JavaScript Part 1
The Language

Dr. Samuel Cho, Ph.D.

NKU ASE/CS

JavaScript - Quick Introduction

Dynamic Language

- JavaScript is a dynamic programming language just like Python.
- Java is a static type language.
- We don't specify the type when we program in dynamic programming languages.

- JavaScript uses 'var' to declare a variable and '=' to assign a value to the variable.
- It's similar to Java, but not the same.
- In Python, an OOP dynamic language, we don't need the 'var.'
- In JavaScript, ';' is optional, just like in Python.

Code #

```
1 int x = 10; // Java
2 x = 10 # Python
3 var x = 10 # JavaScript
```

- We can change a value and its type 'dynamically' using dynamic programming languages.
- This is convenient, but dynamic language compilers cannot detect the errors to raise exceptions that are hard to find.

Code #

```
1 x = 'Hello'; // Java Compiler knows this is an error
2 int y = x/2; // This is not even compiled in Java
3
4 x = 'Hello' # x is dynamically updated its type in Python, and it
    ↪ 's OK.
5 x/2 # ??? Exception in runtime, not in compile time!
```

Weakly Typed Language

- JavaScript is a weakly typed language.
- Java, Python, Ruby, and practically all languages are Strongly typed languagess.
- It means that a value can be interpreted for operation.
- Even the string '1' can be interpreted and converted into 1.

Code #

```
1 1 == '1' // True!
```

Primitive Types

- JavaScript has primitive types: string, number, boolean, null, undefined.
- The first type is a string.
- There is no character type in JavaScript, only string types; so we can use ' and " for strings.

Code #

```
1 var x = "abcd" // this is a string.
2 var x = 'a' // this is also a string, there is no char
    ↪ in JavaScript
```

Primitive Types

number

- JavaScript has double-type values to represent numbers.
- For example, int type 1 and double 1.0 are the same number in JavaScript, as all numbers (numeric values) in JavaScript are represented as 8-byte floating-point numbers.

- Even though we can use int values, there is no distinction between the int and double type values in JavaScript.

Code #

```
1 1 == 1.0 // True!
```

Primitive Types

Boolean

- Boolean represents either 'true' or 'false.'

Code #

```
1 var isTrue = true
2 var isFalse = false
3 2 + 3 == 5 // returns true
```

Null and Undefined

- null is a value.
- undefined is a variable is not referencing anything.

Code #

```
1 var a = null // a has a value null
2 var b // b is undefined
```

JavaScript Object

- Just like Java has primitive values and objects, JavaScript has primitives (number, string, boolean, undefined, null) and objects.
- JavaScript object notation (JSON) is represented in string format.
- When it is parsed as JavaScript code, it is called a JavaScript Object.

JSON Representation

- JSON representation is a string format to express dictionary-like key/value pairs.

Code #

```
{
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
}
```

JSON Representation

There are four JSON Syntax Rules

- Data is in name/value pairs.
- Data is separated by commas.
- Curly braces hold objects.
- Square brackets hold arrays

JSON Representation

JSON Data - A Name and a Value

- JSON data is written as name/value pairs.
- A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value.

Code #

```
1 "firstName":"John"
```

JSON Representation

JSON Objects

- JSON objects are written inside curly braces.
- Objects can contain multiple JSON data (name/value pairs).

Code #

```
1 {"firstName":"John", "lastName":"Doe"}
```

JSON Representation

JSON Arrays

- JSON arrays are written inside square brackets.
- An array can contain objects.

Code #

```
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
```

JSON Representation

Recursive Structure

- At the highest level, JSON representation is objects (curly braces) or arrays (square brackets).
- Considering an array is a specialized type of an array, JSON represents an object.

Code #

```
1 {"a":10, "b":{"c":30, "d":false}}
2 [1, 2, "c":{"x":10, "y":[1,2,3]}]
```

- An object has a series of data separated by a comma.
- A data has a value of any type, including an object.
- So, we can recursively express any complicated data structure using JSON.

Code #

```
1 {"a":10, "b":{"c":30, "d":false}}
```

JSON to JSON object

- The JSON.parse() function transforms any valid JSON string into a JSON object.

Code #

```
1 var text = '{ "employees" : [' +
2 '{ "firstName":"John", "lastName":"Doe" },' +
3 '{ "firstName":"Anna," "lastName":"Smith" },' +
4 '{ "firstName":"Peter," "lastName":"Jones" } ]}';
5 var obj = JSON.parse(text);
6 console.log(obj["employees"][0]["firstName"]) // "John"
```

JSON object to JSON

- Likewise, we can transform any valid JSON object into a JSON string using the JSON.stringify() function.

Code #

```
1 const jsonObject = { key1: 'value1', key2: 42, key3:
    ↪ true };
2
3 // Convert JavaScript object to JSON string
4 const jsonString = JSON.stringify(jsonObject);
```

JavaScript Statements

Selection

- The expression result (true or false) is evaluated to decide which block to execute.
- We can remove the braces when the block has only one statement.

Code #

```
1 if (x == 0) {console.log('x is 0')}
2 else console.log('x is not 0')
```

JavaScript Statements

Loop

- JavaScript supports the for loop for repetition.
- It is a good habit to use 'const' in the loop.

Code #

```
1 var x = [1,2,3,4,5]
2 for (i in x) { // OK
3   console.log(x[i])
4 }
5 for (const i in x) { // Better
6   console.log(x[i])
7 }
```

HW2

- Use HW2 to learn JavaScript syntax and semantics.
- For the rest of this chapter, we focus on the hard-to-understand but important aspects of JavaScript.

Why JavaScript was Invented?

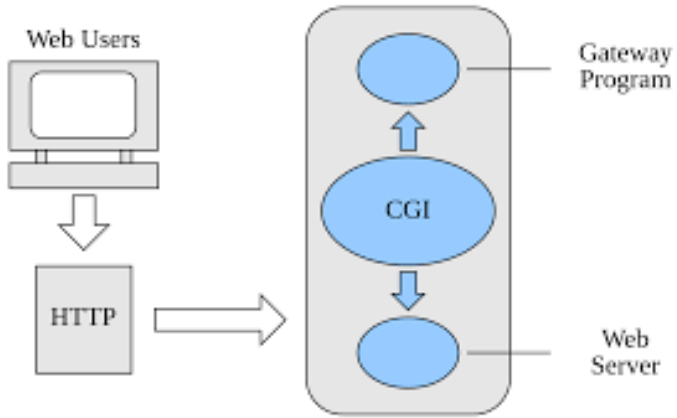
Web 1.0 - Static HTML

- HTML was invented to display Document Object Model (DOM) information on web browsers.
- When web servers get a request from clients (mostly web browsers), they send static HTML files as a response (Web 1.0).

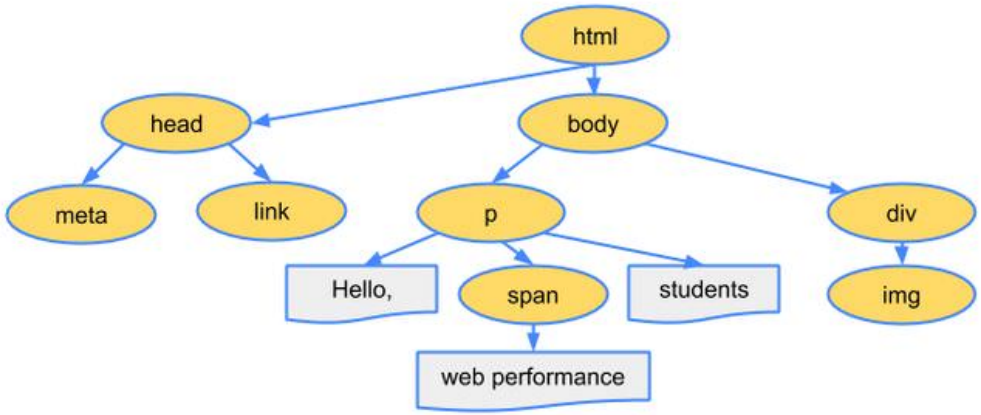
- CSS (Cascading Style Sheet) was invented to render HTML elements on a web browser more flexibly.

Web 2.0 - CGI

- With the invention of CGI (Common Gateway Interface) (Web 2.0), servers can generate HTML DOM dynamically from the request.



- JavaScript was invented to manipulate HTML structure (DOM) on the clients' side.



- With the invention of JavaScript, we can have a dynamic web page (DOM) both from a server-side (CGI) and a client-side (JavaScript).

- This JavaScript code (lines 3 – 6) accesses an HTML element (line 1) with "one" id, for example, `<h1 id="one">...</h1>`, and displays the content using the alert() method.

Code #

```
1 <h1 id="one">Hello</h1>
2
3 <script type="text/javascript">
4   var text = document.getElementById("one").innerHTML;
5   alert(text); <!-- displays Hello -->
6 </script>
```

- JavaScript was invented in 10 days by Brendan Eich for the Netscape web browser.
- He had to make the language as simple as possible.

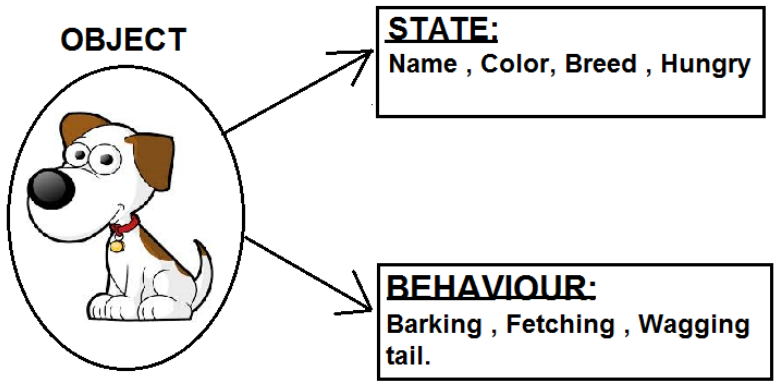
- So, JavaScript models LISP/Scheme (the first functional programming language).
- That is why JavaScript uses functional programming language features.

- For object manipulation, JavaScript copied the Prototype language features from Self.
- JavaScript has no close relationship with Java in language features.

Functional Programming

OOP

- In the OOP paradigm, we have two different entities in programming: fields (value) and methods (code).
- Fields are variables, and methods are functions in a class.



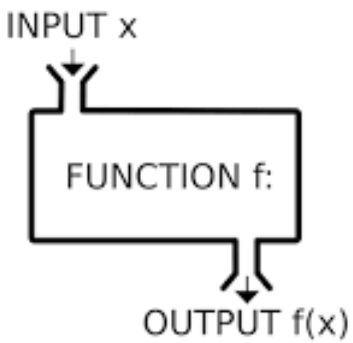
- We treat data and code differently in OOP, as in this example Dog class.

Code #

```
1 class Dog {
2   String name = "Sparky"; // data (fields)
3       ...
4
5   void barking() { // code (methods)
6       ...
7   }
8   ...
9 }
```

FP

- In the FP paradigm, everything is a function (code).
- There is no difference between the function and value.
- In other words, we treat the function (code) as the same as the value (data).



- We define a variable x that references a data (value) 3 (line 1).
- We define a function add (line 2).
- In line 3, the function is a data (value) referenced by a variable add.

Code #

```
1 var x = 3 // data
2 function add(x, y) { return (x + y) } // function
3 // function is a data
4 var add = function (x, y) { return (x + y) }
```

Lambda Expression

- Lambda expression is a nameless function with only inputs and its body.
- In JavaScript, we also call lambda expression as arrow function.

```
() => {}
```

- In JavaScript, the body can be a block enclosed with braces ({}) with single or multiple **statements** and uses the 'return' keyword to return a value (line 1).
- When the body returns a single expression, we can write only the expression (line 2).

Code #

```
1 (x, y) => {return (x + y)}
2 (x, y) => (x + y)
```

Higher-Order Functions

- As a function is a value, a function can be an argument to other functions.
- A function can be a return value.
- A function can be used anywhere values are used.

Callback and Higher-Order Functions

- The function we pass as an argument to another function is called the callback function.
- A function (a) that receives another function as an argument; (b) that returns a new function; (b) or both is called a Higher-order function.

- In this example, we can give the add or sub (callback) functions as an argument to the `higher_order_function`.
- This is possible because the function (code) is the same as the value (data) in FP.

Code #

```
1 var add = (x, y) => (x + y) // callback function
2 var sub = (x, y) => (x - y)
3 var higher_order_function = (f, x, y) => f(x, y)
4 higher_order_function(add, 10, 20) // returns 10 + 20
5 higher_order_function(sub, 10, 20) // returns 10 - 20
```

JavaScript and Lisp/Scheme

- JavaScript uses FP approaches extensively as it is based on Lisp/Scheme.
- It also adds some OOP flavor to it.
- So, we can understand JavaScript using FP concepts with some twists.

Three Ways to Define Function

- The first way is to use the 'function' keyword to define a function (line 1).
- The second way is to define a function object and use the add variable to reference the object (line 2).

Code #

```
1 function add(x,y) {return x + y; }
2 var add = function (x, y){ return x + y; }
```

- The third way (FP way) is to use lambda expression and use the add variable to reference the lambda expression.
- Now, JavaScript (line 1) becomes the Lisp/Scheme code with minor syntax changes (line 3).

Code #

```
1 var add = (x, y) => (x + y) // JavaScript code
2
3 (define (add) (lambda (x y) (+ x y))) ; Scheme code
```

Let, Var, and Const

Let, Var, and Const

- JavaScript has two ways to make a variable: let and var.
- The difference between them is that (a) the variables defined with 'var' can be redefined, but (b) 'let' variables cannot.
- The variables defined with 'const' cannot be modified.

Var and Let Example

- Line 5 is an error as 'a' is defined with the 'let' keyword.
- We can redefine 'b' anywhere (line 7).
- We cannot change the const values (line 8).

Code #

```
1 let a = 1;
2 var b = 2;
3 const c = 3;
4 if (true) { // to make local variables
5   //var a = 4; // Error as a is let defined
6   a = 3; // OK
7   let b = 3;  // OK as b is var defined
8   // c = 4; //Error
9 }
```

Another Issue with Var

- The var defined variable is not block scoped so it can be used anywhere.
- The let defined variable is block scoped just like Java's local variables.

Code #

```
1 function run() {
2   {
3     var moo = "Mooo" // can be used anywhere
4     let baz = "Bazz" // can be used only in this block
5     console.log(moo, baz); // Mooo Bazz
6   }
7   console.log(moo); // Mooo
8   console.log(baz); // Reference Error
9 }
```

Just Use Let

- The 'var' keyword was introduced when JavaScript was invented, but it caused many issues.
- So, the new 'let' keyword was introduced later to address the issue.
- JavaScript has two different ways to do the same thing to add complexity in programming.

- So, to avoid this confusion, it is recommended to use let variables.
- Don't use var variables unless you know what you are doing.

Code #

```
1 var x = 10
2 // This is possible, but why?
3 var x = 20
4 let z = 20
5 //let z = 40 // Error!
6 {let z = 40} // This is OK
7 z = 30 // this is OK
```

JSON, AJAX, and OOP

JSON

- JSON (JavaScript Object Notation) is a representation to express the idea of a dictionary in JavaScript.
- JSON was invented to share information between a web server and a web browser using JavaScript.
- As JavaScript becomes popular, JSON format is used outside JavaScript to make it a language-independent data format.

AJAX

- We learned that JavaScript was invented to update the DOM.
- JavaScript in HTML can access the web server to retrieve JSON information and update its DOM.
- This is how Google Map updates its screen through DOM update without retrieving HTML files from the web server.

- For this, JavaScript communicates with servers with AJAX (Asynchronous JavaScript and XML)
- JSON objects represent the shared information.
- However, using AJAX with JavaScript is unnecessarily complicated.

- Instead, we often use libraries such as jQuery (line 2).
- This is the code to request a web server with the POST function using JSON object.
- Notice the JSON object in the code (lines 2 – 6).

Code #

```
1 <script>
2   $.ajax({
3     method : 'POST',
4     url : '/add',
5     data : 'study'
6   })
7 </script>
```

JavaScript as Prototype Language

- JavaScript is a Prototype language.
- In the prototype languages, they do not instantiate a new object; instead, they copy an existing object and modify the copied object.

JSON and FP to Mimic OOP

- We learned that a function is a value in JavaScript.
- So, we can store the function body as a value in JSON.
- This is how we implement (mimic) objects in JavaScript.

The 'this' Keyword in an Object

- When a function has 'this' in it, the 'this' means the object that contains the function.
- So, when we access any data in an object, we should prepend the 'this' keyword similar to Java.

- We use class and instantiation for OOP programming in Java (lines 1 – 5).
- We use JSON objects for OOP programming in JavaScript (6 – 10).

Code #

```
1 class Hello {
2   int x = 10;
3   void print() { return x;}
4 }
5 Hello h = new Hello(); h.print()
6 h = {
7     x: 10,
8     print: function() { return this.x; }
9 }
10 h.print()
```

This Keyword in => Function

- We can use the arrow function (lambda expression) in an object, but the 'this' in a lambda expression is not defined
- This example shows the difference.

Code #

```
1 var obj = {
2     data: 'Sam',
3     getData: function() {return this.data}, // returns 'Sam'
4     getData2: () => {return this.data;} // returns 'undefined'
5 }
```

Apply and Call Higher-Order Functions

- We understand FP.
- We understand JavaScript functions.
- We understand JSON objects.
- So, we can understand how the apply and call higher-order functions are used in JavaScript.

Apply

- The higher-order apply function applies a function to arguments.
- This is the FP way of function call.
- It is one of the most frequently used in FP.

- As the Lisp/Scheme example shows, when we use the apply function, it applies the function hi, (+ x y z), to the arguments '(1 2 3).
- x is applied to 1, y is to 2, and z is 3).
- The result is (+ 1 2 3) == 6.

Code #

```
1 (define (hi x y z) (+ x y z))
2 (apply hi '(1 2 3)) ; it applies hi function to the
    ↪ arguments (1 2 3)
```

- JavaScript functions can apply itself using 'object.function.apply(arguments).'
- In this example, we apply the function hello() to an object person2.
- The values in the object are accessed using the 'this' keyword.

Code #

```
1 var person = {
2   hello : function(){console.log(this.name)}
3 }
4 var person2 = { name : 'Sam' }
5 person.hello.apply(person2)
```

- If necessary, we can give additional arguments to the function hello.
- In this example, arguments '[1,2,3]' is given to the hello() function.
- Notice that 'this.name' will access the value in person2 Object.

Code #

```
1 var person = {
2   hello : function(x, y, z){
3     console.log(this.name + ' Hello ${x}-${y}-${z}')
4   }
5 }
6 var person2 = { name : 'Sam' }
7 person.hello.apply(person2, [1,2,3]);
```

- Remember that the additional arguments should be in a list, as this idea is borrowed from Lisp/Scheme.
- JavaScript code can combine FP and OOP features to make reading and understanding the code hard.

Call

- The funcall() function in Lisp/Scheme is similar to the apply() function.
- The only difference is that we give each argument to the function (line 1).

Code #

```
1 (funcall hi 1 2 3) ;; Lisp example
```

- JavaScript follows the same convention; when we use call() instead of apply(), we should give arguments one by one.

Code #

```
1 person.hello.call(person2, 1,2,3); // JavaScript
```

The 'this' in JavaScript

- Compared to the **this** in Java that references this object, the **this** in Javascript can refer to different objects depending on where it is invoked.
- This is confusing, and it can be a source of serious bugs. So, we should be careful when we use the **this** in JavaScript.

- This code shows that the **this** returns different objects depending on where it is used.

Code #

```
1 'use strict'
2 console.log(this) // {Window} <- when invoked in a global context
3 function a() {
4   console.log(this)
5 } // undefined <- when invoked in a function
6 var o = {
7   f: function() {return this}
8 }
9 console.log(o.f()) // the JSON object o
```

Strings

JavaScript String

- Java has a char type (primitive value) and a String (reference type object), and we use ’.’ to represent a character and ’’...’’ for a Java String (CSC 260).
- JavaScript has only strings, and we use one notation ’...’, and there are no character types in JavaScript.

- We can concatenate strings and variables using the '+' operator like Java (line 2).
- We also can interpolate strings (using variables in a string) with ‘ . . . ‘ (reverse quotation) and ${...} as in line 3.

Code #

```
1 var cruel = 'cruel';
2 var line = 'hello ' + cruel + ' world';
3 var line2 = `hello ${cruel} world`;
```

Usage of Interpolated Strings: Tagged List

- When the interpolated string is used as an argument to a function, the function can analyze the input to extract values.
- We call this feature 'tagged list.'

- In this example, the analyzer function parses the string argument to retrieve var1 and var2.

Code #

```
1 function analyzer(string, var1, var2) {
2     if(var1 == 0){
3       console.log(`Pants are sold out, Socks:` + var2);
4     }
5 }
6 var pants = 0; var socks = 100
7 analyzer`pants${pants} socks${socks}`
```

Prototype Language

Prototype Function

- JavaScript is a prototype-based language similar to but different from OOP languages.
- This means that we can copy a prototype object to make another object.
- JavaScript function is an object, so we can use it as a prototype.

- In this example, we have a prototype function that contains the member variable 'name.'
- Notice that in the prototype members, 'this' should be prepended.

Code #

```
1 function Machine(){
2   this.name = 'Sam'; // Don't forget this
3 }
```

- We need to follow the JavaScript coding convention that the prototype function name should start with a capital letter.
- We can use the 'new' operator to copy an object from a prototype function.

Code #

```
1 function Machine(){ // prototype function
2   this.name = 'Sam'; // Don't forget 'this'
3 }
4
5 var person1 = new Machine(); // {name: 'Sam'}
6 var person2 = new Machine(); // {name: 'Sam'}
```

- We can add new functions in the prototype function.
- We can call the functions in the objects (lines 9 - 10).

Code #

```
1  function Machine2(){
2    this.name = 'Sam';
3    this.hello = function() {
4      console.log('Hello ' + this.name)
5    }
6  }
7  var person1 = new Machine2();
8  var person2 = new Machine2();
9  person1.hello() // 'Hello Sam'
10 person2.hello() // 'Hello Sam'
```

Constructor

- When we want to change the variable name, we can give **an argument** to the prototype function.
- We call this function constructor because it does the same thing as Java Constructor (CSC 260).

- This is an example of a constructor with an argument.

Code #

```
1 function Machine3(name){ // constructor with an
    ↪ argument
2   this.name = name;
3   this.hello = function() {
4     console.log('Hello ' + this.name)
5   }
6 }
7 var person1 = new Machine3('Sam');
8 var person2 = new Machine3('John');
9 person1.hello() // 'Hello Sam'
10 person2.hello() // 'Hello John'
```

Inheritance Using Prototype

- Any (prototype) objects have the 'prototype' property.
- When we need to extend prototype functions, we can add new features to the prototype using the 'prototype' property.

- This code shows how we can add the 'id' variable to the Machine3 function using the prototype property.
- The person1 object's prototype function is Machine 3.
- In this example, the copied object person1 does not have the prototype's id.

Code #

```
1 Machine3.prototype.id = 'chos5';
2 var person1 = new Machine3('Sam');
3 console.log(person1) // Machine3 { name: 'Sam', hello:
    ↪ }
```

- However, using the `__proto__` property, the person1 can access its prototype values.
- In line 3, the person1 object searches for its values; if 'id' is not one of them, it searches the prototype values.

Code #

```
1 Machine3.prototype.id = 'chos5';
2 console.log(person1.__proto__) // { id: 'chos5' }
3 console.log(person1.id) // chos5
```

Example of Prototype - toString()

- Just like Python, we can use [] to make an array, and it is the simplified syntax of 'new Array(...).'
- We can use the toString() function in the arr object.

Code #

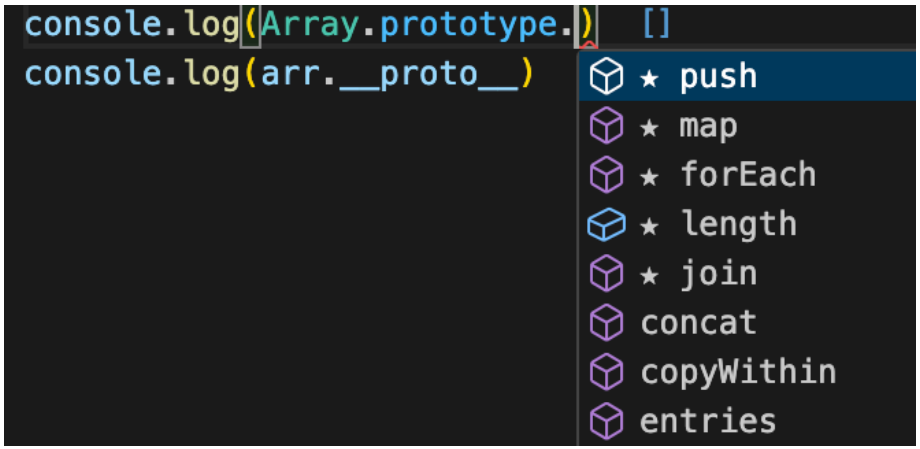
```
1 var arr = [1,2,3]
2 var arr = new Array(1,2,3)
3 arr.toString()
```

- The Array function stores many functions it supports in its prototype variable.
- This code shows the implementation of the Array prototype.

Code #

```
1 function Array { ... }
2 Array.prototype.toArray = function ...
```

- Using Visual Studio Code, we can display the prototype values.
- This picture shows the available functions of the Array function; toArray is one of them.



- We can use the 'Array.prototype' to check the available functions; also, we can use 'arr.__proto__' to get the same information.
- We need to use Chrome's Developer feature to get the full list, as Node.js shows no information.

Code #

```
1 console.log(Array.prototype); // to get parent's
    ↪ prototype (Object(0))
2 console.log(arr.__proto__); // Object(0)
```

Dynamic Object Extension

- JavaScript can dynamically extend objects.
- In this example, the p object can be the prototype of an empty object c.
- As a result, c can access all the object elements of p.

Code #

```
1 var p = {name: 'Sam'}; var c = {}
2 c.__proto__ = p
3 console.log(c) // {}
4 console.log(c.name) // 'Sam'
```

ES5 Features

- JavaScript ES5 is the first major revision of JavaScript.
- JavaScript ES5 adds a new function Object.create().

- We can use Object.create() to make an object, and the copied object's prototype has the p using the dynamic object extension (line 2).
- So, when we print c, we get an empty object, but we can access the p.id variable through the prototype.

Code #

```
1 var p = {name: 'Sam', id:'chos5'}
2 var c = Object.create(p)
3 console.log(c) // {}
4 console.log(c.id) // c can access id
```

- However, when we add a new variable id in the c object (line 1), the id becomes a member of the object c dynamically (line 2).
- The id in its prototype is overridden, as the c.id will be accessed, not the p.id.
- We need to use `__proto__.id` to access the hidden prototype values.

Code #

```
1 c.id = 'chos6' // we create a new id
2 console.log(c) // {id: 'chos6'} prototype id is hidden
3 console.log(c.id) // chos6 is printed
4 console.log(c.__proto__.id) // chos5 is printed
```

OOP Langauge

ES6 Class Feature

- JavaScript ES6 actively adopts the OOP ideas.
- JavaScript introduces the 'class' keyword to support the OOP idea.
- However, behind the scenes, it is syntactic sugar, meaning that it mimics OOP programming methodology, not that ES6 is the true OOP language.

- This example shows how we can use 'class' to incorporate the constructor function.

Code #

```
1  // ES5 or before
2  function Machine3(name){
3    this.name = name;
4    this.hello = function() {
5      return('Hello ' + this.name)
6    }
7  }
8  // ES6
9  class Machine4 {
10   constructor(name) {
11     this.name = name
12     this.hello = function() {
13       return('Hello ' + this.name)
14     }
15   }
16 }
```

When we need to add more prototype functions, we have two approaches.

- The first approach is to use a prototype to add a new function.

Code #

```
1 // Adding a function to prototype - method 1
2 Machine4.prototype.hello2 = function() {
3   return('Hello2 ' + this.name)
4 }
```

- The second approach is to make the prototype function in the class (lines 9 - 11).

Code #

```
1  // Adding a function to prototype - method 2
2  class Machine5 {
3    constructor(name) {
4      this.name = name
5      this.hello = function() { // added to this
6        return('Hello ' + this.name)
7      }
8    }
9    hello2() { // added to prototype
10     return('Hello2 ' + this.name)
11   }
12 }
```

- We can make an object (line 1).
- In this example, person1 is an object from the class NewMachine5, so it has all the functions and variables in the constructor.
- However, even though it does not have hello2() function, it can access and use it (line 3).

Code #

```
1 var person1 = new NewMachine5('Sam','GH565');
2 console.log(person1); // Machine5 { name: 'Sam', hello:
    ↪ ...}
3 console.log(person1.hello2()) // Hello2 Sam
```

Properties

- We learned that a function (as a value) can be an element of the JSON object (lines 1 – 4).
- We can write the function as in lines 5 – 8.

Code #

```
1 h = {
2   x: 10,
3   print: function() { return this.x; }
4 }
5 h2 = {
6   x: 10,
7   print() { return this.x; }
8 }
```

- In this example, the setAge() setter method is to access and update the age in the person JSON object.

Code #

```
1 var person = {
2   name: 'Sam', age: 10,
3   setAge(age) { this.age = parseInt(age); }
4 }
5 person.setAge('15')
6 console.log(person)
```

- JavaScript supports syntactic sugar; when we add the set/get modifier, we can make function properties.

Code #

```
1  var person = {
2    name: 'Sam',
3    age: 10,
4    // set makes setAge function a setAge property
5    set setAge(age) {
6      this.age = parseInt(age);
7    },
8    get nextAge() {
9      return this.age + 1
10   }
11 }
```

- Properties are functions that can be used as if they are variables.

Code #

```
1 person.setAge = 20 // setAge becomes a property
2 console.log(person.nextAge)
3 console.log(person)
```

Interesting features

Spread Operator

- We can use an array to aggregate multiple values (lines 1 – 2).

Code #

```
1 var array = ['hello', 'world'];
2 console.log(array);    // ['hello', 'world']
```

- When we use an array with '...' prepended, we call the '...' a spread operator.
- The spread operator spreads out each element of an array as function arguments.
- (For Python Programmers) This is the same as the Python's explode (*) operator.

Code #

```
1 console.log(...array); // hello world
```

Spread Operator

Usage 1

- Let's say we have a function hello() that requires two arguments.

Code #

```
1 var array = ['hello', 'world'];
2
3 function hello(x, y) {
4   console.log(x);
5   console.log(y);
6 }
```

- When we don't use the spread operator, we should extract the two elements in an array as in line 1.
- But we don't have to do it with the spread operator that does the work for us.

Code #

```
1 hello(array[0], array[1]); // ugly
2 hello(...array); // better
```

Spread Operator

Usage 2

- We can use the spread operator to concatenate arrays into one.

Code #

```
1 var a = [1,2,3];
2 var b = [4,5];
3 var c = [...a, ...b]; // [1,2,3,4,5]
```

Spread Operator

Object Copy

- We can use the spread operator for an object.
- This example shows how we can copy all the elements in an object o1 into o2 with a new element added.

Code #

```
1 var o1 = { a : 1, b : 2 };
2 var o2 = { c : 3, ...o1 };
3 console.log(o2); // { c: 3, a: 1, b: 2 }
```

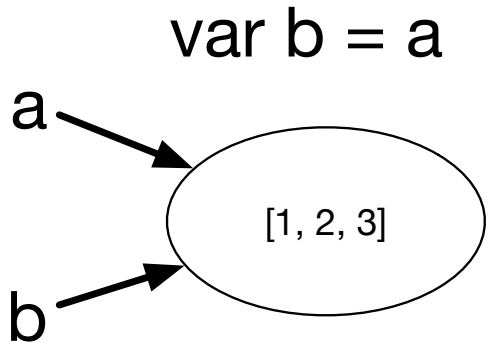
Spread Operator

Deep Copy and Shallow Copy

- When we copy arrays, we sometimes make the mistake of copying only a reference; we call it a shallow copy.

Code #

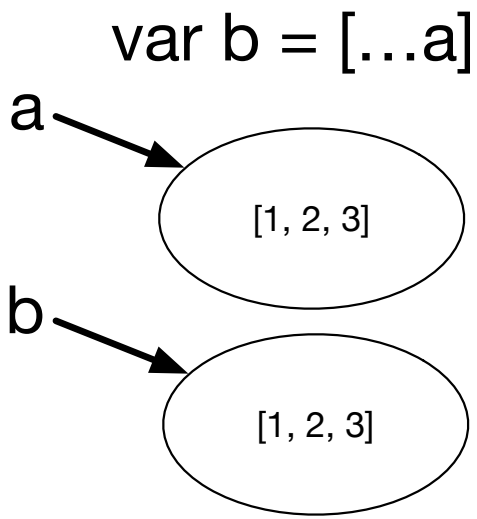
```
1 var a = [1,2,3];
2 var b = a;
3 b[0] = 1000;
4 console.log(a[0] == b[0]); // true
```



- To avoid this, we should copy all the elements; we call it a deep copy.

Code #

```
1 var a = [1,2,3]
2 var b = [...a]
3 b[0] = 2000;
4 console.log(b[0] != a[0]) // true
```



Spread Operator

Rest Parameter

- The spread operator can be used as a function parameter when we don't know the number of arguments given.
- Javascript aggregates all the inputs in an array, and the spread operator breaks into elements.

Code #

```
1 function f(...params){ console.log(params); }
2 f(1,2,3,4,5,6,7); // [1,2,3,4,5,6,7]
```

- When we select only some of the arguments, we can use the normal parameter and the spread parameters to select only the arguments into an array.

Code #

```
1 function f2(a,b, ...params){
2   console.log(params);
3 }
4 f2(1,2,3,4,5,6,7); // [3,4,5,6,7]
```

Sameness Check

- You may remember that in Java, we use '==' operator for primitive type values, and equals() method to compare objects (CSC 260).

Code #

```
1 System.out.println(1 == 2); // compare primitive values
2 System.out.println("Hello".equals("Hello")); //
    ↪ compare objects
```

- JavaScript is a weakly typed language.
- JavaScript will interpret the type of the value whenever necessary, depending on the situation.
- Even the string '1' can be interpreted and converted into 1.

- JavaScript uses '==' operator when we compare two values ignoring data type.
- So, as long as values represent the same number or print the same thing, such as 1, 1.0, or even '1'. They are the same with the '==' operator.

Code #

```
1 console.log(1 == '1') // true
2 console.log(1 == 1.0) // true
```

Sameness Check

The === Operator

- So, when we compare two values and when we need to check the data type also, we should use the '===' or '!==' operator.
- In this example, line 1 returns false.

Code #

```
1 console.log(1 !== '1') // true
```

- Notice that this number comparison returns true because there is no integer type in JavaScript; all the numbers are floating-point numbers.

Code #

```
1 console.log(1 === 1.0) // true
2 console.log(1 !== 1.0) // false
```

- We should also remember that the number 1 and true are the same with the '==' operator but not with the '===' operator.
- Also, 0 and false are the same with '==' operator, but false with '===' operator.

Code #

```
1 console.log(1 == true) // true
2 console.log(1 === true) // false
3
4 console.log(0 == false) // true
5 console.log(0 === false) // false
```

Sameness Check

Object comparison

- In this example, the two different objects comparison with '==' and '===' operators return false.
- It is because a and b store each object's reference (address).

Code #

```
1 var a = { name : 'Sam' }; var b = {...a};
2
3 console.log(a == b)  // false
4 console.log(a === b) // false
```

- Likewise, to check shallow equality, we can use '==' or '===' operators

Code #

```
1 var a = { name : 'Sam' };
2 var b = a
3
4 console.log(a == b)  // true
5 console.log(a === b) // true
```

Sameness Check

Implementing deepEqual()

- Remember that JavaScript's object is a JSON object.
- To check object equality, we need to compare if the two (JSON) objects have the same values recursively.
- This is similar to Java's equals() method that checks if two objects (not primitives) have same values or not.

- For implementing the deepEqual, we define the function isObject() that checks whether an argument is a (JSON) object or not.
- A number 1 is represented with 8 byte floating point, so it's not a JSON object, but an empty object is a JSON object.

Code #

```
1 function isObject(object) {
2   return object != null && typeof object === 'object';
3 }
4 console.log(isObject(1)) // false
5 console.log(isObject({})) // true
```

- The first step is to check if two objects have the same keys (names in a JSON object).

Code #

```
1 function deepEqual(object1, object2) {
2   const keys1 = Object.keys(object1);
3   const keys2 = Object.keys(object2);
4   if (keys1.length !== keys2.length) {
5     return false;
6   }
7   ...
8   return true;
9 }
```

- Then, we retrieve the value from the key and check if they are all objects.
- If so, we recursively apply the deepEqual function.

Code #

```
1   for (const key of keys1) {
2     const val1 = object1[key];
3     const val2 = object2[key];
4     const areObjects = isObject(val1) && isObject(val2);
5     if (
6       areObjects && !deepEqual(val1, val2) ||
7       !areObjects && val1 !== val2
8     ) {
9       return false;
10    }
11  }
```

null and undefined Comparison

- When we use '==' to compare 'null' and 'undefined', it returns true.
- We can understand it, as '==' just compares if the two inputs are equivalent; like 1 and '1'.
- The '===' returns false when we compare 'null' and 'undefined.'

Code #

```
1 console.log(null == undefined) // true
2 console.log(null === undefined) // false
```