HOMEWORK

FRESHER TRAINING COURSE

JS - MODULE

FRESHER TRAINING COURSE

TABLE OF CONTENT

WHAT IS MODULE

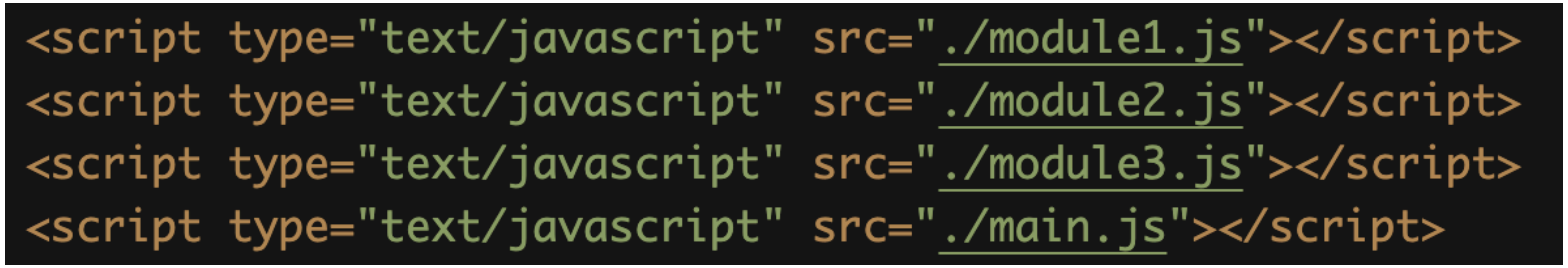
- Modules are just clusters of code, but it should
  - highly self-contained with distinct functionality,
  - can be shuffled, removed, or added as necessary
  - aims to lessen the dependencies on parts of the codebase as much as possible

WHY USING MODULE

- Maintainability
- Avoid namespace pollution
- Reusability

MODULAR SCRIPTING

- Once upon a time

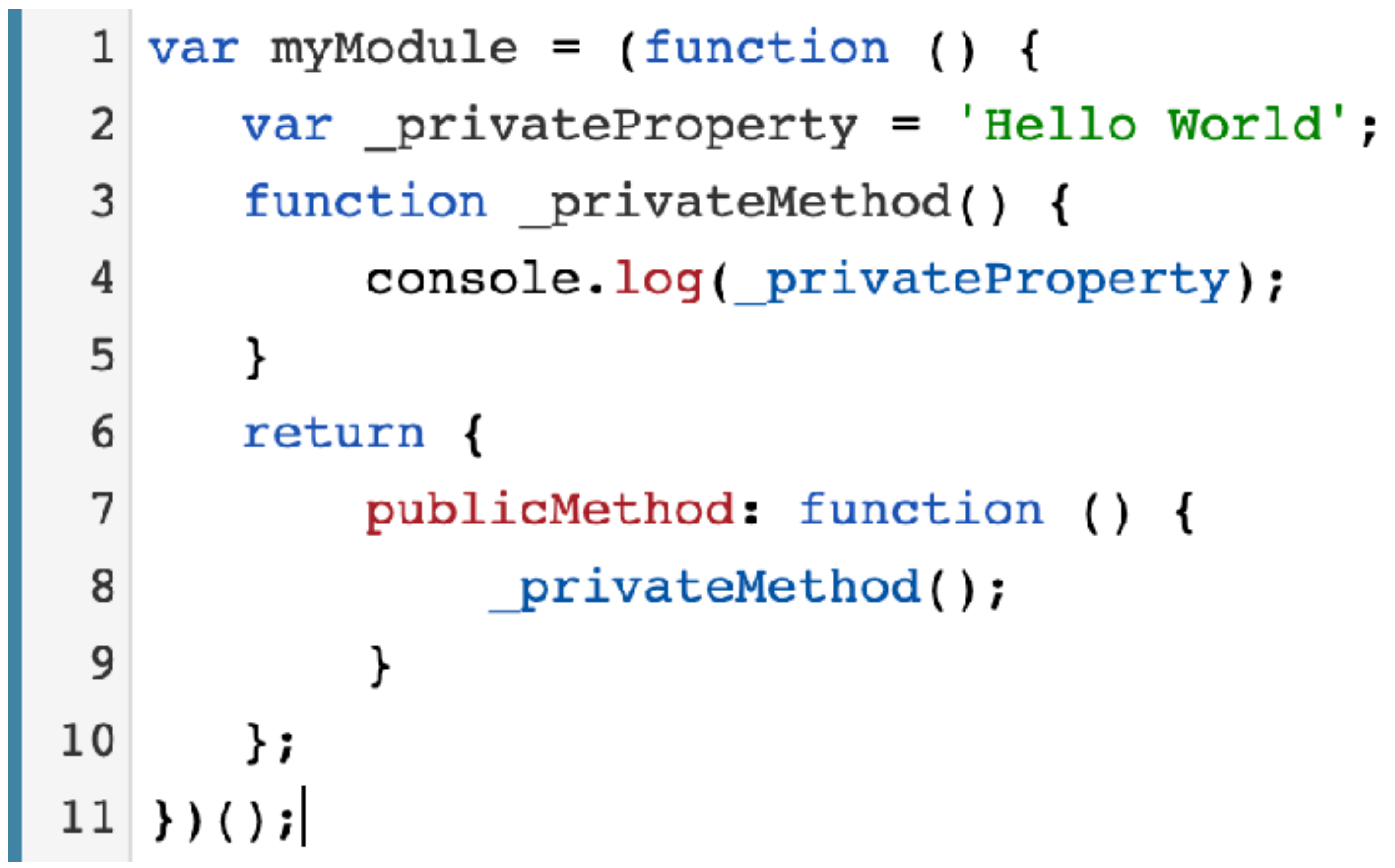


```
<script type="text/javascript" src="./module1.js"></script>
<script type="text/javascript" src="./module2.js"></script>
<script type="text/javascript" src="./module3.js"></script>
<script type="text/javascript" src="./main.js"></script>
```

- **Problem:**
  - Lack of Dependency Resolution
  - Pollution of global namespace.

MODULE PATTERN JAVASCRIPT

- Using an anounymous function to wrapper the variables.
  - => decrease the namespace pollution.



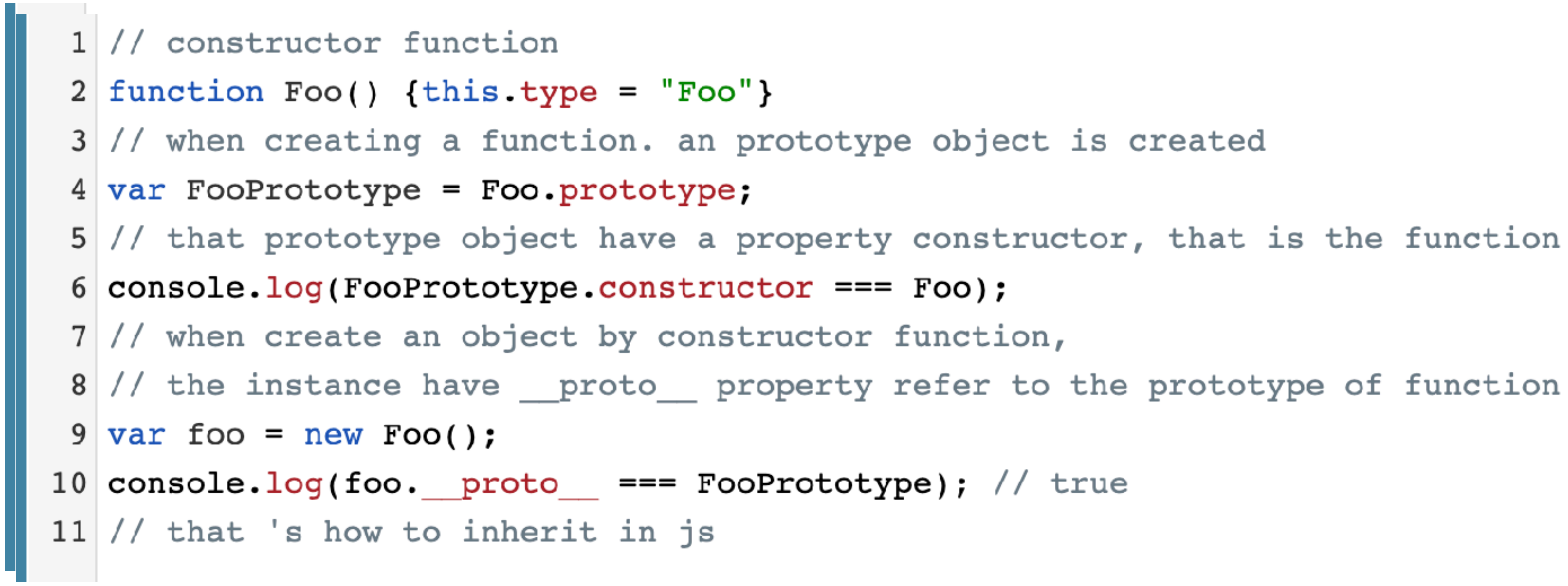
```
var myModule = (function () {
   var _privateProperty = 'Hello World';
   function _privateMethod() {
       console.log(_privateProperty);
   }
   return {
       publicMethod: function () {
           _privateMethod();
       }
   };
})();
```

PROTOTYPE

- When **a function** is created in JavaScript, the JavaScript engine *adds a prototype property to the function.*
- This **prototype property** is an object that *has a constructor property*
- The **constructor property** *points back to* **the function** on which prototype object is a property.
- We can access the function's prototype property using functionName.prototype.

PROTOTYPE – WHEN IT BE CREATED

- object inherits methods and properties from it's prototype.

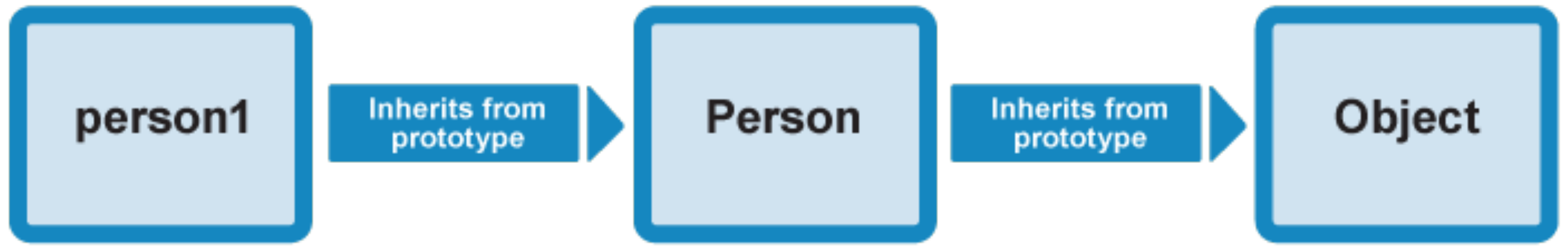


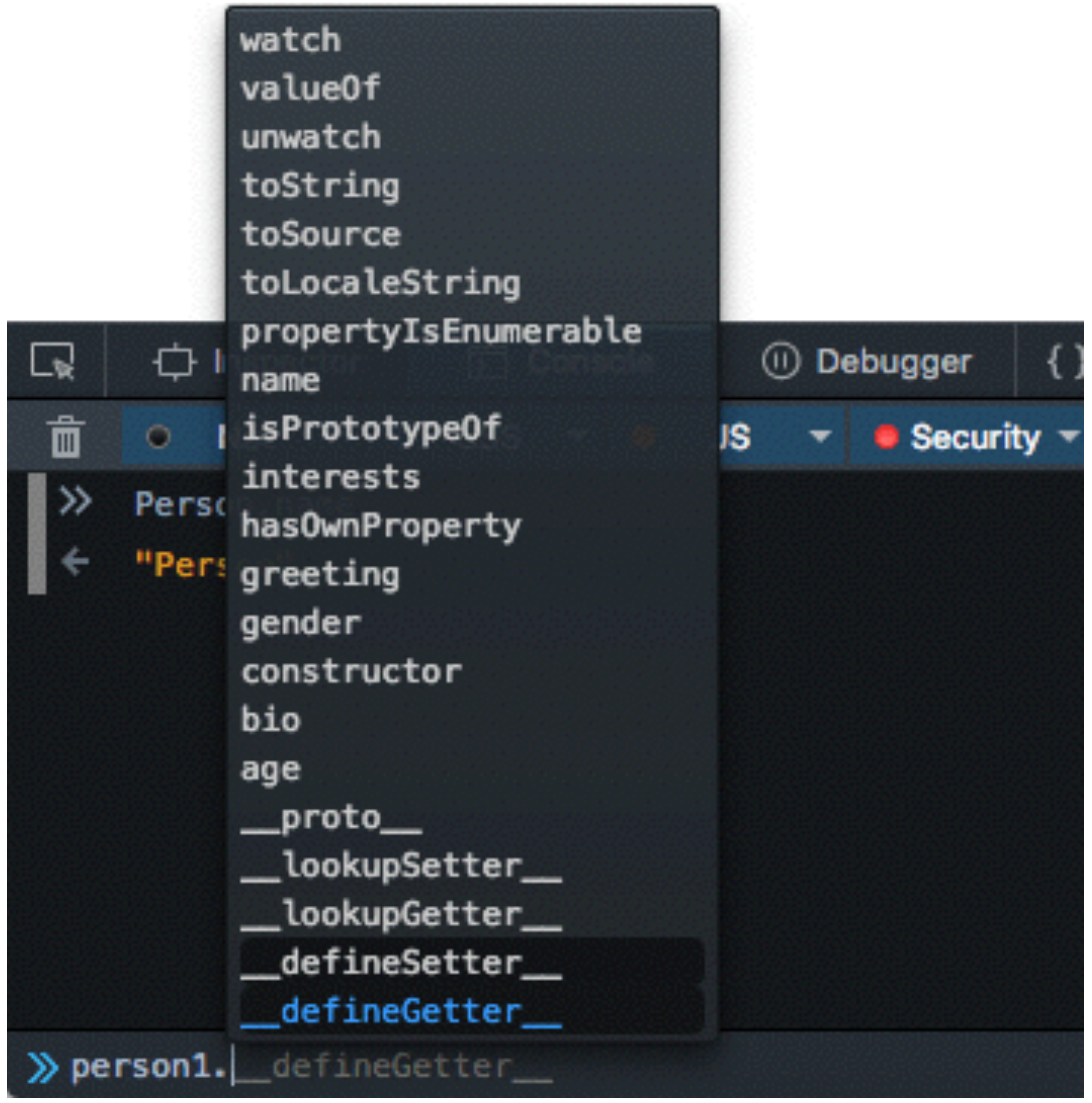
```
// constructor function
function Foo() {this.type = "Foo"}
// when creating a function. an prototype object is created
var FooPrototype = Foo.prototype;
// that prototype object have a property constructor, that is the function
console.log(FooPrototype.constructor === Foo);
// when create an object by constructor function,
// the instance have __proto__ property refer to the prototype of function
var foo = new Foo();
console.log(foo.__proto__ === FooPrototype); // true
// that 's how to inherit in js
```

PROTOTYPE – THE CORE OF INHERITING IN JS

```
function Person(first, last, age) {

  // property and method definitions
  this.name = {
    'first': first,
    'last' : last
  };
  this.age = age;
}
let person1 = new Person('Bob', 'Smith', 32);
```



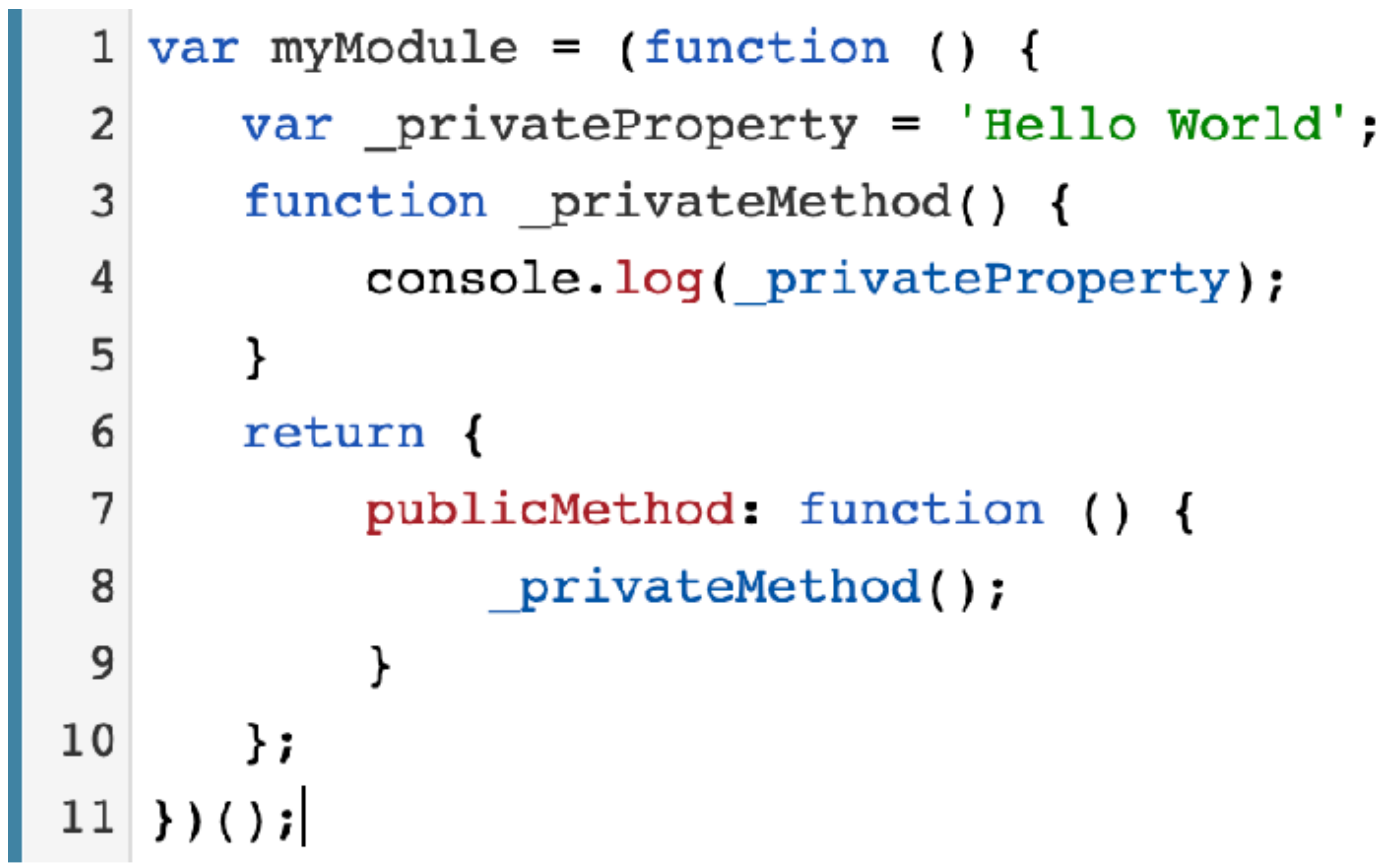


```
watch
valueOf
unwatch
toString
toSource
toLocaleString
propertyIsEnumerable
name
isPrototypeOf
interests
hasOwnProperty
greeting
gender
constructor
bio
age
__proto__
__lookupSetter__
__lookupGetter__
__defineSetter__
__defineGetter__

» person1.__defineGetter__
```

MODULE PATTERN JAVASCRIPT

- Using an anounymous function to wrapper the variables.
  - => decrease the namespace pollution.



```
var myModule = (function () {
    var _privateProperty = 'Hello World';
    function _privateMethod() {
        console.log(_privateProperty);
    }
    return {
        publicMethod: function () {
            _privateMethod();
        }
    };
})();
```

CLASS DECLARATIONS

- Class declarations: using “class” key word.

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

- Function declarations are hoisted but class declarations **are not**

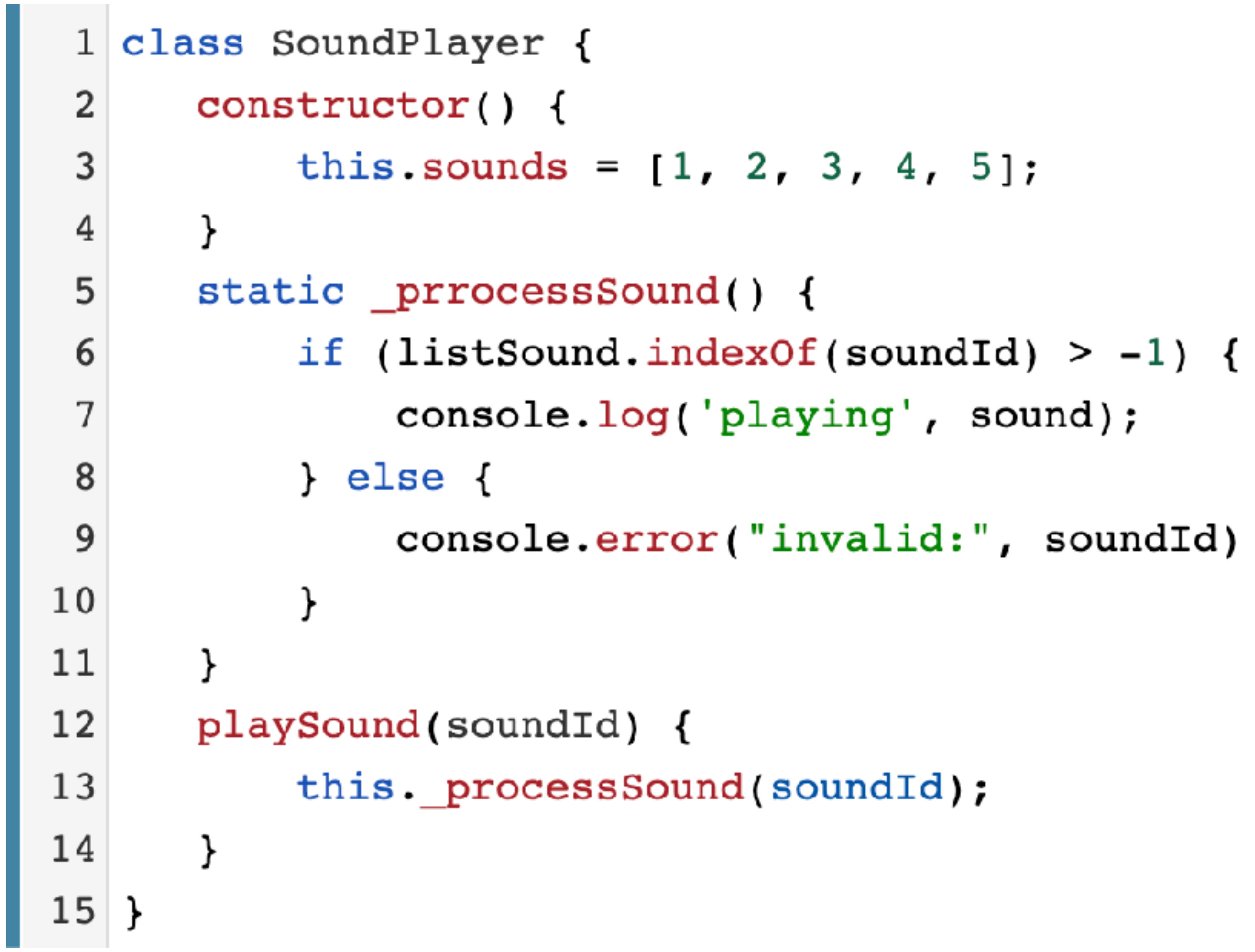
CLASS EXPRESSION

- Class expressions can be named or unnamed

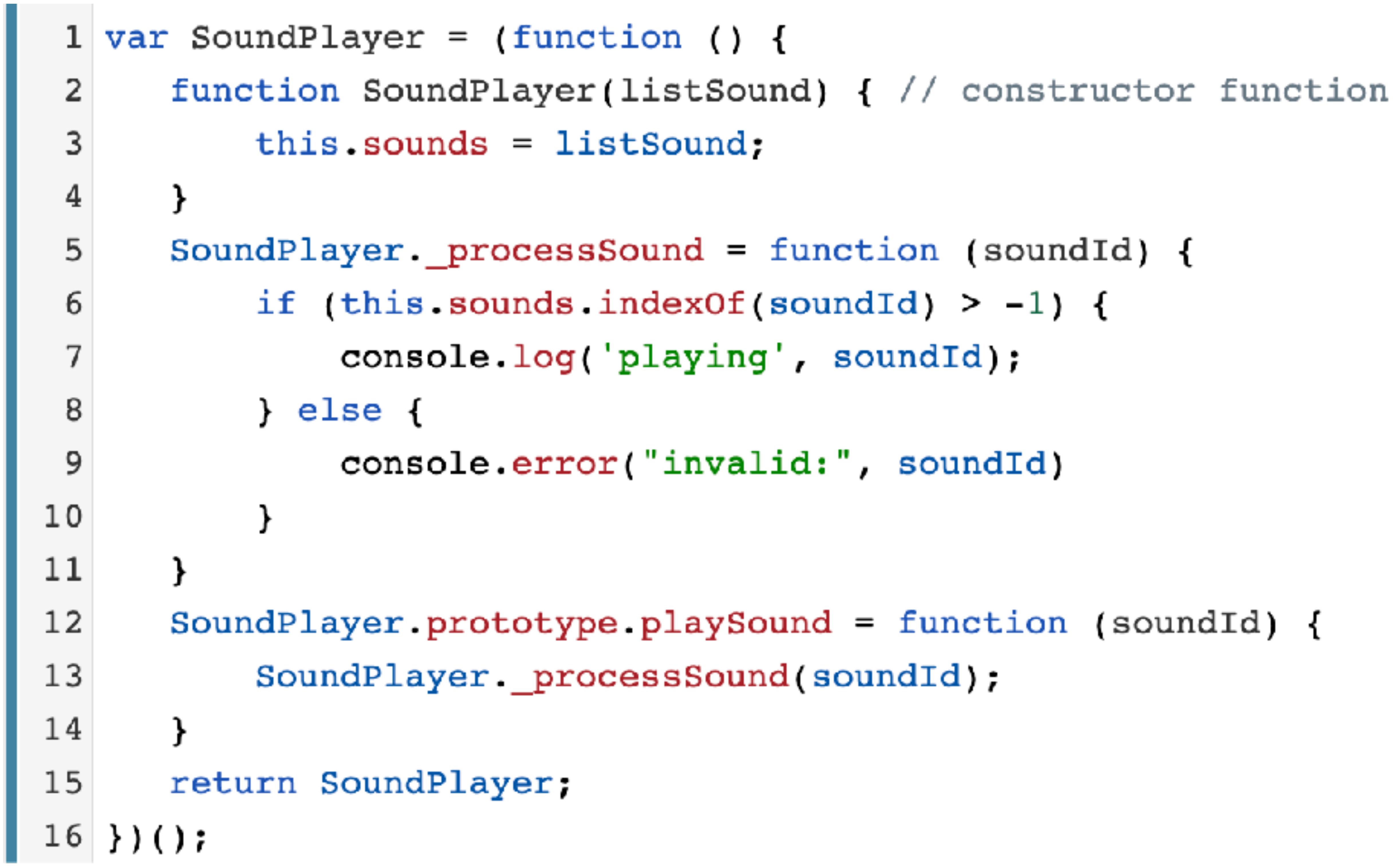
```
// unnamed
let Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
console.log(Rectangle.name);
// output: "Rectangle"
```

```
// named
let Rectangle = class Rectangle2 {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
console.log(Rectangle.name);
// output: "Rectangle2"
```

CLASS VS MODULE PATTERN



```
class SoundPlayer {
    constructor() {
        this.sounds = [1, 2, 3, 4, 5];
    }
    static _prrocessSound() {
        if (listSound.indexOf(soundId) > -1) {
            console.log('playing', sound);
        } else {
            console.error("invalid:", soundId)
        }
    }
    playSound(soundId) {
        this._processSound(soundId);
    }
}
```



```
var SoundPlayer = (function () {
    function SoundPlayer(listSound) { // constructor function
        this.sounds = listSound;
    }
    SoundPlayer._processSound = function (soundId) {
        if (this.sounds.indexOf(soundId) > -1) {
            console.log('playing', soundId);
        } else {
            console.error("invalid:", soundId)
        }
    }
    SoundPlayer.prototype.playSound = function (soundId) {
        SoundPlayer._processSound(soundId);
    }
    return SoundPlayer;
})();
```

CLASS CONSTRUCTOR

- The special method for creating and initializing an object
- It is called when create new object from a class
- Just have one constructor inside a class.

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
const square = new Rectangle(10, 10);
```

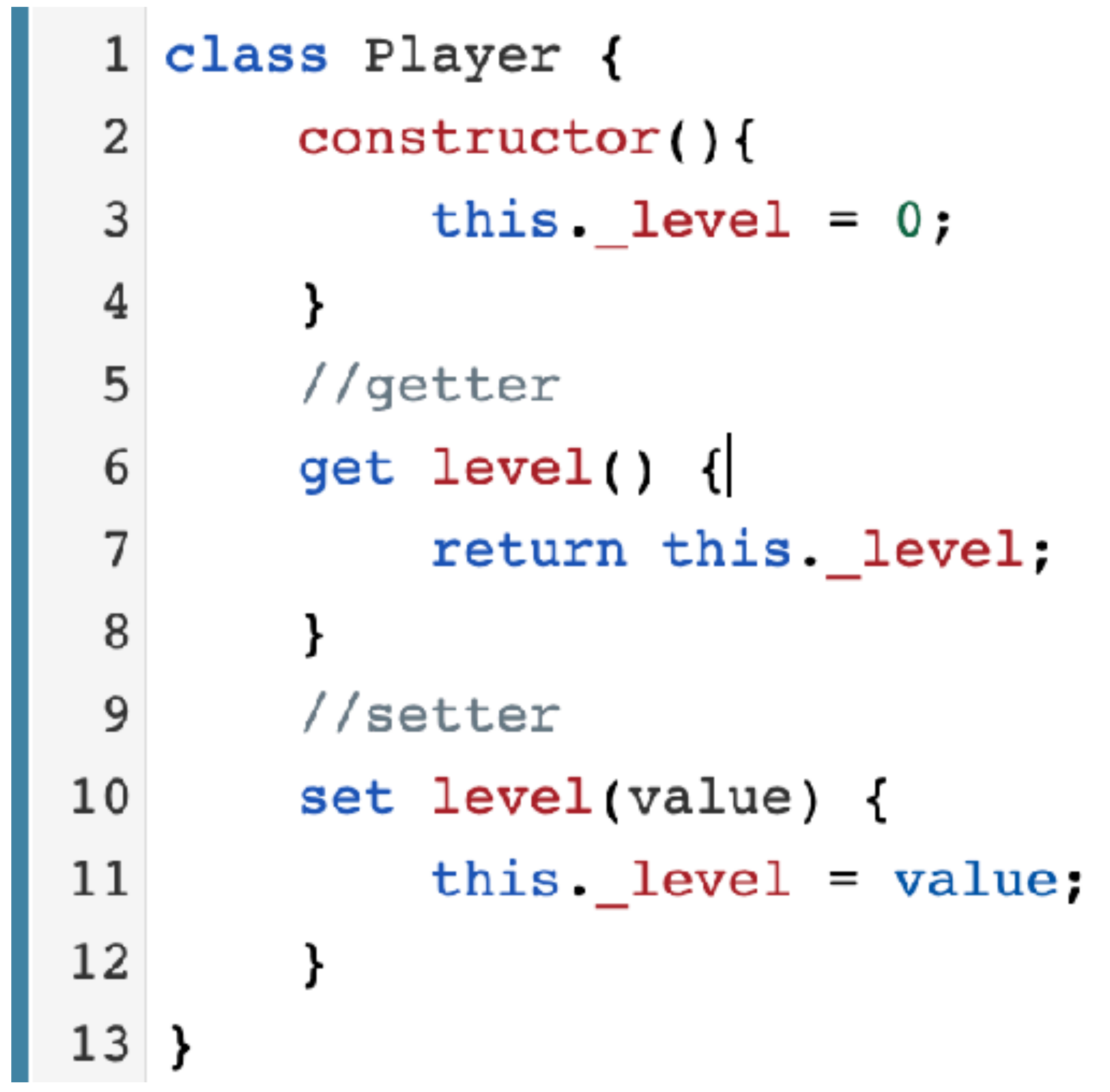
CLASS - INSTANCE PROPERTIES

- The instance properties must define inside of class methods.

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

CLASS – GETTER AND SETTER

- `get` - binds property to a function that will be called when access that property.
- `set` - binds property to a function that will be called that property be set.



```
class Player {
    constructor(){
        this._level = 0;
    }
    //getter
    get level() {
        return this._level;
    }
    //setter
    set level(value) {
        this._level = value;
    }
}
```

CLASS - PROTOTYPE METHODS

- The method **can** be called through a class instance.

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // prototype Method
  calcArea() {
    return this.height * this.width;
  }
}

const square = new Rectangle(10, 10);

console.log(square.area); // 100
```

CLASS - STATIC MEMBERS

- The method **cannot** be called through a class instance.
- a static members (properties and methods) are called without instantiating the class

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  static displayName = "Point";
  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;

    return Math.hypot(dx, dy);
  }
}
```

```
const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
p1.displayName; // undefined
p1.distance;    // undefined
p2.displayName; // undefined
p2.distance;    // undefined

console.log(Point.displayName);      // "Point"
console.log(Point.distance(p1, p2)); // 7.07106
```

CLASS - BINDING THIS

- Binding this for class methods.

```
class Item {
    constructor() {
        this.isClicked = false;
        this.createButton();
    }
    createButton() {
        this.button = document.createElement("button");
        this._onClickFunc = this.onClickButton.bind(this);
        this.button.addEventListener("click", this._onClickFunc);
    }
    onClickButton(){
        this.isClicked = true;
        this.button.removeEventListener("click", this._onClickFunc);
    }
}
```

CLASS - SUBCLASS WITH EXTENDS

- The `extends` keyword is used in *class declarations* or *class expressions* to create a class as a child of another class.

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

  speak() {
    console.log(`${this.name}`);
  }
}
```

```
class Dog extends Animal {
  constructor(name) {
    super(name);
  }

  speak() {
    console.log(`${this.name} barks.`);
  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

CLASS - SUPER CLASS CALL WITH SUPER

- The `super` keyword is used to call corresponding methods of super class. This is one advantage over prototype-based inheritance.

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

  speak() {
    console.log(`${this.name} makes noise.`);
  }
}
```

```
class Lion extends Cat {
  speak() {
    super.speak();
    console.log(`${this.name} roars.`);
  }
}

let l = new Lion('Fuzzy');
l.speak();
// Fuzzy makes a noise.
// Fuzzy roars.
```

IMPORT AND EXPORT

- The `export:` to share functions, objects, or primitive values from the module.
- The im`port:` to read which are **exported** by another module.

```
// Exporting individual features
export let name1, name2, …, nameN
export let name1 = …, name2 = …, …, nameN;
export function functionName(){...}
export class ClassName {...}
```

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export1 } from "module-name";
import { export1 as alias1 } from "module-name";
import { export1 , export2 } from "module-name";
```

REFACTOR CODE