ES6 Features

CPEN 400A

Object-oriented Programming

`class` and `constructor` keyword

```
class Car {
  constructor (name, power=1){
    this.name = name;
    this.power = power;
    this.velocity = 0;
  }
  accelerate (fuel){
    this.velocity
      += fuel * this.power;
  }
}

var myCar = new Car("Smart");
myCar.accelerate(10);
```

Object-oriented Programming

`extends` and `super` keyword

```
class RacingCar extends Car {
  constructor (name){
    super(name, 3.5);
  }

  turbo (fuel){
    this.velocity += fuel * this.power * 1.5;
  }

}

var superCar = new RacingCar("F1");
superCar.accelerate(10);
superCar.turbo(5);
```

Functional Programming

- JavaScript supports functional programming
- When used appropriately, `function`s 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

Syntax Example:

```
(radius, height) => {
   return radius * radius * Math.PI * height;
}

(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

```
// Regular function
function(arg1, arg2){
   // do some stuff here
   return arg1 + arg2;
}

// Imperative usage
(arg1, arg2) => {
   // do some stuff here
   return arg1 + arg2;
}

// Pure function
(arg1, arg2) => (arg1 + arg2);
```

Functional Programming

- Arrow Function usage scenario

```
class Timer {
  constructor (){
    this.seconds = 0;
    this.reference = null;
  }
  start (){
    this.reference = setInterval(function(){
      this.seconds += 1;
    }, 1000);
  }
  stop (){
    clearInterval(this.reference);
  }
}
```

Functional Programming

- Arrow Function usage scenario

```
class Timer {
  constructor (){
    this.seconds = 0;
    this.reference = null;
  }
  start (){
    var self = this;
    this.reference = setInterval(function(){
      self.seconds += 1;
    }, 1000);
  }
  stop (){
    clearInterval(this.reference);
  }
}
```

Functional Programming

- Arrow Function usage scenario

```
class Timer {
  constructor (){
    this.seconds = 0;
    this.reference = null;
  }
  start (){
    this.reference = setInterval(()=> {
      this.seconds += 1;
    }, 1000);
  }
  stop (){
    clearInterval(this.reference);
  }
}
```

What is a Promise

- Promise is a new built-in object **introduced in ES6**
- Provides a **cleaner interface** for handling **asynchronous operations**
- When multiple asynchronous operations need to be made, the **callback** pattern **becomes hard to follow**
    - Scope of variables in multiple nested closures
    - Error handling for each of the callback steps

Promise

- `Promise` is an object with the following methods
  - `then (onResolve, onReject)`: used to register resolve and reject callbacks
  - `catch (onReject)`: used to register reject callback
  - `finally (onComplete)`: used to register settlement callback
- `Promise` will be in one of the three states: pending, resolved, rejected
- `Promise` also has static methods
  - `resolve (value)`: returns a `Promise` that resolves immediately to `value`
  - `reject (error)`: returns a `Promise` that rejects immediately to `error`
  - `all (promises)`: returns a `Promise` that resolves when all promises resolve
  - `race (promises)`: returns a `Promise` that resolves if any of the promises resolve

Promise

- Creating a `Promise` object
  - `new Promise(func)`: The `Promise` constructor expects a single argument *`func`*, which is a function with 2 arguments: `resolve`, `reject`
  - `resolve` and `reject` are callback functions for emitting the result of the operation
    - `resolve(result)` to emit the result of a successful operation
    - `reject(error)` to emit the error from a failed operation

```
var action = new Promise((resolve, reject)=> {
   setTimeout(()=> {
      if (Math.random() > 0.5) resolve("Success!");
      else reject(new Error("LowValueError"));
   }, 1000);
});

```

Promise

- Using the result of a `Promise` fulfillment through the `then` method
    - `then(onResolve, onReject)`: used to register callbacks for handling the result of the `Promise`. It returns another `Promise`, making this function **chainable**
    - `onResolve` is called **if the previous `Promise` resolves**; it receives the resolved value as the only argument
    - `onReject` is called **if the previous `Promise` rejects** or **throws an error**; it receives the rejected value or the error object as the only argument

```
action.then(
    (result)=> console.log(result), // result: "Success!"
    (error)=> console.log(error)    // error: Error("LowValueError")
)
.then(()=> console.log("A"))
.then(()=> console.log("B"));
```

Promise

- The `catch` method is used to handle the result of a rejected `Promise`
    - `catch(onReject)`: used to register a callback for handling the result of the failed `Promise`. It returns another `Promise`, making this function **chainable**
    - `onReject` is called **if the previous `Promise` rejects** or **throws an error**; it receives the rejected value or the error object as the only argument

```
action.then(
   (result)=> console.log(result), // result: "Success!"
   (error)=> console.log(error)    // error: Error("LowValueError")
)
.catch((err)=> console.log(err));

```

Promise

- The `finally` method is used to register a callback to be called when a `Promise` is settled, regardless of the result
  - `finally(onComplete)`: It returns another `Promise`, making this function **chainable**
  - `onComplete` is called **if the previous `Promise` is settled**

```
action.then(
   (result)=> console.log(result), // result: "Success!"
   (error)=> console.log(error)    // error: Error("LowValueError")
)
.catch((err)=> console.log(err))
.finally(()=> console.log("The End!"));
```

Promise

- The static functions `Promise.resolve` and `Promise.reject` are used to create a `Promise` object that immediately resolves or rejects with the given data
    - Useful when the next asynchronous operation expects a `Promise` object

```
action.then(
   (result)=> console.log(result), // result: "Success!"
   (error)=> console.log(error)    // error: Error("LowValueError")
)
.catch((err)=> console.log(err))
.finally(()=> console.log("The End!"));
```

Promise

- The return values of the callback functions given to `then`, `catch`, and `finally` method are wrapped as a resolved `Promise`, if it is not already a `Promise`

```javascript
action.then(
    (result)=> {
        return "Action Resolved"
    },
    (error)=> {
        return "Action Rejected"
    })
.then((result)=> console.log("Success: " + result),
    (error)=> console.log("Error: " + error.message));

// if action resolves, what is printed? what if it rejects?
```

Promise

- Using the static function `Promise.all`, we can wait for multiple concurrent `Promise`s to be resolved (sort of like joining threads)
  - `Promise.all` accepts an Array of promises and returns a `Promise` that resolves to an array of results (in the same order as the promises given)

```
var multi = Promise.all([
   new Promise((resolve)=> setTimeout(()=> resolve("A"), 2000)),
   new Promise((resolve)=> setTimeout(()=> resolve("B"), 3000)),
   new Promise((resolve)=> setTimeout(()=> resolve("C"), 1000)),
]);

multi.then(
   (results)=> console.log(results),
   (error)=> console.log(error));

```

Promise

- Using the static function `Promise.race`, we can retrieve the first `Promise` to resolve out of a set of concurrent `Promises`
    - `Promise.race` accepts an Array of promises and returns the first `Promise` that resolves

```
var multi = Promise.race([
   new Promise((resolve)=> setTimeout(()=> resolve("A"), 2000)),
   new Promise((resolve)=> setTimeout(()=> resolve("B"), 3000)),
   new Promise((resolve)=> setTimeout(()=> resolve("C"), 1000)),
]);

multi.then(
   (result)=> console.log(result),
   (error)=> console.log(error));

```