Java Script
Object-Oriented JavaScript

Assitant Prof. Dr Ali NİZAM

JavaScript objects

- JavaScript objects can be seen as mutable key-value-based collections
- Numbers, strings, and Booleans are object-like constructs that are immutable but have methods.

Javascript philosophy

- Most OOP systems define complex and unnecessary class and type hierarchies.
- Another big drawback was that in the pursuit of hiding the state, OOP considered the object state almost immaterial.
- JavaScript objects very versatile. In their seminal work,
  - Program to an interface and not to an implementation
  - Object composition over class inheritance

Inheritance

- In classical inheritance can't fundamentally differ from what you have got from the ancestors. This inhibits reuse. other problems as follows:
  - Inheritance introduces tight coupling. Child classes have knowledge about their ancestors. This tightly couples a child class with its parent.
  - When you subclass from a parent, you don't have a choice to select what you want to inherit and what you don't.
- Joe Armstrong (the inventor of Erlang) explains this situation very well—his now famous quote: "The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

Object literals

Json like key-value pair

```
var car = {
        type:"Fiat",
        model:"500",
        color:"White"
};
```

Object Methods

- Objects can also have **methods**.
- Methods are **actions** that can be performed on objects.
- Methods are stored in properties as **function definitions**.

Object Methods


```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

Accessing Object Methods

- You access an object method with the following syntax:
  - objectName.methodName()
- Example
  - name = person.fullName();
- If you access a method without the () parentheses, it will return the function definition:
  - name = person.fullName;

Lab:

- Create an employee object have property as firstName, lastName, sal[]
- Create salary array;
- Create a function totasalary: getTotalSalary() that return total salary of employee

Constructor

- Invocation as a constructor Constructor functions are declared just like any other functions and there's nothing special about a function that's going to be used as a constructor.
- However, the way in which they are invoked is very different.
- To invoke the function as a constructor, we precede the function invocation with the new keyword.
- When this happens, this is bound to the new object.

New Keyword

- It creates a new object. The type of this object is simply object.
- It sets this new object's internal, inaccessible, [[prototype]] (i.e. __proto__) property to be the constructor function's external, accessible, prototype object (every function object automatically has a prototype property).
- It makes the this variable point to the newly created object.
- It executes the constructor function, using the newly created object whenever this is mentioned.
- It returns the newly created object, unless the constructor function returns a non-null object reference. In this case, that object reference is returned instead.

https://stackoverflow.com/questions/1646698/what-is-the-new-keyword-in-javascript

new

- Once this is done, if an undefined property of the new object is requested, the script will check the object's [[prototype]] object for the property instead. This is how you can get something similar to traditional class inheritance in JavaScript.

- The most difficult part about this is point number 2. Every object (including functions) has this internal property called [[prototype]]. It can only be set at object creation time, either with new, with Object.create, or based on the literal (functions default to Function.prototype, numbers to Number.prototype, etc.). It can only be read with Object.getPrototypeOf(someObject). There is no other way to set or read this value.

- Functions, in addition to the hidden [[prototype]] property, also have a property called prototype, and it is this that you can access, and modify, to provide inherited properties and methods for the objects you make.

new

- ObjMaker = function() {this.a = 'first';};
- // ObjMaker is just a function, there's nothing special about it that makes // it a constructor.

- ObjMaker.prototype.b = 'second';
- // like all functions, ObjMaker has an accessible prototype property that // we can alter. I just added a property called 'b' to it. Like // all objects, ObjMaker also has an inaccessible [[prototype]] property // that we can't do anything with

- obj1 = new ObjMaker();
- // A new, empty object was created called obj1. At first obj1 was the same // as {}. The [[prototype]] property of obj1 was then set to the current // object value of the ObjMaker.prototype (if ObjMaker.prototype is later // assigned a new object value The ObjMaker function was executed, with // obj1 in place of this... so obj1.a was set to 'first'.

- obj1.a;   // returns 'first'
- obj1.b;
- // obj1 doesn't have a property called 'b', so JavaScript checks
- // its [[prototype]]. Its [[prototype]] is the same as ObjMaker.prototype
- // ObjMaker.prototype has a property called 'b' with value 'second'
- // returns 'second'

Prototypes

- Apart from the properties that we add to an object, there is one default property for almost all objects, called a prototype .
- When an object does not have a requested property, JavaScript goes to its prototype to look for it. The Object. getPrototypeOf() function returns the prototype of an object.
- Many programmers consider prototypes closely related to objects' inheritance
- They are indeed a way of defining object type but fundamentally, they are closely associated with functions.

Why?

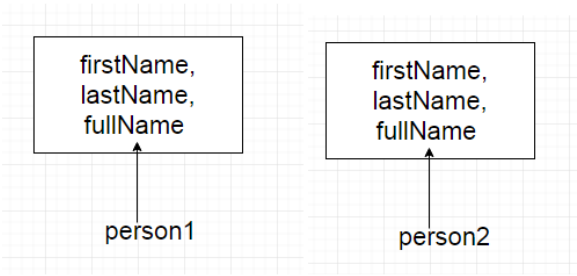
- Problem with creating objects with the constructor function:
- Consider the constructor function below:

```
function Human(firstName, lastName) {
        this.firstName = firstName,
        this.lastName = lastName,
        this.fullName = function() {
                return this.firstName + " " + this.lastName;
        }
}
```

```
var person1 = new Human("Virat", "Kohli");
```

```
console.log(person1)
```

```
var person1 = new Human("Virat", "Kohli");
var person2 = new Human("Sachin", "Tendulkar");
```



Prototype

- In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and inheritance is only of structure and behaviour. In ECMAScript, the state and methods are carried by objects, and structure, behaviour, and state are all inherited.

https://www.ecma-international.org/ecma-262/5.1/

JavaScript object’s type

- JavaScript has two types of objects:
  - function object
  - non-function object.
- Conceptually, all objects have a prototype (NOT A PROTOTYPE PROPERTY).
- Internally, JavaScript names an object's prototype as [[Prototype]].
- Only a function (a callable) object has the **prototype property**.

Extending object

- you can **not** add a new property to an existing object constructor:

- Human.prototype === person1.__proto__ //true

Prototypes

- var author = {};

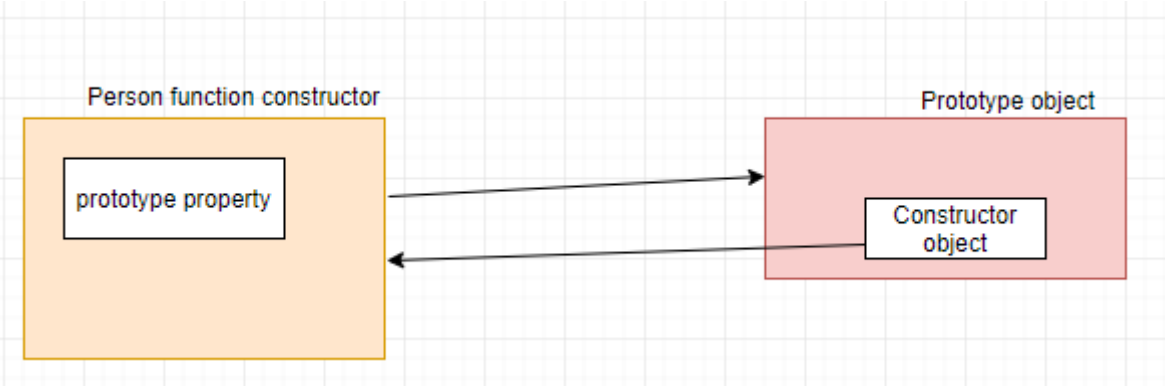
  author.firstname = 'Douglas';

  author.lastname = 'Crockford';

- you will immediately see that there is no encapsulation and the usual class structure

Prototype

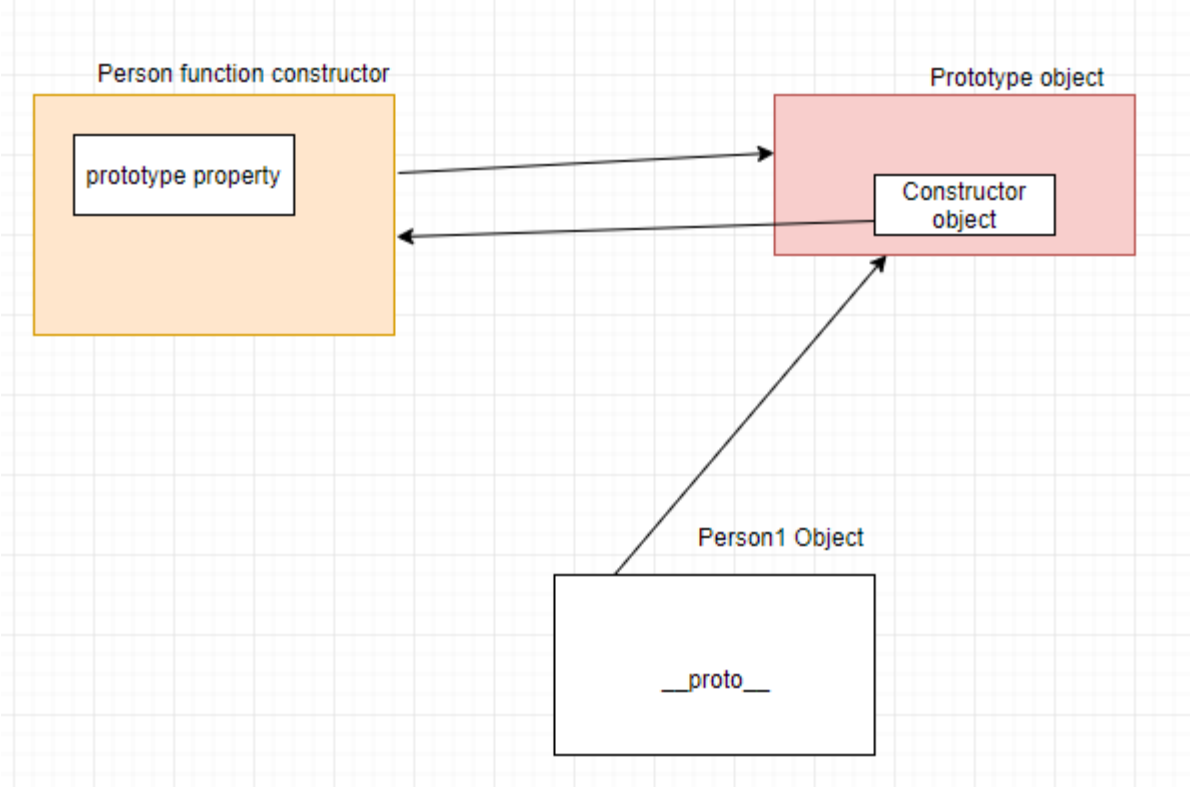

```
<script>
// function constructor
function Person(name, job, yearOfBirth){
    this.name= name;
    this.job= job;
    this.yearOfBirth= yearOfBirth;
}
// this will show Person's prototype property.
console.log(Person.prototype);
</script>
```



Person has a prototype property and that prototype property has a constructor object which again points to the Person constructor function.

https://www.geeksforgeeks.org/prototype-in-javascript/

Javascript Prototype



When we create an object using the above function constructor, JavaScript Engine will add dunder proto or __proto__ in the object which will point to the prototype's constructor object.

Prototype Example


```
function Person(name, job, yearOfBirth){
    this.name= name;
    this.job= job;
    this.yearOfBirth= yearOfBirth;
}
// adding calculateAge() method to the Prototype property
Person.prototype.calculateAge= function(){
    console.log('The current age is: '+(2019- this.yearOfBirth));
}
console.log(Person.prototype);

// creating Object Person1
let Person1= new Person('Jenni', 'clerk', 1986);
console.log(Person1)
let Person2= new Person('Madhu', 'Developer', 1997);
console.log(Person2)

Person1.calculateAge();
Person2.calculateAge();

</script>
```

Prototypes

- function Player() {};
- //Add a function to the prototype property of the function Player.prototype.usesBat = function () { return true; }
- //We call player() as a function and prove that nothing happens

```
var crazyBob = Player();
if (crazyBob === undefined) {
    console.log("CrazyBob is not defined");
}
```

- //Now we call player() as a constructor along with 'new' //1. The instance is created //2. method usesBat() is derived from the prototype of the function
- var swingJay = new Player();

```
if (swingJay && swingJay.usesBat && swingJay.usesBat()) {
    console.log("SwingJay exists and can use bat");
}
```

Instance properties versus prototype properties

- Instance property takes precedence when the same name is used

```
function Person(first, last, age) {
    this.firstName = first;
    this.lastName = last;
    this.age = function(){
        return age;
    };
}

p1 = new Person("Ahmet", "Ak", 39);
Person.prototype.age = function(){
        return 30;
};

console.log(p1.age());
```

- Return 39 instead of 30.

The possible variable declaration

- If a variable is declared using the var keyword in a function it is like private. They can be accessed by private function and privileged metot .
- Private functions are declared in an object constractor, They can be called by privileged methods.
- Privileged methods can be declared with this.method=function(){}
- Public methods are declared with Class.prototype.method=function(){}
- Public properties can be declared with this.property and accessed from outside the object.

The possible variable declaration

- while varhas been available in JavaScript since its initial releast, letand const are only available in ES6 (ES2015) and up. See this page for browser compatibility.
- Variables declared with var are available in the scope of the enclosing function. If there is no enclosing function, they are available globally.
  - Example:

```
function sayHello(){
  var hello = "Hello World";
  return hello;
}
console.log(hello);
```

- This will cause an error ReferenceError: hello is not defined, as the variable hello is only available within the function sayHello.

https://www.sitepoint.com/how-to-declare-variables-javascript/

Variable Declaration: let

- let is the descendant of var in modern JavaScript. Its scope is not only limited to the enclosing function, but also to its enclosing block statement.
- Syntax:

```
let x; // Declaration and initialization
x = "Hello World"; // Assignment

// Or all in one
let y = "Hello World";
```

Inheritence

- In classical inheritance, instance methods can not be invoked on a class definition itself.
- In prototypal inheritance, instance inherited from other instance.
- Each oject has a link to another object called its prototype

Inheritence


```
function Person() {}
Person.prototype.cry = function(){
   console.log('cry');
}
function Child() {}
Child.prototype={cry:Person.prototype.cry};
var aChild=new Child();
console.log(aChild instanceof Person);
```

subclass

- ObjMaker = function() {this.a = 'first';};
- ObjMaker.prototype.b = 'second';
- obj1 = new ObjMaker();
- // 3 things just happened A new, empty object was created called obj1. At first obj1 was the same as {}. The [[prototype]] property of obj1 was then set to the current object value of the ObjMaker.prototype (if ObjMaker.prototype is later assigned a new object value, obj1's [[prototype]] will not change, but you can alter the properties of ObjMaker.prototype to add to both the prototype and [[prototype]]). The ObjMaker function was executed, with obj1 in place of this... so obj1.a was set to 'first'.

- SubObjMaker = function () {};
- SubObjMaker.prototype = new ObjMaker(); // note: this pattern is deprecated!
- // Because we used 'new', the [[prototype]] property of SubObjMaker.prototype // is now set to the object value of ObjMaker.prototype.
- // The modern way to do this is with Object.create(), which was added in ECMAScript 5: // SubObjMaker.prototype = Object.create(ObjMaker.prototype);

- SubObjMaker.prototype.c = 'third';
- obj2 = new SubObjMaker();
- // [[prototype]] property of obj2 is now set to SubObjMaker.prototype Remember that the [[prototype]] property of SubObjMaker.prototype
- // is ObjMaker.prototype. So now obj2 has a prototype chain! obj2 ---> SubObjMaker.prototype ---> ObjMaker.prototype

- obj2.c; // returns 'third', from SubObjMaker.prototype
- obj2.b; // returns 'second', from ObjMaker.prototype
- obj2.a; // returns 'first', from SubObjMaker.prototype, because SubObjMaker.prototype was created with the ObjMaker function, which assigned a for us

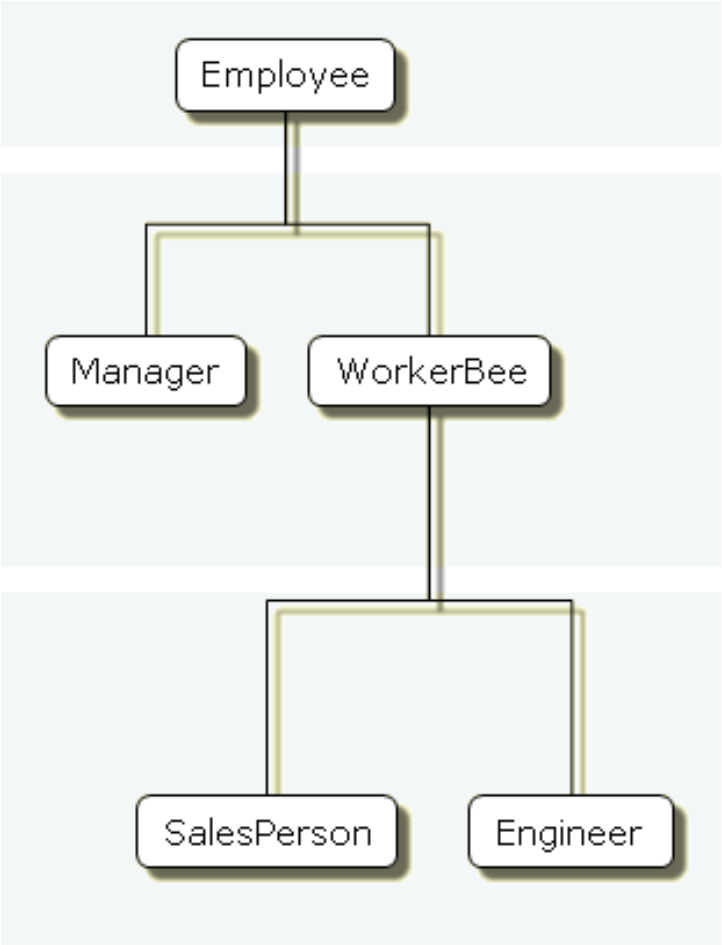
Java Script vs Java


```
function Employee() {
  this.name = '';
  this.dept = 'general';
}


function Manager() {
  Employee.call(this);
  this.reports = [];
}
Manager.prototype = Object.create(Employee.prototype);
Manager.prototype.constructor = Manager;

function WorkerBee() {
  Employee.call(this);
  this.projects = [];
}
WorkerBee.prototype = Object.create(Employee.prototype);
WorkerBee.prototype.constructor = WorkerBee;
```

```
public class Employee {
   public String name = "";
   public String dept = "general";
}


public class Manager extends Employee {
   public Employee[] reports =
       new Employee[0];
}
public class WorkerBee extends Employee {
   public String[] projects = new String[0];
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

Object hierarchy



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

Java Script vs Java


```
function SalesPerson() {
  WorkerBee.call(this);
  this.dept = 'sales';
  this.quota = 100;
}
SalesPerson.prototype =
Object.create(WorkerBee.prototype);
SalesPerson.prototype.constructor = SalesPerson;

function Engineer() {
  WorkerBee.call(this);
  this.dept = 'engineering';
  this.machine = '';
}
Engineer.prototype =
Object.create(WorkerBee.prototype)
Engineer.prototype.constructor = Engineer;
```

```
public class SalesPerson extends WorkerBee {
  public String dept = "sales";
  public double quota = 100.0;
}

public class Engineer extends WorkerBee {
  public String dept = "engineering";
  public String machine = "";
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

Working Example

- var jim = new Employee;
- // Parentheses can be omitted if the // constructor takes no arguments.
- // jim.name is ‘’ // jim.dept is 'general'

- var sally = new Manager;
- // sally.name is ‘’ // sally.dept is 'general’ // sally.reports is []

- var mark = new WorkerBee;
- // mark.name is ‘’ // mark.dept is 'general’ // mark.projects is []

- var fred = new SalesPerson;
- // fred.name is ‘’ // fred.dept is 'sales’ // fred.projects is [] // fred.quota is 100

- var jane = new Engineer;
- // jane.name is ‘’ // jane.dept is 'engineering'
- // jane.projects is []
- // jane.machine is "

Class Keyword ECMAScript 2015

- JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

Class declarationsn

- One way to define a class is using a class declaration. To declare a class, you use the class keyword with the name of the class ("Rectangle" here).

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Hoisting and Classes

- An important difference between function declarations and class declarations is that function declarations are hoisted and class declarations are not. You first need to declare your class and then access it, otherwise code like the following will throw a ReferenceError:

  - const p = new Rectangle(); // ReferenceError

  - class Rectangle {}

Class expressions

- A class expression is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body. (it can be retrieved through the class's (not an instance's) name property, though).

```
// unnamed
let Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
console.log(Rectangle.name);
// output: "Rectangle"
```

Sub classing with extends

The extends keyword is used in class declarations or class expressions to create a class as a child of another class.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' makes a noise.');
  }
}

class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }
  speak() {
    console.log(this.name + ' barks.');
  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

Classes Are Functions

- A JavaScript class is a type of function. Classes are declared with the class keyword. We will use function expression syntax to initialize a function and class expression syntax to initialize a class.
- With prototypes, any function can become a constructor instance using the new keyword.

Defining Methods

- The common practice with constructor functions is to assign methods directly to the prototype instead of in the initialization, as seen in the greet() method below.

- constructor.js
- function Hero(name, level) {
- this.name = name;
- this.level = level;
- }

- // Adding a method to the constructor
- Hero.prototype.greet = function() {
- return `${this.name} says hello.`;
- }

https://www.digitalocean.com/community/tutorials/understanding-classes-in-javascript

Defining Methods


```
class Hero {
    constructor(name, level) {
        this.name = name;
        this.level = level;
    }

    // Adding a method to the constructor
    greet() {
        return `${this.name} says hello.`;
    }
}
```

Defining Methods

- console.log(hero1), we can see more details about what is happening with the class initialization.

- Output
- Hero {name: "Varg", level: 1}
- __proto__:
- ▶ constructor: class Hero
- ▶ greet: *f* greet()

- We can see in the output that the constructor() and greet() functions were applied to the __proto__, or [[Prototype]] of hero1, and not directly as a method on the hero1 object. While this is clear when making constructor functions, it is not obvious while creating classes. Classes allow for a more simple and succinct syntax, but sacrifice some clarity in the process.

The Object.create() method

- creates a new object, using an existing object as the prototype of the newly created object.
- const person = {
- isHuman: false,
- printIntroduction: function () {
- console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);
- }
- };
-
- const me = Object.create(person);
-
- me.name = "Matthew"; // "name" is a property set on "me", but not on "person"
- me.isHuman = true; // inherited properties can be overwritten
-
- me.printIntroduction();
- // expected output: "My name is Matthew. Am I human? true"
-

Questions