

ECMAScript 2015 (ES6)



Building Modern Web Applications - VSP2019

Karthik Pattabiraman
Kumseok Jung

What is ES6?

1. **What is ES6?**
2. Object-oriented Programming
3. Functional Programming



What is ES6?

- JavaScript specifications are maintained by an international organization - ECMA International
 - ECMA-262 & ISO/IEC-22275
 - ECMAScript is a **living and evolving standard**
 - Goal is to **standardize JS**, as different browser vendors implement different versions: JavaScript, JScript, ActionScript, etc.
 - Current latest edition (as of 2019) is ES10
 - ES5 has been the longest serving standard and still the most prevalent
 - ES6 has gained a lot of momentum and becoming mainstream



ES5 vs ES6

- ES5 still has quirks that create confusion among users
 - Prototypal inheritance
 - Semantics of keywords like: `var`, `this`
- ES6 introduces many useful features
 - Syntactic sugar for commonly used code patterns
 - Better support for object-oriented programming
 - Better support for functional programming
- Good coverage of ES6 features can be found at:
 - <http://es6-features.org>
 - <https://github.com/lukehoban/es6features>
- In this class we will focus on a subset of the ES6 features



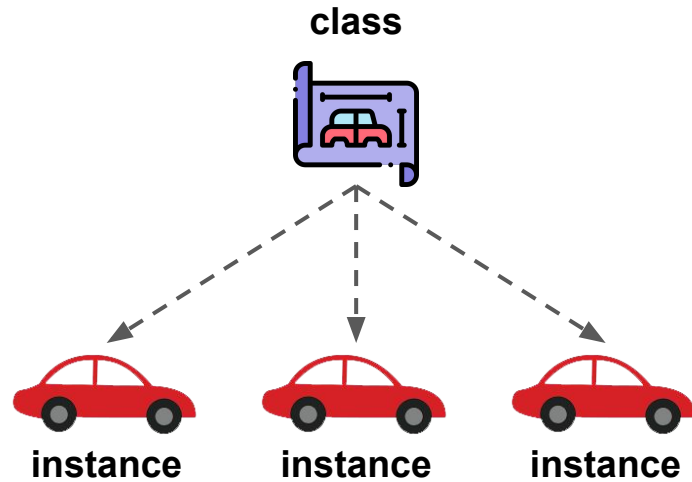
Object-oriented Programming

1. What is ES6?
- 2. Object-oriented Programming**
3. Functional Programming



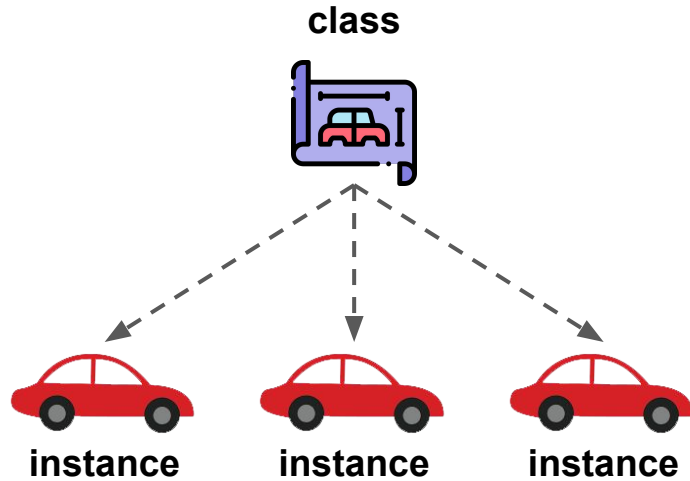
Object-oriented Programming

Object-oriented

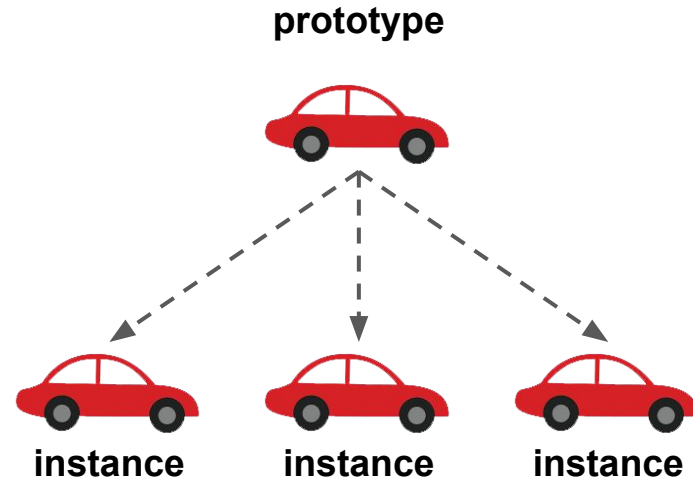


Object-oriented Programming

Object-oriented



Prototypal



Object-oriented Programming

- JavaScript is still prototypal at its core
- Prototypes can emulate OOP patterns
 - However, it is syntactically and semantically different
- ES6 introduces the `class` keyword to support OOP



Object-oriented Programming

New keywords introduced in this chapter

- `new` : for creating an instance of an Object
- `this` : for referencing the function invocation context
- `instanceof` : for checking whether A is an instance of B
- `class` : ES6 keyword for declaring a Class
- `constructor` : for defining the constructor function for a class
- `extends` : ES6 keyword for extending/inheriting from a Class
- `super` : ES6 keyword for referencing the superclass



Object-oriented Programming

this keyword

- `this` refers to the object on which the function is called



```
1 var myCar = {  
2   name: "Smart",  
3   power: 1,  
4   velocity: 0,  
5   accelerate: function (fuel){  
6     this.velocity += fuel * this.power;  
7   }  
8 }  
9  
10 myCar.accelerate(10);
```

Object-oriented Programming

`this` keyword

- `this` refers to the object on which the function is called



```
1 function accelerate (fuel){  
2   this.velocity += fuel * this.power;  
3 }  
4 var myCar = {  
5   name: "Smart",  
6   power: 1,  
7   velocity: 0,  
8   accelerate: accelerate  
9 }  
10 myCar.accelerate(10);  
11
```

Object-oriented Programming

this keyword

- `this` refers to the object on which the function is called



```
1 function accelerate (fuel){
2   this.velocity += fuel * this.power;
3 }
4 var myCar = {
5   name: "Smart",
6   power: 1,
7   velocity: 0,
8   accelerate: accelerate
9 }
10 myCar.accelerate(10);
11 accelerate(12);           // What is "this"?
```

Object-oriented Programming

this keyword

- Function objects have a method called `bind`, which can be used to “lock” what `this` refers to



```
1 function accelerate (fuel){
2   this.velocity += fuel * this.power;
3 }
4 var myCar = {
5   name: "Smart",
6   power: 1,
7   velocity: 0,
8   accelerate: accelerate
9 }
10 myCar.accelerate(10);
11 accelerate.bind(myCar)(12);           // What is "this"?
```

Object-oriented Programming

new keyword

```
1 function Car (){  
2  
3  
4 };  
5 Car.prototype = {  
6   name: "Smart",  
7   power: 1,  
8   velocity: 0  
9 };  
10  
11  
12  
13  
14
```



Object-oriented Programming

new keyword

```
1 function Car (name, power=1){  
2     this.name = name;  
3     this.power = power;  
4     this.velocity = 0;  
5 };  
6 Car.prototype = {  
7     name: "Smart",  
8     power: 1,  
9     velocity: 0  
10 };  
11  
12  
13  
14
```



Object-oriented Programming

new keyword

```
1 function Car (name, power=1){  
2     this.name = name;  
3     this.power = power;  
4     this.velocity = 0;  
5 };  
6 Car.prototype = {};  
7  
8  
9  
10  
11  
12  
13  
14
```



Object-oriented Programming

new keyword

```
1 function Car (name, power=1){
2   this.name = name;
3   this.power = power;
4   this.velocity = 0;
5 };
6 Car.prototype = {};
7 Car.prototype.accelerate = function(fuel){
8   this.velocity += fuel * this.power;
9 };
10
11
12
13
14
```



Object-oriented Programming

new keyword

```
1 function Car (name, power=1){
2   this.name = name;
3   this.power = power;
4   this.velocity = 0;
5 };
6 Car.prototype = {};
7 Car.prototype.accelerate = function(fuel){
8   this.velocity += fuel * this.power;
9 };
10
11 var myCar = new Car("Smart");
12 myCar.accelerate(10);
13
14
```



Object-oriented Programming

new keyword

- Invoking `new Foo(arg1, arg2)` will perform the following:
 - Create a new object by shallow-copying `Foo.prototype`; we will refer to this new object as `newFoo` in this slide
 - Since this is a “shallow copy”, functions and objects bound to `newFoo`’s properties will all point to the corresponding objects on `Foo.prototype`.
(e.g., `myCar.accelerate === Car.prototype.accelerate`)
 - Invoke the `Foo` function in the context of the newly created object `newFoo` (i.e., `Foo.bind(newFoo)(arg1, arg2)`); we thus refer to the `Foo` function as “a constructor”



Object-oriented Programming

Object-oriented

```
1 class Car {  
2     constructor (name, power=1){  
3         this.name = name;  
4         this.power = power;  
5         this.velocity = 0;  
6     }  
7     accelerate (fuel){  
8         this.velocity  
9         += fuel * this.power;  
10    }  
11 }  
12 var myCar = new Car("Smart");  
13 myCar.accelerate(10);  
14
```

Prototypal



Object-oriented Programming

Object-oriented

```
1 class Car {  
2   constructor (name, power=1){  
3     this.name = name;  
4     this.power = power;  
5     this.velocity = 0;  
6   }  
7   accelerate (fuel){  
8     this.velocity  
9     += fuel * this.power;  
10  }  
11 }  
12 var myCar = new Car("Smart");  
13 myCar.accelerate(10);  
14
```

Prototypal

```
1 function Car (name, power=1){  
2   this.name = name;  
3   this.power = power;  
4   this.velocity = 0;  
5 };  
6 Car.prototype.accelerate  
7   = function(fuel){  
8     this.velocity  
9     += fuel * this.power;  
10  };  
11  
12 var myCar = new Car("Smart");  
13 myCar.accelerate(10);  
14
```



Object-oriented Programming

class and constructor keyword

```
1 class Car {  
2     constructor (name, power=1){  
3         this.name = name;  
4         this.power = power;  
5         this.velocity = 0;  
6     }  
7     accelerate (fuel){  
8         this.velocity  
9         += fuel * this.power;  
10    }  
11 }  
12  
13 var myCar = new Car("Smart");  
14 myCar.accelerate(10);
```



Class Activity: Defining a Class



[lectures/lecture-6/activity1.js](https://github.com/lectures/lecture-6/activity1.js)

- Define a class named “Thing” and implement the following:
 - The constructor accepts a single argument `id`, and initializes 2 instance properties `id` and `live`. The property `id` is set to the argument `id` and `live` is set to `false`
 - `printStatus` method, printing in the format “{id} [on|off]” using `console.log`
 - `powerOn` method, setting `live` property to `true`
 - `powerOff` method, setting `live` property to `false`



```
1 class Thing {
2     // To implement
3 }
4
5 var thing = new Thing("thing-0");
6 thing.printStatus();    // prints: thing-0 (off)
7 thing.powerOn();
8 thing.printStatus();    // prints: thing-0 (on)
```

Object-oriented Programming

extends and super keyword

```
1 class RacingCar extends Car {  
2     constructor (name){  
3         super(name, 3.5);  
4     }  
5  
6     turbo (fuel){  
7         this.velocity += fuel * this.power * 1.5;  
8     }  
9  
10 }  
11  
12  
13  
14
```



Object-oriented Programming

extends and super keyword

```
1 class RacingCar extends Car {  
2     constructor (name){  
3         super(name, 3.5);  
4     }  
5  
6     turbo (fuel){  
7         this.velocity += fuel * this.power * 1.5;  
8     }  
9  
10 }  
11  
12 var superCar = new RacingCar("F1");  
13 superCar.accelerate(10);  
14 superCar.turbo(5);
```



Class Activity: Inheritance



[lectures/lecture-6/activity2.js](#)



- Implement the classes `Sensor` and `Actuator`, which inherits from the `Thing` class from the previous activity
 - `Sensor` and `Actuator` should, in addition to calling the superclass constructor, initialize a property `value` to `null`
 - `Sensor` should have its own method `readValue`. If `live` is `true`, it should set the `value` property to a random value and return it. Else, it should return `null`
 - `Actuator` should have its own method `writeValue`, taking in a single argument `val`. If `live` is `true`, it should set the `value` property to `val`. Else, it should do nothing
 - Override the `printStatus` method as below:
 - For `Sensors`, it should print in the format “`{id} [on|off] -> {value}`”
 - For `Actuators`, it should print in the format “`{id} [on|off] <- {value}`”

Functional Programming

1. What is ES6?
2. Object-oriented Programming
3. **Functional Programming**



Functional Programming

- JavaScript supports functional programming
- When used appropriately, **functions** can implement pure functions
 - Except it is not actually a pure function
 - Keywords like **this**, **arguments** make JavaScript functions impure
- ES6 introduces **arrow functions** to support real functional programming



Functional Programming

- Arrow functions are **not replacements** for ES5 functions
- Arrow functions are **anonymous functions**
- `this` and `arguments` inside arrow functions are lexically bound



Functional Programming

- Arrow functions are **not replacements** for ES5 functions
- Arrow functions are **anonymous functions**
- **this** and **arguments** inside arrow functions are lexically bound



Syntax Example:

```
1 (radius, height) => {  
2   return radius * radius * Math.PI * height;  
3 }  
4  
5 (radius, height) => (radius * radius * Math.PI * height);
```

Functional Programming

- Pure functions

- Always returns the same value given the same arguments
- Have no side effects like mutating an external object (e.g., I/O, network resource, variables outside of its scope)
- Examples:
 - area of circle, distance between 2 points in 3-dimensional space

- Impure functions

- Might depend on an external context
- Might change an external object
- Examples:
 - `Date.now()`
 - `console.log()`



Functional Programming

Arrow function syntax

```
1 // Regular function
2 function(arg1, arg2){
3     // do some stuff here
4     return arg1 + arg2;
5 }
6
7 // Imperative usage
8 (arg1, arg2) => {
9     // do some stuff here
10    return arg1 + arg2;
11 }
12
13 // Pure function
14 (arg1, arg2) => (arg1 + arg2);
```



Functional Programming

Regular ES5 Function

```
1 var f = function (g, x, y){  
2   var gx = g(x);  
3   var gy = g(y);  
4   var result = gx + gy;  
5   return result;  
6 }
```

ES6 Arrow Function

```
1 var f = (g, x, y)=> {  
2   var gx = g(x);  
3   var gy = g(y);  
4   var result = gx + gy;  
5   return result;  
6 };
```



Functional Programming

Regular ES5 Function

```
1 var f = function (g, x, y){  
2   return g(x) + g(y);  
3 }  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14
```

ES6 Arrow Function

```
1 var f = (g, x, y)=>(g(x)+g(y));  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14
```



Functional Programming

Regular ES5 Function

```
1 var u = function(f){  
2   return function(x){  
3     return f(x, u(f));  
4   }  
5 }  
6  
7  
8  
9  
10  
11  
12  
13  
14
```

ES6 Arrow Function

```
1 var u = f=> x=> f(x, u(f));  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14
```



Functional Programming

Regular ES5 Function

```
1  var Y = function(f){
2    return (function(x){
3      return x(x);
4    })(function(y){
5      return f(function(x){
6        return y(y)(x);
7      });
8    });
9  }
```

10
11
12
13
14

ES6 Arrow Function

```
1  var Y = f=>
2    (x=> x(x))(y=> f(x=> y(y)(x)));
3
4
5
6
7
8
9
10
11
12
13
14
```



Class Activity: Rewriting Code with Arrow Functions



[lectures/lecture-6/activity3.js](#)



```
1 var fib = function(n){
2   if (n > 1) return fib(n-1) + fib(n-2);
3   else return 1;
4 }
5
6
7
8
9
10
11
12
13
14
```

Class Activity: Rewriting Code with Arrow Functions

Solution



[lectures/lecture-6/activity3.js](https://github.com/lectures/lecture-6/activity3.js)



```
1  var fib = function(n){
2      if (n > 1) return fib(n-1) + fib(n-2);
3      else return 1;
4  }
5
6  var fib = n=> (n > 1 ? fib(n-1) + fib(n-2) : 1);
7
8
9
10
11
12
13
14
```

Class Activity: Rewriting Code with Arrow Functions



[lectures/lecture-6/activity3.js](#)



```
1 var fib = function(n){
2   if (n > 1) return fib(n-1) + fib(n-2);
3   else return 1;
4 }
5
6 var fib = n=> (n > 1 ? fib(n-1) + fib(n-2) : 1);
7
8 var factorial ?
9
10
11
12
13
14
```

Class Activity: Rewriting Code with Arrow Functions



[lectures/lecture-6/activity3.js](#)



```
1 var fib = function(n){
2   if (n > 1) return fib(n-1) + fib(n-2);
3   else return 1;
4 }
5
6 var fib = n=> (n > 1 ? fib(n-1) + fib(n-2) : 1);
7
8 var factorial = n=> (n > 1 ? n * factorial(n-1) : 1);
9
10
11
12
13
14
```


Functional Programming

- Arrow Function usage scenario

```
1 class Timer {
2   constructor () {
3     this.seconds = 0;
4     this.reference = null;
5   }
6   start () {
7     this.reference = setInterval(function() {
8       this.seconds += 1;
9     }, 1000);
10  }
11  stop () {
12    clearInterval(this.reference);
13  }
14 }
```



Functional Programming

- Arrow Function usage scenario

```
1 class Timer {  
2     constructor () {  
3         this.seconds = 0;  
4         this.reference = null;  
5     }  
6     start () {  
7         var self = this;  
8         this.reference = setInterval(function () {  
9             self.seconds += 1;  
10        }, 1000);  
11    }  
12    stop () {  
13        clearInterval(this.reference);  
14    }  
15 }
```



Functional Programming

- Arrow Function usage scenario

```
1 class Timer {  
2   constructor () {  
3     this.seconds = 0;  
4     this.reference = null;  
5   }  
6   start () {  
7     this.reference = setInterval(() => {  
8       this.seconds += 1;  
9     }, 1000);  
10  }  
11  stop () {  
12    clearInterval(this.reference);  
13  }  
14 }
```



Class Activity: Rewriting Code with Arrow Functions

Find the problem in the following code and fix it



[lectures/lecture-6/activity4.js](https://github.com/UBC-CPSC/lectures/lecture-6/activity4.js)



```
1 class User {
2   constructor (username){
3     this.id = username;
4   }
5
6   readAllSensors (things){
7     var mine = things.filter(function(thing){
8       return (thing.owner === this.id && thing instanceof Sensor);
9     });
10    // ... more code
11  }
12 }
13
14
```

Class Activity: Rewriting Code with Arrow Functions

Solution

```
1 class User {  
2   constructor (username){  
3     this.id = username;  
4   }  
5  
6   readAllSensors (things){  
7     var mine = things.filter(thing =>  
8       (thing.owner === this.id && thing instanceof Sensor));  
9     // ... more code  
10  }  
11 }  
12  
13  
14
```

