ADVANCED JAVASCRIPT

CLASSES, OOPS AND CLOSURES

NEW AND THIS KEYWORDS

- The new operator allows us to create a new instance of a user-defined object type or of one of the built-in object types that has a constructor function.
- ‘this’ refers to the calling object.

- In the given code snippet, we create a new object of the given function using the 'new' keyword.
- 'batman' now contains an object of Superhero class.

```
16  function Superhero(name, age, villains) {
17      this.name = name
18      this.age = age
19      this.villains = villains
20  }
21
22  let batman = new Superhero('Batman', 30, ['Joker', 'Penguin', 'Deathstroke'])
23  console.log(batman)
```

Note that the batman object has 'Superhero' written before the object definition. This indicates that it is an instance of Superhero.

```
Superhero {
  name: 'Batman',
  age: 30,
  villains: [ 'Joker', 'Penguin', 'Deathstroke' ]
}
```

PROTOTYPES

- Property inheritance.
- Object.create() function

```
let p = {
    a: 10
}

let q = Object.create(p)
q.b = 20

let r = Object.create(q)
r.c = 30

console.log(p)
console.log(q)
console.log(r)
console.log(r.c)
console.log(r.b)
console.log(r.a)
console.log(r.__proto__ === q)
```

```
{ a: 10 }
{ b: 20 }
{ c: 30 }
30
20
10
true
```

HIGHER ORDER FUNCTIONS

- Functions that accept another function as argument or that return another function
- Eg - Filter, map, sort, reduce

FUNCTION THAT ACCEPTS OTHER FUNCTIONS AS ARGUMENTS

```
1   function formalGreeting() {
2       console.log("How are you?");
3   }
4   function casualGreeting() {
5       console.log("What's up?");
6   }
7   function greet(type, greetFormal, greetCasual) {
8       if(type === 'formal') {
9           greetFormal();
10      } else if(type === 'casual') {
11          greetCasual();
12      }
13  }
14
15  greet('formal', formalGreeting, casualGreeting)
16  greet('casual', formalGreeting, casualGreeting)
```

FUNCTION THAT RETURNS ANOTHER FUNCTION

```
1   function adder(x) {
2       return (y) => {
3           return x + y;
4       }
5   }
6
7   const fourAdder = adder(4)
8   console.log(fourAdder(5))
9
10  const tenAdder = adder(10)
11  console.log(tenAdder(20))
```

```
9
30
```

FILTER FUNCTION

- The filter() method creates an array filled with all array elements that pass a test (provided as a function).

```
let arr = [1, 5, 4, 3, 6, 8, 7, 2]

let even = arr.filter((x) => {
    return x % 2 == 0;
})

let odd = arr.filter((x) => {
    return x % 2 == 1;
})

console.log(even)
console.log(odd)
```

```
[ 4, 6, 8, 2 ]
[ 1, 5, 3, 7 ]
```

MAP FUNCTION

- The map() method creates a new array with the results of calling a function for every array element.

```
let arr = [1, 5, 4, 3, 6, 8, 7, 2]

let squares = arr.map((x) => {
    return x * x;
})

console.log(squares)
```

```
[
  1, 25, 16, 9,
 36, 64, 49, 4
]
```

REDUCE FUNCTION

- The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in single output value.

```
let arr = [1, 2, 3, 4]

let totalSum = arr.reduce((accumulate, currentValue) => {
    return accumulate + currentValue;
})

console.log(totalSum)
```

```
10
```

SETTIMEOUT FUNCTION

- Executes the given function after a delay of specified time.
- The time argument is passed as number of milliseconds
- (1000 milliseconds = 1 second)

```
14  setTimeout(() => {
15      console.log('Timeout after 5 seconds')
16  },5000)
17
```

SETINTERVAL FUNCTION

- Executes the given function infinitely after specified interval of time.
- The given example snippet will print ‘hello’ infinitely after every one second.

```
10  setInterval(() => {
11      console.log('hello')
12  }, 1000)
```

- The execution of setInterval function can be stopped by clearInterval function.

```
9   let count = 0;
10  let intervalId = setInterval(() => {
11      console.log('Hello ',count)
12      count++;
13      if (count >= 5) {
14          clearInterval(intervalId)
15      }
16  }, 1000)
```

```
Hello  0
Hello  1
Hello  2
Hello  3
Hello  4
```

CLASSES

- Classes are in fact "special functions", and just as you can define function expressions and function declarations.

```
49  class Superhero {
50      constructor(name, age, villains) {
51          this.name = name
52          this.age = age
53          this.villains = villains
54      }
55
56      speak(dialogue) {
57          console.log(dialogue)
58      }
59  }
60
61  let batman = new Superhero('Batman', 30, ['Joker', 'Penguin', 'Deathstroke'])
62  console.log(batman)
63  batman.speak('You either die a hero, or live long enough to see yourself become a villain.')
```

Note that this is extremely similar to the object created using function with the new keyword (covered in previous lecture). That's because they are essentially the same thing.

```
Superhero {
  name: 'Batman',
  age: 30,
  villains: [ 'Joker', 'Penguin', 'Deathstroke' ]
}
You either die a hero, or live long enough to see yourself become a villain.
```

INHERITANCE

```
15 class Superhero {
16     constructor(name, age, villains) {
17         this.name = name
18         this.age = age
19         this.villains = villains
20     }
21
22     speak(dialogue) {
23         console.log(dialogue)
24     }
25 }
26
27 class Avenger extends Superhero {
28     constructor(name, age, villains, species) {
29         super(name, age, villains)
30         this.species = species
31     }
32 }
33
34 let thor = new Avenger('Thor', 1000, ['Surtur', 'Gorr', 'Malekith'], 'Asgardian')
35 let ironman = new Avenger('Tony Stark', 35, ['Iron Monger', 'Mandarin'], 'Human')
36
37
38 thor.speak('You're big. I've fought bigger.')
39 console.log(ironman.age)
40
```

```
You're big. I've fought bigger.
35
```

CLOSURES

A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created.

```
function incrementCreator() {
    let counter = 0     //Should execute only once
    return function () {
        counter++;
        return counter;
    };
}

const increment = incrementCreator()

console.log(increment())
console.log(increment())
console.log(increment())
```

```
1
2
3
```

CALLBACKS

- Callback functions are functions are used to maintain synchronization in async functions. They are passed as arguments to async functions and called after async process is finished.

```javascript
function startTimer(callback) {
    setTimeout(() => {
        console.log("Timer of 2 seconds");
        callback();
    }, 2000);
}

function afterTimer() {
    console.log("Timer finished");
}

startTimer(afterTimer);
```

```
Timer of 2 seconds
Timer finished
```

CALL STACK

JavaScript Call Stack is a mechanism to keep track of the function calls.