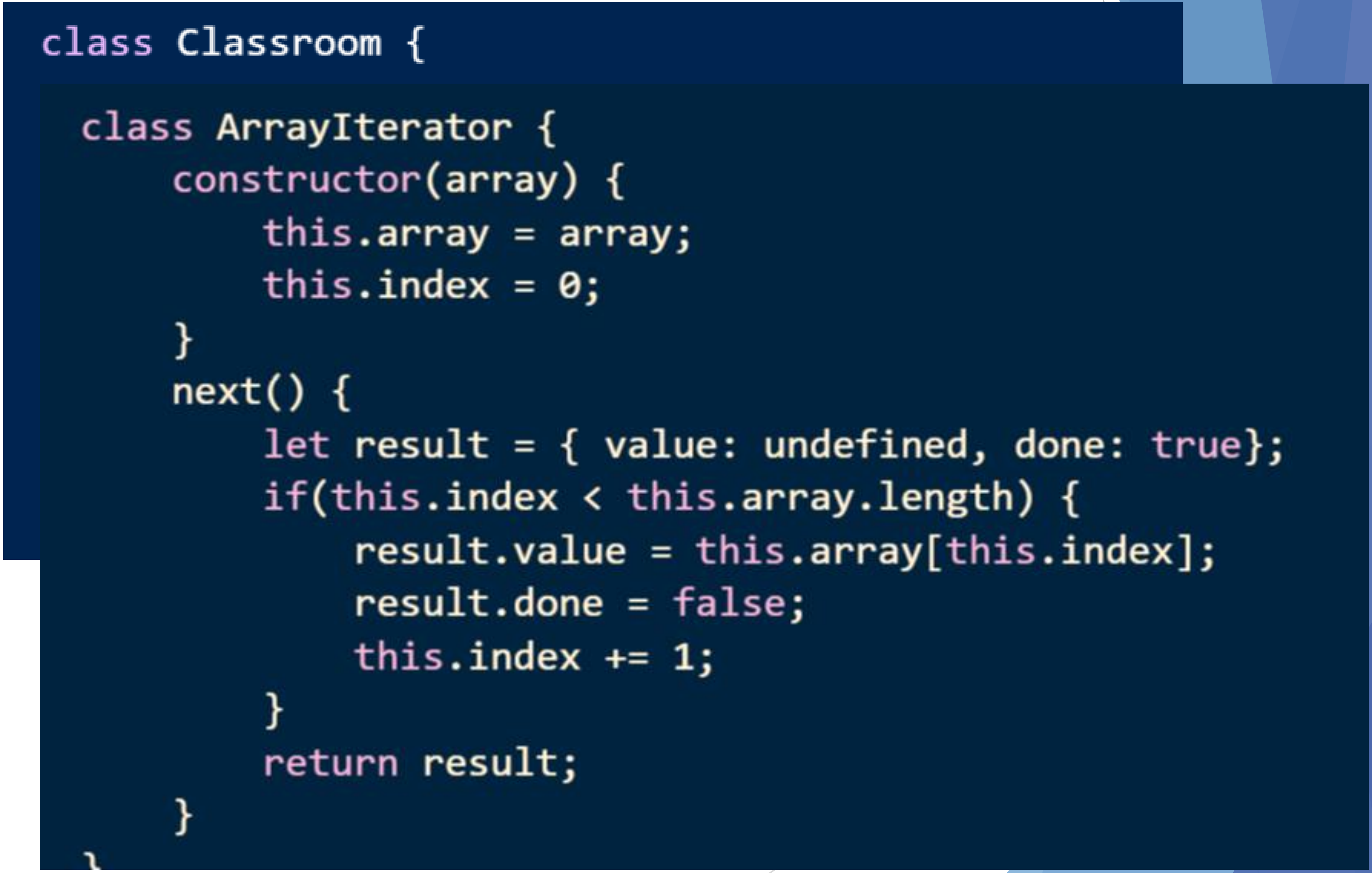
Symbol.iterator

- A magic method that makes an object iterable



```
var site = "OdeToCode.com";
var values = [1,2,3,4];
var number = 45;

expect(site[Symbol.iterator]).toBeDefined();
expect(values[Symbol.iterator]).toBeDefined();
expect(number[Symbol.iterator]).toBeUndefined();
```

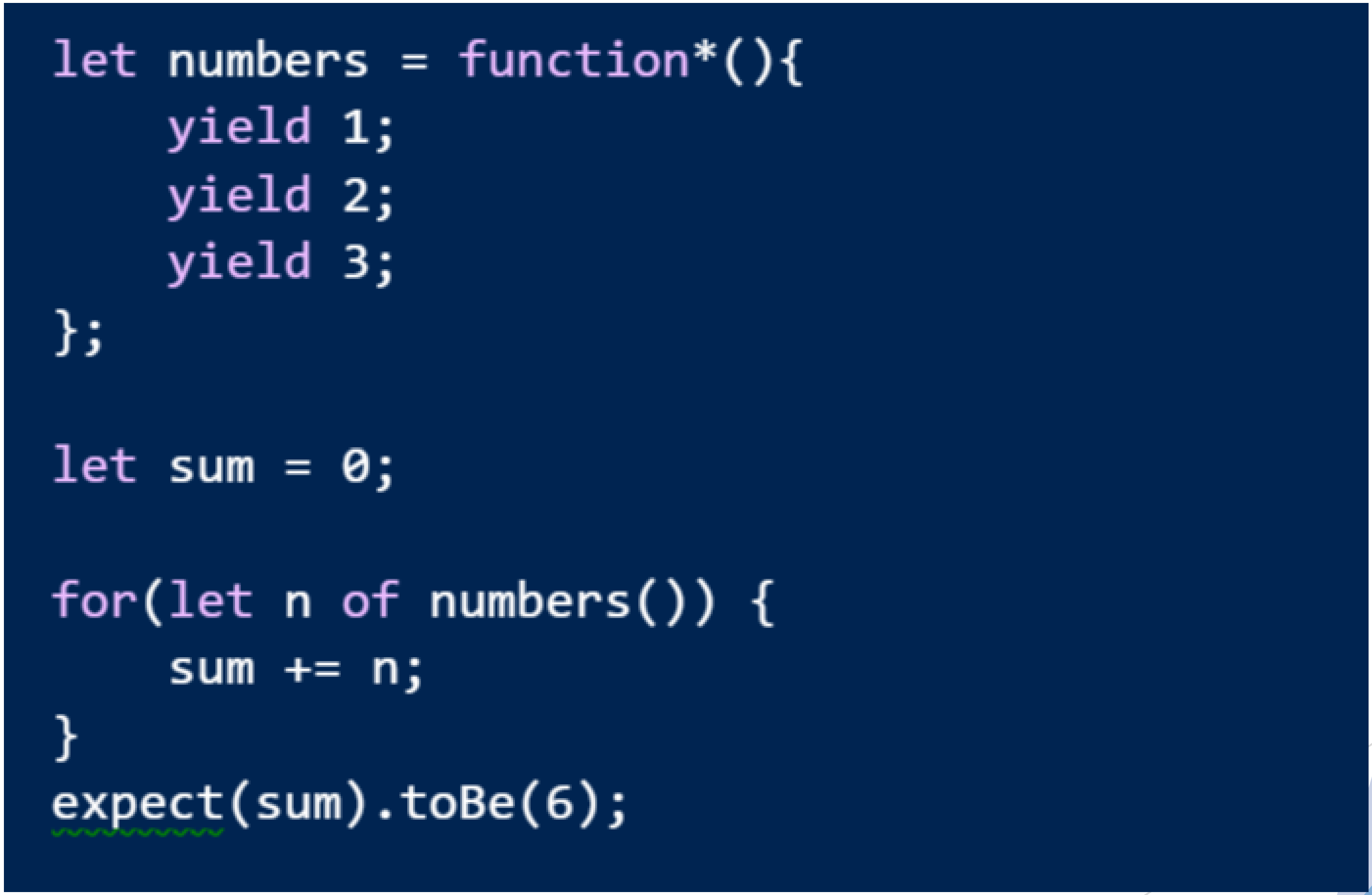
Make Your Own Iterable



```
class Classroom {
```

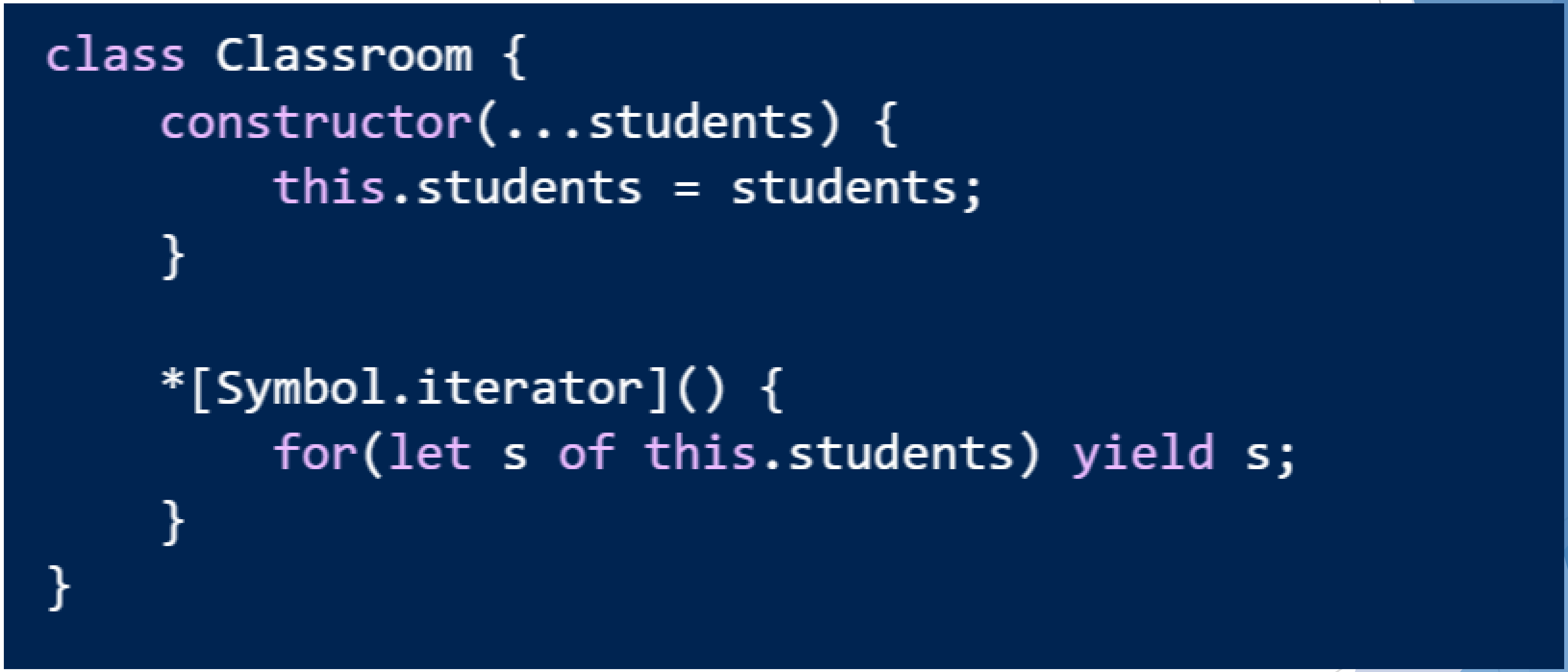
```
class ArrayIterator {
    constructor(array) {
        this.array = array;
        this.index = 0;
    }
    next() {
        let result = { value: undefined, done: true};
        if(this.index < this.array.length) {
            result.value = this.array[this.index];
            result.done = false;
            this.index += 1;
        }
        return result;
    }
}
```

Generators



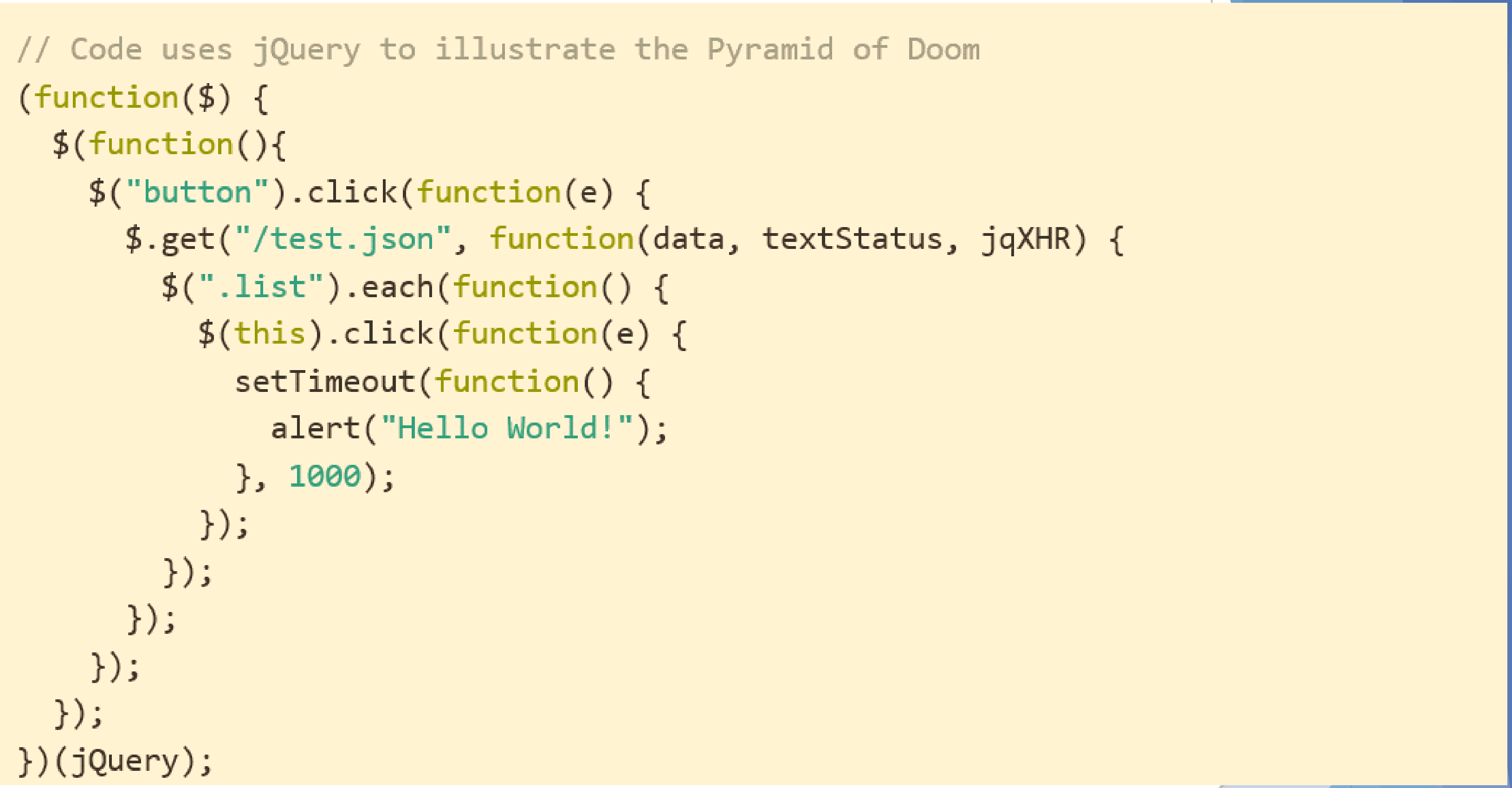
```
let numbers = function*(){
    yield 1;
    yield 2;
    yield 3;
};

let sum = 0;

for(let n of numbers()) {
    sum += n;
}
expect(sum).toBe(6);
```

Easy To Make Iterables



```
class Classroom {
    constructor(...students) {
        this.students = students;
    }

    *[Symbol.iterator]() {
        for(let s of this.students) yield s;
    }
}
```

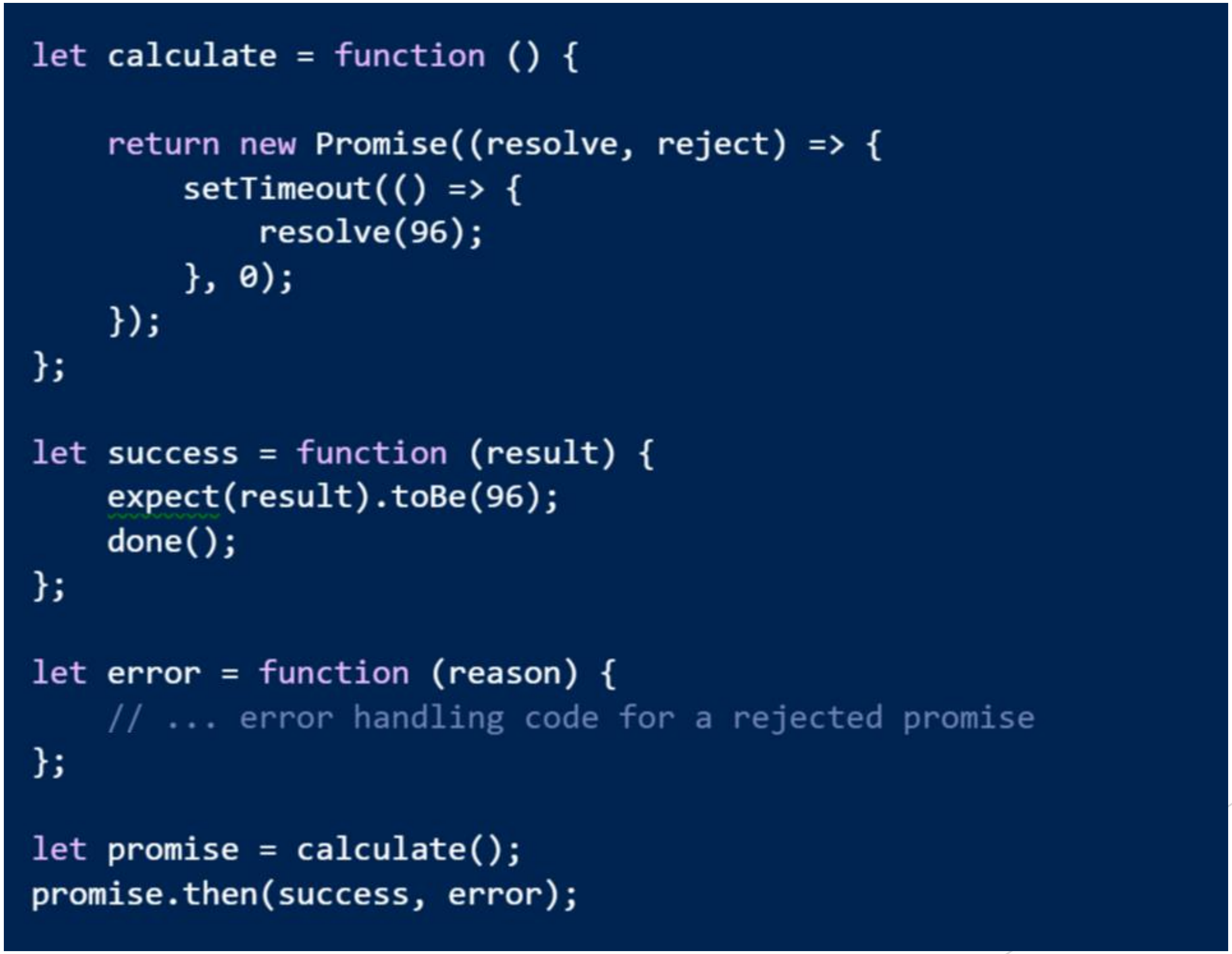
Problem: Async Code



```
// Code uses jQuery to illustrate the Pyramid of Doom
(function($) {
  $(function(){
    $("button").click(function(e) {
      $.get("/test.json", function(data, textStatus, jqXHR) {
        $(".list").each(function() {
          $(this).click(function(e) {
            setTimeout(function() {
              alert("Hello World!");
            }, 1000);
          });
        });
      });
    });
  });
})(jQuery);
```

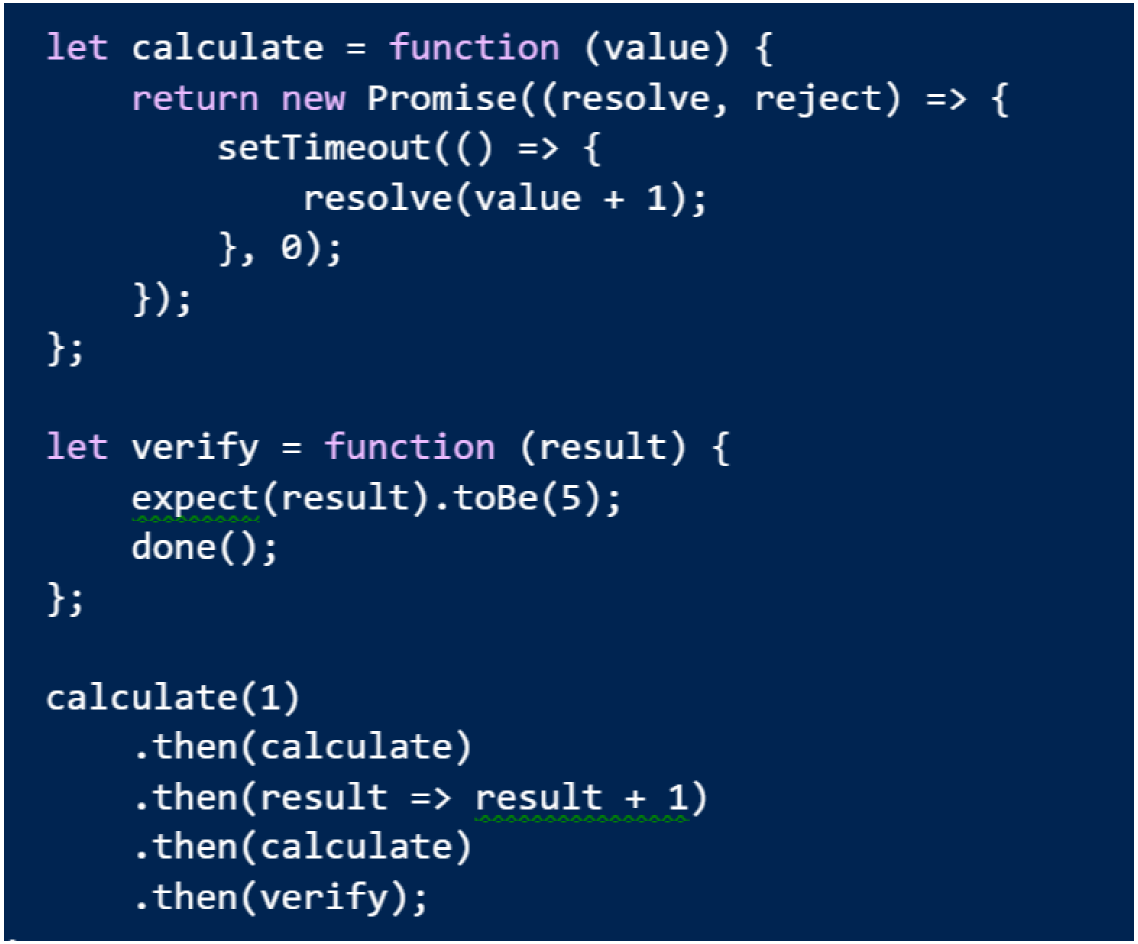
http://tritarget.org/blog/2012/11/28/the-pyramid-of-doom-a-javascript-style-trap/

Solution: Promises



```
let calculate = function () {

    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(96);
        }, 0);
    });
};

let success = function (result) {
    expect(result).toBe(96);
    done();
};

let error = function (reason) {
    // ... error handling code for a rejected promise
};

let promise = calculate();
promise.then(success, error);
```

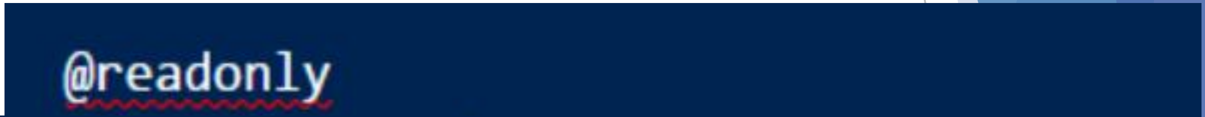
Promises Chain



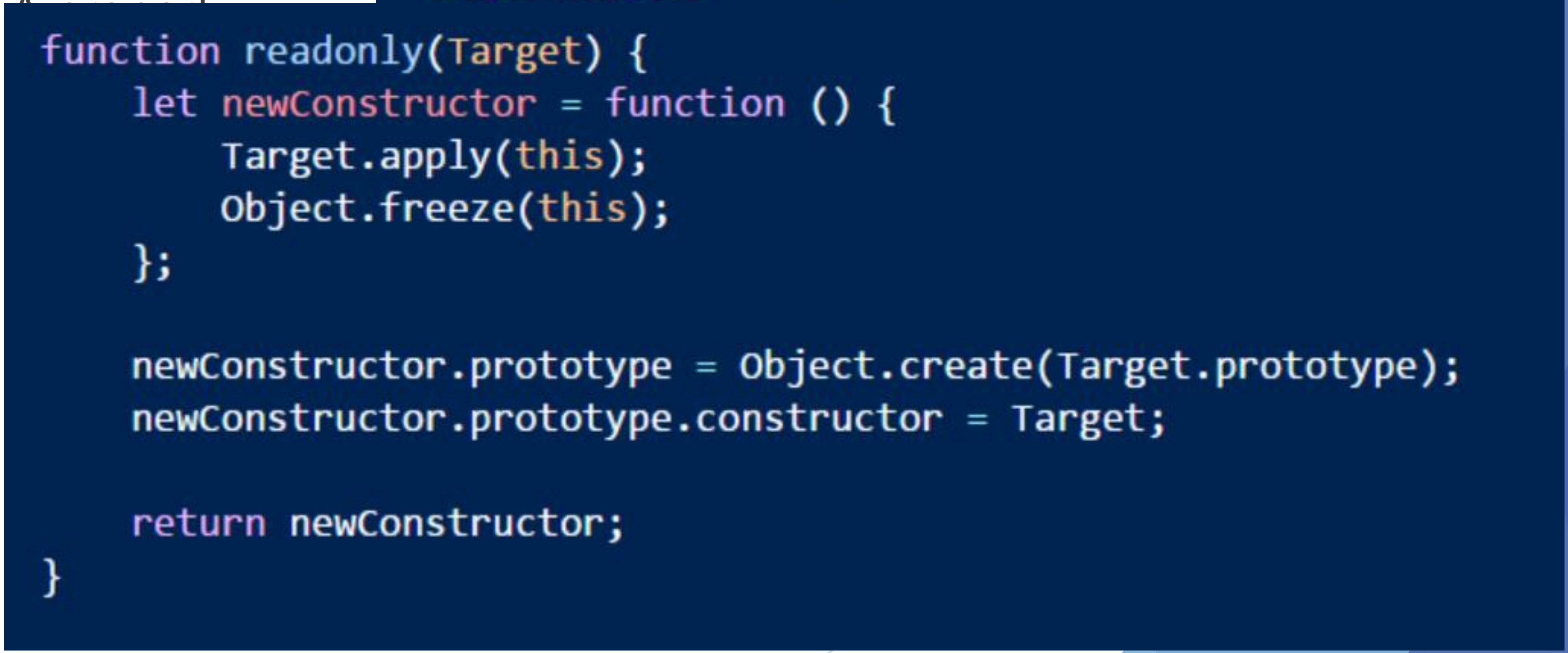
```
let calculate = function (value) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(value + 1);
        }, 0);
    });
};

let verify = function (result) {
    expect(result).toBe(5);
    done();
};

calculate(1)
    .then(calculate)
    .then(result => result + 1)
    .then(calculate)
    .then(verify);
```

Decorators

- @ symbol followed by a function
- Function can modify
  - □ A class
  - □ A
  - □
  - □



```
@readonly
```



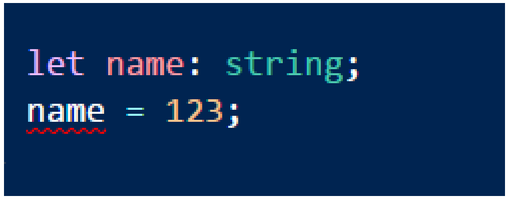
```
function readonly(Target) {
    let newConstructor = function () {
        Target.apply(this);
        Object.freeze(this);
    };

    newConstructor.prototype = Object.create(Target.prototype);
    newConstructor.prototype.constructor = Target;

    return newConstructor;
}
```

TypeScript

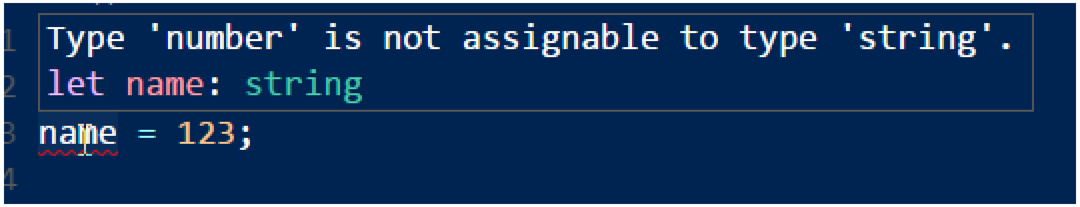
- Designed by Microsoft
  - Anders Hejlsberg
- One of the Official Languages at Google
- Open Source
  - https://github.com/Microsoft/TypeScript
- Superset of JavaScript
  - Adds optional types and interfaces

Type Annotations

- Declare the intended type of a variable
  - □ Default is “any”
  - □ boolean, number, string, Array, enum, void



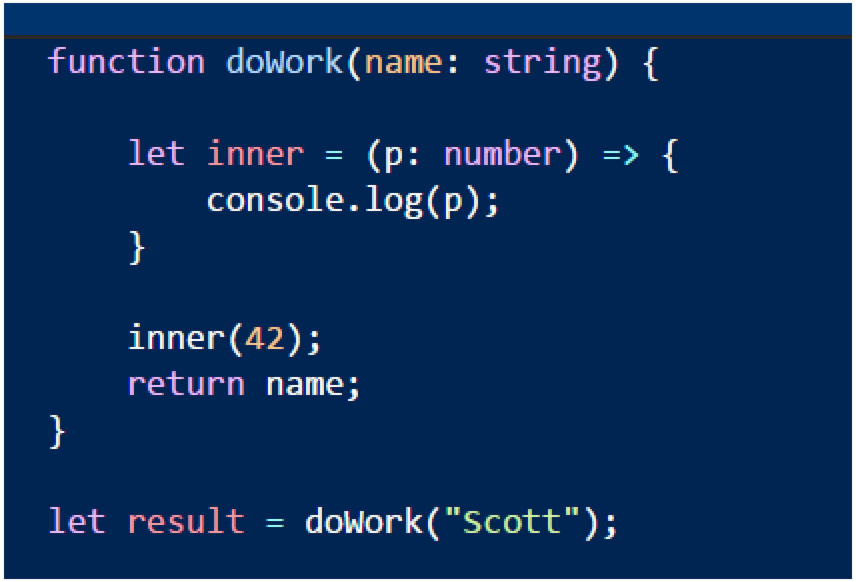
```
let name: string;
name = 123;
```



```
Type 'number' is not assignable to type 'string'.
let name: string
name = 123;
```

Types and Functions

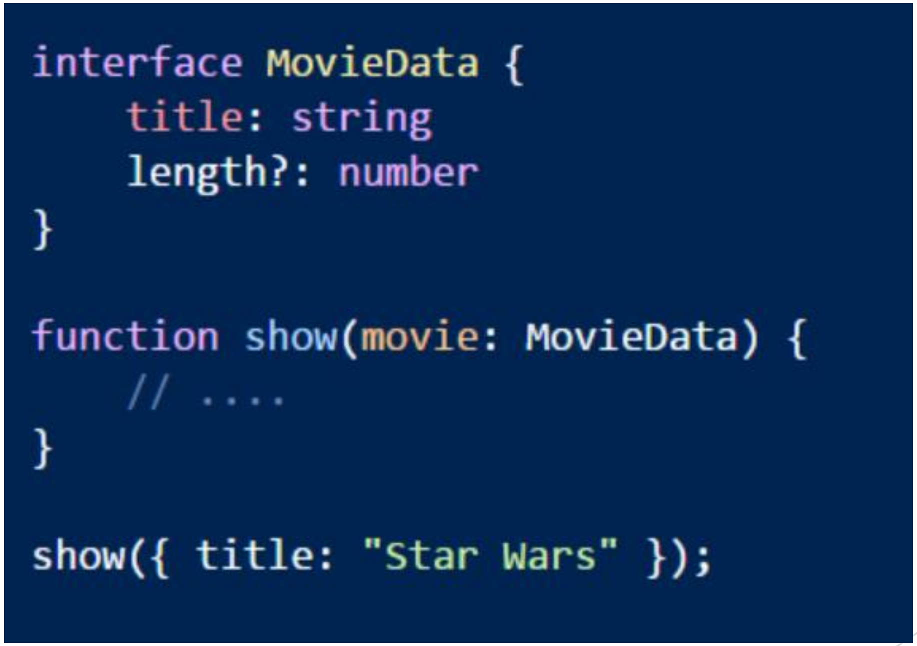
- Parameters can be typed
- Return value can also be typed
  - □ Often can be inferred



```
function doWork(name: string) {

    let inner = (p: number) => {
        console.log(p);
    }

    inner(42);
    return name;
}

let result = doWork("Scott");
```

Interfaces

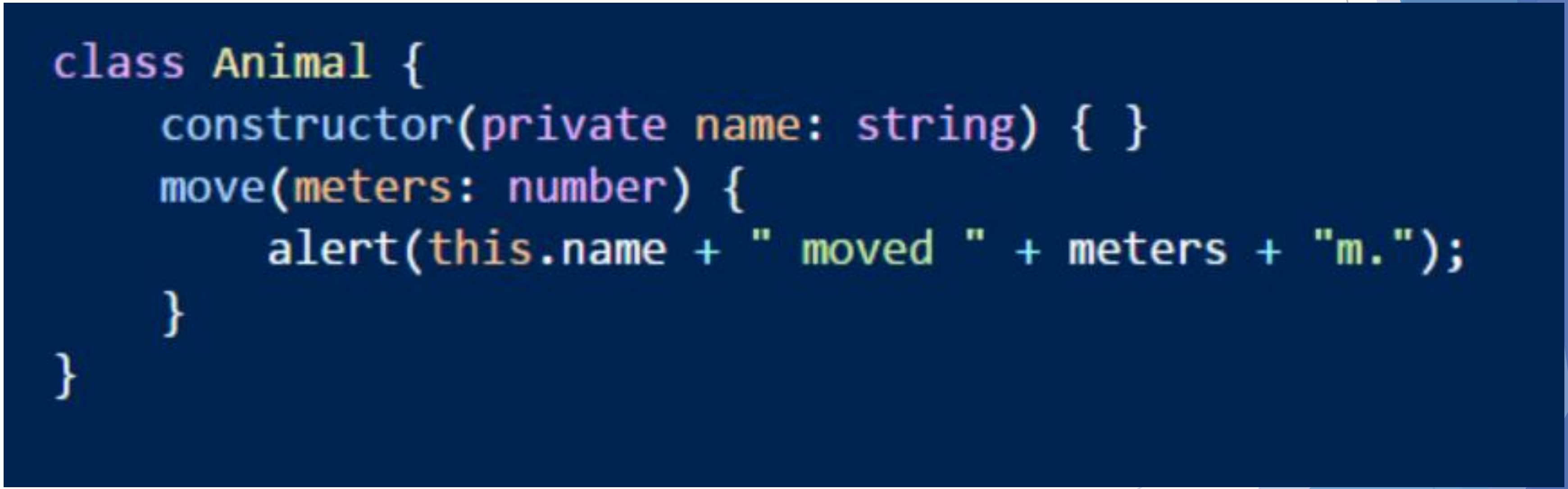
- Focus on the shape
  - □ Allows for duck typing
  - □ Can use optional properties
  - □ Can also describe functions



```
interface MovieData {
    title: string
    length?: number
}

function show(movie: MovieData) {
    // ....
}

show({ title: "Star Wars" });
```

Public, Protected and Private

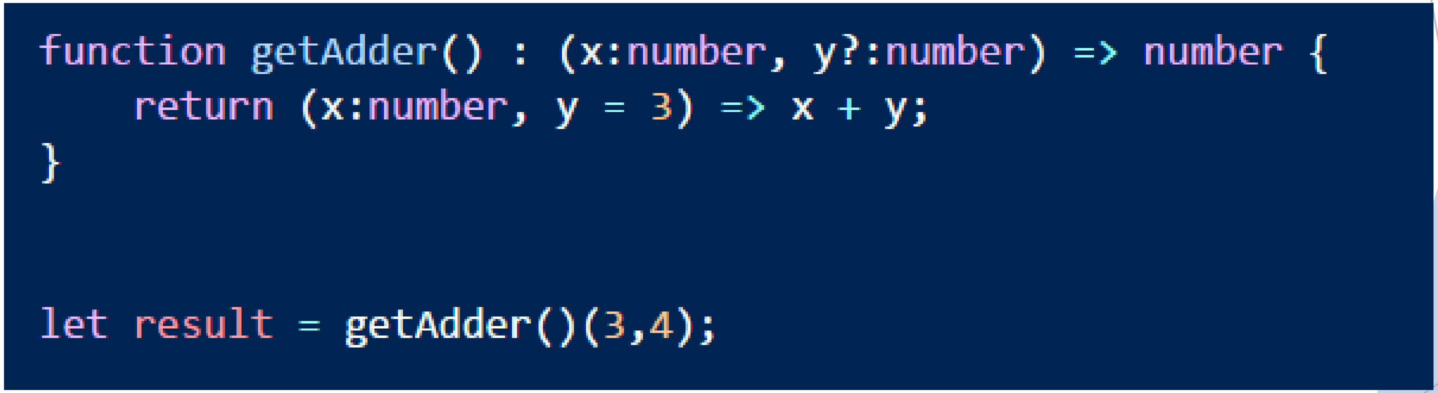
- Public is the default
  - □ Compiler enforces protected and private keyword



```
class Animal {
    constructor(private name: string) { }
    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
```

Functions

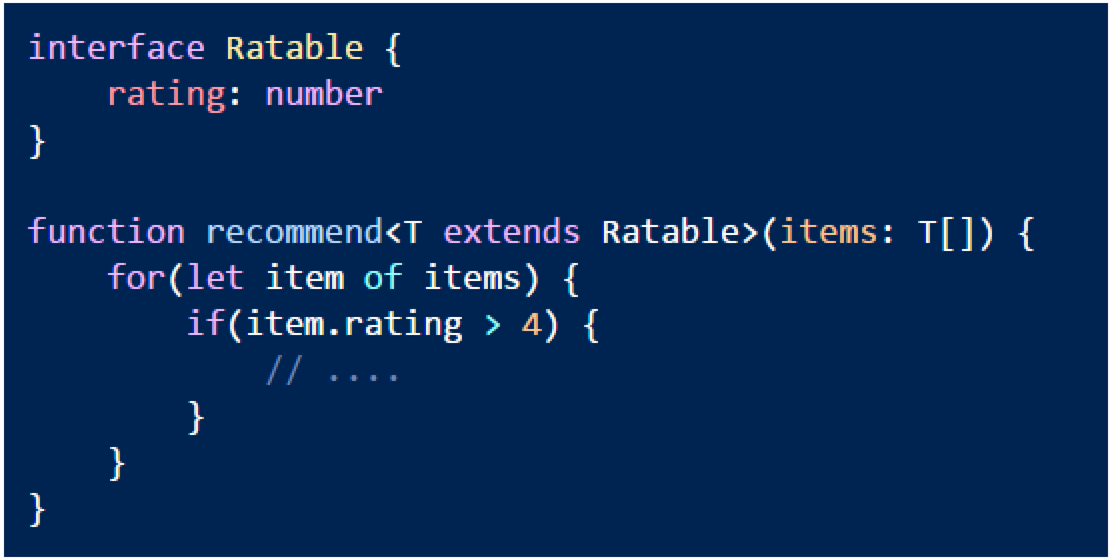
- Can have return types, optional and default parameters



```
function getAdder() : (x:number, y?:number) => number {
    return (x:number, y = 3) => x + y;
}


let result = getAdder()(3,4);
```

Generics

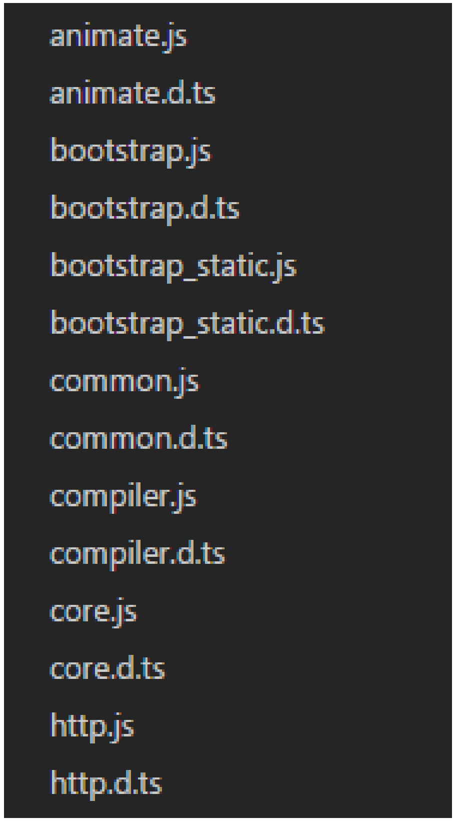
- Use generic types to parameterize a function or class
  - Generic constraints can make type programmable



```
interface Ratable {
    rating: number
}

function recommend<T extends Ratable>(items: T[]) {
    for(let item of items) {
        if(item.rating > 4) {
            // ....
        }
    }
}
```

Declaration Files

- .d.ts files provide type metadata for 3rd parties



```
animate.js
animate.d.ts
bootstrap.js
bootstrap.d.ts
bootstrap_static.js
bootstrap_static.d.ts
common.js
common.d.ts
compiler.js
compiler.d.ts
core.js
core.d.ts
http.js
http.d.ts
```

Summary

- ES2015 is a new language
  - □ Classes, arrow functions, generators, and more
- TypeScript adds optional type annotations
  - □ Types are structural
  - □ Type annotations useful for tooling and compile time checks