# ECMAScript Day01 Part01

Axle Barr

## Topics

# Typical function

```
const num = [5,6];
function x(a,b){
  let sum = a + b;
  return sum;
}
//
console.log(x(num[0], num[1]))
//shows 11 as expected
```

## Default Parameters

# Requires 3 parameters but got 2

```
const num = [5,6];
function x(a,b,c){
  let sum = a + b;
  return sum;
}
//
console.log(x(num[0], num[1]))
//shows 11 no errors
```

## Default Parameters

# Requires 3 parameters but got 2

```
const num = [5,6];
function x(a,b,c){
  let sum = a + b + c;
  return sum;
}
//
console.log(x(num[0], num[1]))
//shows NaN, third parameter not passed
```

## Default Parameters

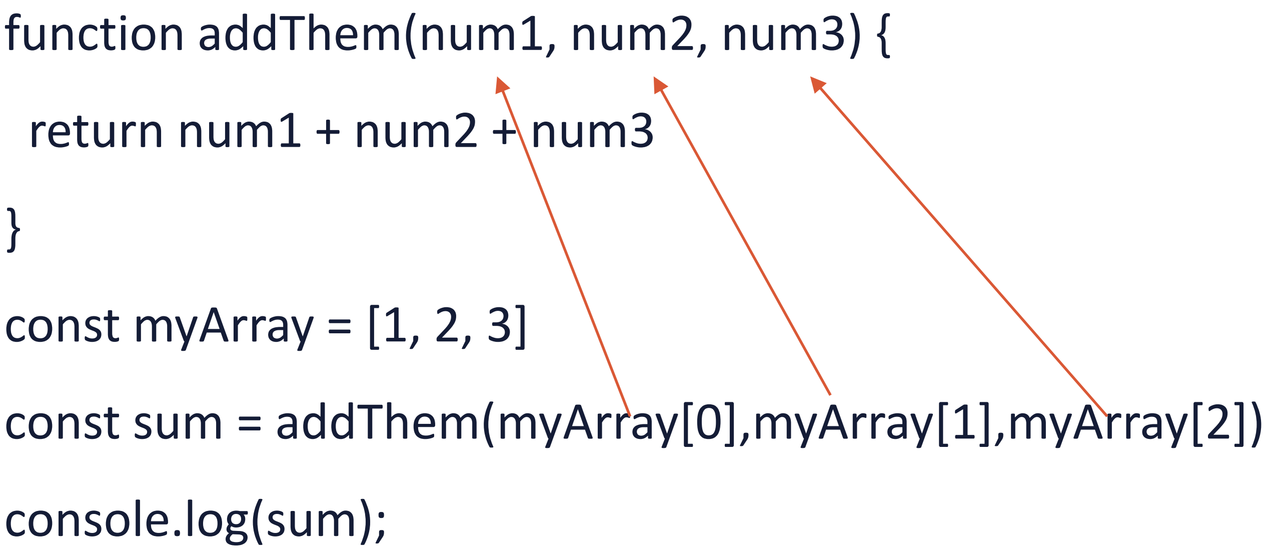
## Requires 3 parameters got 2 and one default

```
const num = [5,6];
function x(a,b,c = 0){
  let sum = a + b + c;
  return sum;
}
//
console.log(x(num[0], num[1]))
//shows 11, default integer assigned
```

# Default Parameters

# Spread Operator

## Traditional functions



```
function addThem(num1, num2, num3) {
  return num1 + num2 + num3
}
const myArray = [1, 2, 3]
const sum = addThem(myArray[0],myArray[1],myArray[2])
console.log(sum);
```

# Spread Operator

## Spread Syntax

```
function addThem(num1, num2, num3) {
  return num1 + num2 + num3
}
const myArray = [1, 2, 3]
const sum = addThem(...myArray)
console.log(sum);
```

**myArray** can hold any number of arguments

# Spread Operator

## Practical Application

```
const myArray = [4,5,6];

const allNumbers = [1, 2, 3, ...myArray];

console.log(allNumbers);

//[ 1, 2, 3, 4, 5, 6 ]
```

```
const myArray = [4,5,6];

const allNumbers = [1, 2, 3, myArray];

console.log(allNumbers);

// [ 1, 2, 3, [ 4, 5, 6 ] ]
```

The ellipses (…) are removed

# Spread Operator

## Practical Application

```
function addThem(num1, num2, num3, num4) {
  return num1 + num2 + num3 + num4;
}
const myArray = [1, 2, 3]
const sum = addThem(...myArray, 2);
console.log(sum);
//prints 8
```

Pass the additional parameter

# Spread Operator

## Practical Application

```
function addThem(num1, num2) {
  return num1 + num2;
}
const myArray = [1, 2, 3]
const sum = addThem(...myArray);
console.log(sum);
//prints 3
```

will work, we passed 3 arguments but used only 2

# Deep copying of objects

## Practical Application

```
let axle = {
        name : "axle",
        department : "Software Development"
};

let barr = axle;
barr.name = "Axle";
console.log(barr.name); //prints Axle
console.log(axle.name); //prints Axle
```

# Deep copying of objects

## Practical Application

```
let axle = {
            name : "axle",
            department : "Software Development"
};

let barr = {...axle};
barr.name = "Axle";
console.log(barr.name); //prints Axle
console.log(axle.name); //prints axle
```

Using the spread operator, we get a new object, completely de-coupled from the original **axle** object

# The Rest syntax

The rest operator puts remaining values into a JavaScript array. The spread syntax expands an collectable object into individual elements.

```
function addThem(num1, num2, ...anythingElse) {
  return anythingElse;
}
const sum = addThem(1,2,3,4);
console.log(sum);
//returns [ 3, 4 ]
```

**Rest Operator condenses arguments into an array**

# The Rest syntax

```
function addThem(num1, num2, ...anythingElse) {
  let sum = anythingElse.reduce((x,y) => x + y);
  return sum + num1 + num2;
}
const sum = addThem(1,2,3,4);
console.log(sum);
//returns 10
```

**Rest Operator condenses arguments into an array**

# Dynamic Functions in JS

```
const aFunction = new Function(arg1, arg2, { } );
```

**It is possible to use the Function object to create new functions dynamically**

# Dynamic Functions in JS

```
const func = new Function('number', 'return number + number');

let sum = func(2);

console.log(sum);
```

**It is possible to use the Function object to create new functions dynamically**

# Dynamic Functions in JS

```
const func = new Function('number', 'return number + number');

let sum = func(2);

console.log(sum);
```

```
let sum = function( x = 2 ) {
    return x+x;
};
console.log( sum( ) );
```

**It is possible to use the Function object to create new functions dynamically**

# The Rest syntax – Practical Application

```
const func = new Function('...numbers', 'return numbers’);
let sum = func(2,5,3);
console.log(sum);
```

**It is possible to use the Function object to create new functions dynamically**

# The Rest syntax – Practical Application

```
const func = new Function('...numbers', 'return
numbers.reduce( (prev, curr) => prev + curr)’);

let sum = func(2,5,3);

console.log(sum);
```

**It is possible to use the Function object to create new functions dynamically**

# The Rest syntax – Practical Application

```
const func = new Function('...numbers', 'return
numbers.reduce( (prev, curr) => prev + curr)’);
let sum = func(2,5,3);
console.log(sum); //prints 10
```

**It is possible to use the Function object to create new functions dynamically**

# Destructuring - Arrays

```
let myArray = [1,2,3,4,5,6];
let a = myArray[0];
console.log(a);
```

**From ES6 we can now destructure objects like arrays and JavaScript Objects**

# Destructuring - Arrays

```
let myArray = [1,2,3,4,5,6];
let [a] = myArray;
console.log(a);
```

**From ES6 we can now destructure objects like arrays and JavaScript Objects**

# Destructuring - Arrays

```
myArray = [1,2,3,4,5,6];
let [a,b] = myArray;
console.log(a+b);
//returns 3
```

**From ES6 we can now destructure objects like arrays and JavaScript Objects**

# Destructuring - Arrays

```
function numbers() {
            return [1,2,3,4,5,6];
}
//
let [x, y, z] = numbers();
//
console.log(x); // returns 1
console.log(y); // returns 2
console.log(z); // returns 3
```

**From ES6 we can now destructure objects like arrays and JavaScript Objects**

# Destructuring - Arrays

```
function numbers() {
        return [1,2,3,4,5,6];
}
//
let [x, …rest] = numbers();
//
console.log(x);
console.log(rest);
```

**From ES6 we can now destructure objects like arrays and JavaScript Objects**

# Destructuring - Arrays

```
myArray = [1,2,3,4,5,6];
let [a,,,b] = myArray;
console.log(a+b);
```

**From ES6 we can now destructure objects like arrays and JavaScript Objects**

# Destructuring - Arrays

```
let myArray = [1,2,3,4,5,6];
let [a,...rest] = myArray;
console.log(rest);
```

**From ES6 we can now destructure objects like arrays and JavaScript Objects**

# Destructuring – Complex Objects

```
const myObj = {
  'key01': 'value01',
  'key02': 'value02',
  'key03': 'value03'
}
let {key02} = myObj;
console.log(key02);
```

**From ES6 we can now destructure objects like arrays and JavaScript Objects**

# Destructuring – Complex Objects

```
const myObj = {
  'key01': 'value01',
  'key02': 'value02',
  'key03': 'value03'
}
let {key02, key01} = myObj;
console.log(key01, key02);
//returns value01 value02
```

**From ES6 we can now destructure objects like arrays and JavaScript Objects**

# Iterate over object values (without for)

## Object.values

```
const myObj = {
  'key01': 'value01',
  'key02': 'value02',
  'key03': 'value03’
}
const myValues = Object.values(myObj);
console.log(myValues);
```

# Iterate over object (without for)

## Object.entries

```
const myObj = {
  'key01': 'value01',
  'key02': 'value02',
  'key03': 'value03’
}
const myValues = Object.entries(myObj);
console.log(myValues);
//
```

# Iterate over object (without for)

## Object.entries

```
const myObj = {
  'key01': 'value01',
  'key02': 'value02',
  'key03': 'value03’
}
const myValues =
new Map(Object.entries(myObj));
console.log(myValues);
```

# Iterate over object (without for)

## Object.entries

```
const myObj = {
  'key01': 'value01',
  'key02': 'value02',
  'key03': 'value03'
}
const myValues = new Map(Object.entries(myObj));
const myObjLength = myValues.size;
const hasValue03 = myValues.has("key03");
console.log(`myObj is ${myObjLength} in size and it does contain value03, ${hasValue03}.`);
```

# Object.entries Practical Example

## Iterate over object to get totals

```
//total sales for 5 days
let weekSales = {
    "Monday": 100,
    "Tuesday": 300,
    "Wednesday": 700,
    "Thursday": 500,
    "Friday": 200
  };
//
function sumSales(weekSales) {
    return Object.values(weekSales);
}
//
```

# String Padding

The padLength represents the length of string **after** it is padded. If the padLength is less than the length of the string, the original string is returned and no visible differences are made.

```
let fName = "Axle";
let lName = "Barr";
console.log(fName + lName.padStart(4));
```

## Padding with characters

# String Padding

```
//total sales for 5 days
let weekSales = {
    "Monday": 100,
    "Tuesday": 300,
    "Wednesday": 700,
    "Thursday": 500,
    "Friday": 200
  };
//
function sumSales(weekSales) {
    return Object.values(weekSales).reduce((a, b) => a + b, 0);
}
let weeklySales = sumSales(weekSales).toString().padStart(8,'0');
console.log(weeklySales);
//prints: 00001800
```

## Padding with Zeros

# String Padding

```
//total sales for 5 days
let weekSales = {
    "Monday": 100,
    "Tuesday": 300,
    "Wednesday": 700,
    "Thursday": 500,
    "Friday": 200
  };
//
function sumSales(weekSales) {
    return Object.values(weekSales).reduce((a, b) => a + b, 0);
}
let weeklySales = sumSales(weekSales).toString().padEnd(6,'0');
console.log(weeklySales);
//prints: 180000
```

## Trailing with Zeros

# String Padding

```
//total sales for 5 days
let weekSales = {
    "Monday": 100,
    "Tuesday": 300,
    "Wednesday": 700,
    "Thursday": 500,
    "Friday": 200
  };
//
for (const [key, value] of Object.entries(weekSales)) {
  console.log(key + value);
}
```

## Trailing with Zeros

# String Padding

```
//total sales for 5 days
let weekSales = {
    "Monday": 100,
    "Tuesday": 300,
    "Wednesday": 700,
    "Thursday": 500,
    "Friday": 200
  };
//
for (const [key, value] of Object.entries(weekSales)) {
  console.log(key.padEnd(10) + ":" + value);
}
```

## Trailing with Zeros

# String Padding

```
//total sales for 5 days
let weekSales = {
    "Monday": 100,
    "Tuesday": 300,
    "Wednesday": 700,
    "Thursday": 500,
    "Friday": 200
  };
//
for (const [key, value] of Object.entries(weekSales)) {
  console.log(key.padEnd(10) + ":" + value.toString().padStart(5));
}
```

## Trailing with Zeros

# String Padding

```
let weekSales = {
  "Monday": 1009,
  "Tuesday": 360,
  "Wednesday": 7800,
  "Thursday": 50,
  "Friday": 2077
};
//
for (const [key, value] of Object.entries(weekSales)) {
  console.log(key.padEnd(10) + ":" + value.toString().padStart(5)+".00");
}
//
function sumSales(weekSales) {
  return Object.values(weekSales).reduce((a, b) => a + b, 0);
}
//
console.log("--------".padStart(19));
let weeklySales = (sumSales(weekSales).toString()+".00").padStart(19);
console.log(weeklySales);
```

## Trailing with Zeros

# Symbols

A Symbol is another data type in JS

A Symbol is a primitive type part of the ES2015 specifications.

It is unique and cannot be mutated once created. However it behaves like an object.

Each new symbol created is guaranteed to return a unique Symbol.

Mainly used in libraries - code safety

**Symbol is a JS Data Type**

# Symbols

```
const employee = {
  empId: 'Emp1',
  fName: 'Axle',
  lName: 'Barr'
};
//
console.log(employee);
employee.empId = "Emp2";
console.log(employee);
```

```
{ empId: 'Emp2', fName: 'Axle', lName: 'Barr' }
```

**Symbol is a JS Data Type**

# Symbols

```
const empId = Symbol('empId');
const employee = {
  empId: 'Emp1',
  fName: 'Axle',
  lName: 'Barr'
};
//
console.log(employee);
employee.empId = "Emp2";
console.log(employee);
```

```
{ empId: 'Emp2', fName: 'Axle', lName: 'Barr' }
```

## Symbol is a JS Data Type

# Symbols

```
const empId = Symbol('empId');
const employee = {
  [empId]: 'Emp1',
  fName: 'Axle',
  lName: 'Barr'
};
//
console.log(employee);
employee.empId = "Emp2";
console.log(employee);
```

```
{ fName: 'Axle', lName: 'Barr', [Symbol(empId)]: 'Emp1' }
```

**Symbol is a JS Data Type**

# Symbols

```
const empId = Symbol('empId');
const employee = {
  [empId]: 'Emp1',
  fName: 'Axle',
  lName: 'Barr'
};
//
console.log(employee);
employee.empId = "Emp2";
console.log(employee);
```

```
{ fName: 'Axle', lName: 'Barr', [Symbol(empId)]: 'Emp1’ }
{  fName: 'Axle',  lName: 'Barr',  empId: 'Emp2',
[Symbol(empId)]: 'Emp1’ }
```

## Symbol is a JS Data Type

## Classes encapsulate an object

# Classes in JavaScript

```
class Employee {
  empId = 0;
  fName = "";
  lName = "";
};
```

## Classes encapsulate an object

# Classes in JavaScript

```
class Employee {
  empId = 0;
  fName = "";
  lName = "";
};
```

## Classes encapsulate an object

# Classes in JavaScript

```
class Employee {
  empId = 0;
  fName = "";
  lName = "";
  constructor(empId, fName, lName){
    this.empId = empId;
    this.fName = fName;
    this.lName = lName;
  }
};
```

**Classes encapsulate an object**

# Classes in JavaScript

```
class Employee {
constructor(empId, fName, lName){
   this.empId = empId;
   this.fName = fName;
   this.lName = lName;
 }
};
```

**Classes encapsulate an object**

# Classes in JavaScript

```
class Employee {
constructor(empId, fName, lName){
    this.empId = empId;
    this.fName = fName;
    this.lName = lName;
  }
};
const emp1 = new Employee(100, "Axle", "Barr");
```

## Classes encapsulate an object

# Classes in JavaScript

```
class Employee {
constructor(empId, fName, lName){
    this.empId = empId;
    this.fName = fName;
    this.lName = lName;
  }
   aboutEmp(){
    return this.fName + " " + this.lName + " is employee #" + this.empId;
  }
};
const emp1 = new Employee(100, "Axle", "Barr");
console.log( emp1.aboutEmp());
```