OOP Using JavaScript

Outline

- JavaScript OOP
  - Object Literal using JSON
  - Class-based OOP
  - Object-Based Inheritance
- Modules
- NPM - Node Package Manager

JavaScript OOP

Properties & Methods

JavaScript OOP

- JavaScript object is a collection of **properties**
- An **object property** is association between a **name and a value**. A value can be either:
  - a **data** (e.g., a number or a string) or
  - a **method** (i.e., function)
- An object can be either instantiated from a class or it can be created from another object
- Classes and objects can be altered during the execution of a program

OOP in JavaScript

JavaScript has 3 ways to create an objects:

- **Object Literal**: create an object using JSON
- **Class-based OOP**: create a class then instantiate objects from the class
- **Object-Based Inheritance**: create objects from other objects
  - Creates new copies of objects from an existing object
  - Code reuse done by **cloning**

```
e.g,  let myCat = Object.create(animal);
```

Object Literal using JSON

Create an Object Literal using JSON

```
var person = {
    firstName: 'Samir',
    lastName: 'Saghir',
    height: 54,
    name () {
      return this.firstName + ' ' + this.lastName;
    }
};

//Two ways to access the object properties
console.log(person['height'] === person.height);

console.log(person.name());
```

Creating an object using {} or new Object()

- Another way to create an object is to use the built-in **`Object`** data type or simply assigning **{}** to the variable

```
var joha = {}; //or new Object();
joha.name = "Juha Nasreddin";
joha.age = 28;

joha.toString = function() {
      return 'Name: ' + this.name + ', Age: '
                    + this.age;
};
```

Object Literals

- ES2015 adds a new feature to the way of defining properties:
  - Instead of

```
let name = 'Samir Saghir',
    age = 25;
let person = {
  name: name,
  age: age
};
```

  - We can do just:

```
let name = 'Samir Saghir';
age = 25;
let person = {
  name,
  age
};
```

Get, set and delete

- **get**

  object.name

  object[expression]

- **set**

  object.name = value;

  object[expression] = value;

- **delete**

  delete object.name

  delete object[expression]

JSON.stringify and JSON.parse

```
/* Serialise the object to a string in JSON
  format -- only attributes getr serialised */

var jsonString = JSON.stringify(person);
console.log(jsonString);


//Deserialise a JSON string to an object
//Create an object from a string!

var personObject = JSON.parse(jsonString);
console.log(personObject);
```

- More info https://developer.mozilla.org/en-US/docs/JSON

Destructuring Object

- Destructuring assignments allow to extract values from an object and assign them to variables in an easier way:

```
let person = {
  name: 'Samir Saghir',
  address: {
    city: 'Doha',
    street: 'University'
  }
};

let {name, address: {city}} = person;
console.log(name, city);
```

Class-based OOP

Class-based OOP

- Class-based OOP uses classes and inheritance

```
class Person extends Mammal {
  constructor(fname, lname, age){
    super(age);
    this.fname = fname;
    this.lname = lname;
  }
  get fullname() {
    return `{this.firstname} ${this.lname}`;
  }
  set fullname(newfullname) {
    //setter property of fullname
  }
  toString() {
  }
  // more class members…
}
```

Constructor of the class

Getters and setters

Methods

Object-Based Inheritance

Object-Based Inheritance

- Object-Based Inheritance enables creating objects from other objects (instead of creating them from classes)
  - Instead of creating classes, you **make prototype objects**, and then use the **Object.create**(…) to make new instances that inherit form the prototype object
  - Customize the new objects by adding new properties and methods
- We don't need classes to make lots of similar objects. **Objects inherit from objects!**

Object-Based Inheritance

- Create an object from another object! Clone an object then customize it. The cloned object inherits the properties and methods of the source object.

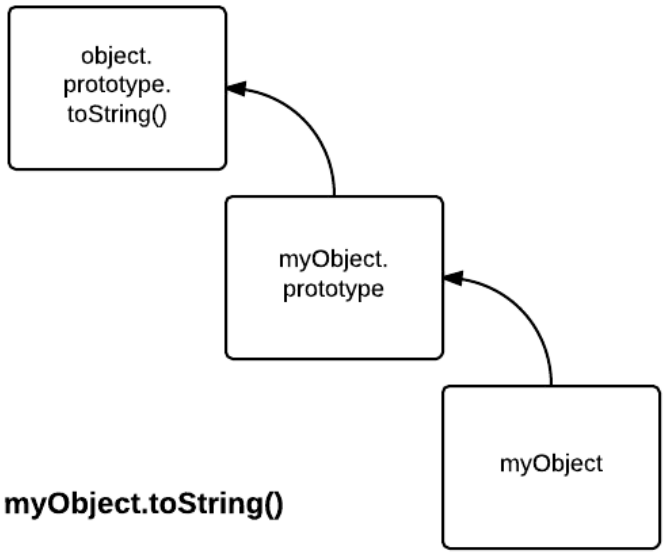
  - See ***7.object-based-inheritance.js*** example

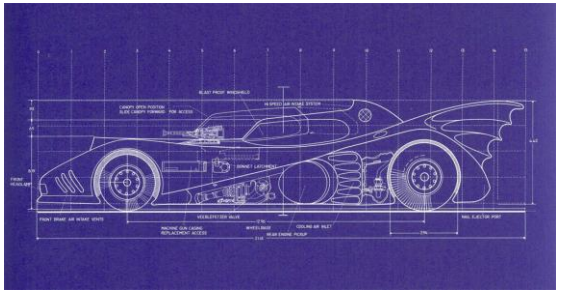
```
let animal = {
    eyes: 2,
    legs: 4,
    name: "Animal",
    toString () {
     return this.name + " with " + this.eyes + " eyes & " + this.legs + " legs."
    }
}

let myDog = Object.create(animal);
myDog.name = "Max";
//Add a new property to myDog object.
myDog.avgLifeSpan = 13;
myDog.speak = function() {
    console.log(`${this.name}.speak... Woof, Woof`);
}
```


17

Prototype Chain

```
▼ myCar: Car
  ▼ __proto__: Vehicle
    ▼ __proto__: Machine
        whoAmI: "I am a machine"
      ▼ __proto__: Machine
        ▶ constructor: function Machine() {
        ▶ __proto__: Object
```





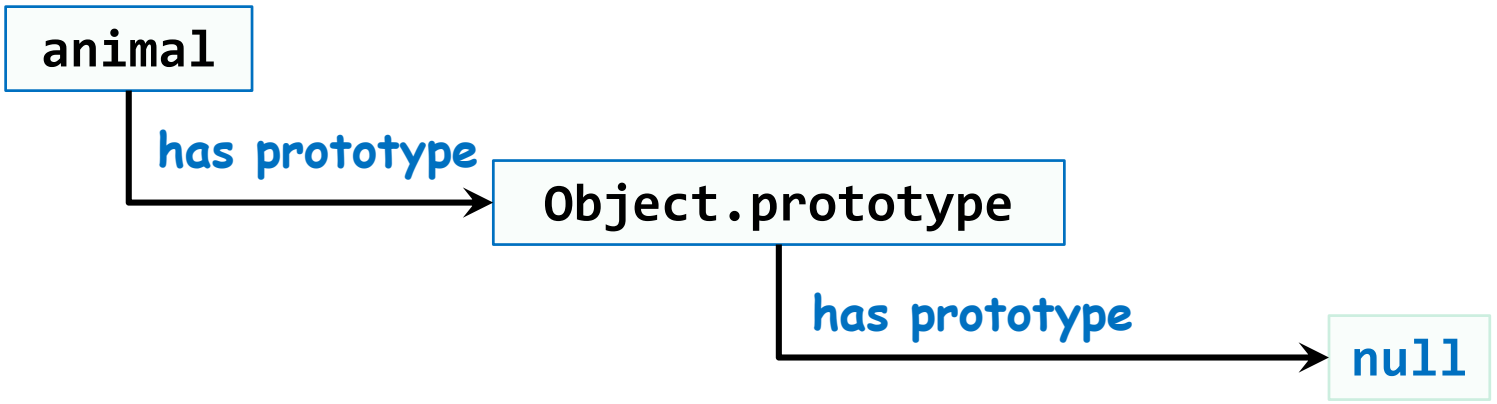
- Inheritance in JavaScript is based on the **Prototype Chain**
- Every object has a an internal prototype property pointing to another object or null. It can be used to:
  - **Extend a class** (i.e., add properties and methods to a class)
  - **Implement inheritance**
- Can be accessed using the __proto__ property

Object Prototypes: Example

- Every object has its own prototype
  - By default, set to **Object.prototype**
  - This forms the so called "Prototype chain"
  - Object has for prototype null, ending the prototype chain

```
let animal = {
  /* properties and methods */
};
```

- **The prototype chain is:**



Prototype can be used to extend classes

- We can use the `prototype` object to add custom properties / methods to a class
  - That is reflected on all instances of the class
  - Simply reference the **`prototype`** property on the class before adding the custom property

See *6.inheritance2.js*

```
class Circle {
}
Circle.prototype.pi = 3.14159;
Circle.prototype.radius = 5;
Circle.prototype.calculateArea = function () {
  return this.pi * this.radius * 2;
}
let circle = new Circle();
let area = circle.calculateArea();
console.log(area);  // 31.4159
```

Using prototype object to Add Functionality to Build-in Classes

- Dynamically add a function to a built-in class at runtime using the **prototype** object:

```
//adding a method to arrays to sum their number elements
Array.prototype.sum = function(){
  let sum = 0;
  for(let e of this){
    if(typeof e === "number"){
      sum += e;
    }
  }
  return sum;
}

let numbers = [1,2,3,4,5];
console.log(numbers.sum());   //logs 15
```

Attaching a method to the Array class

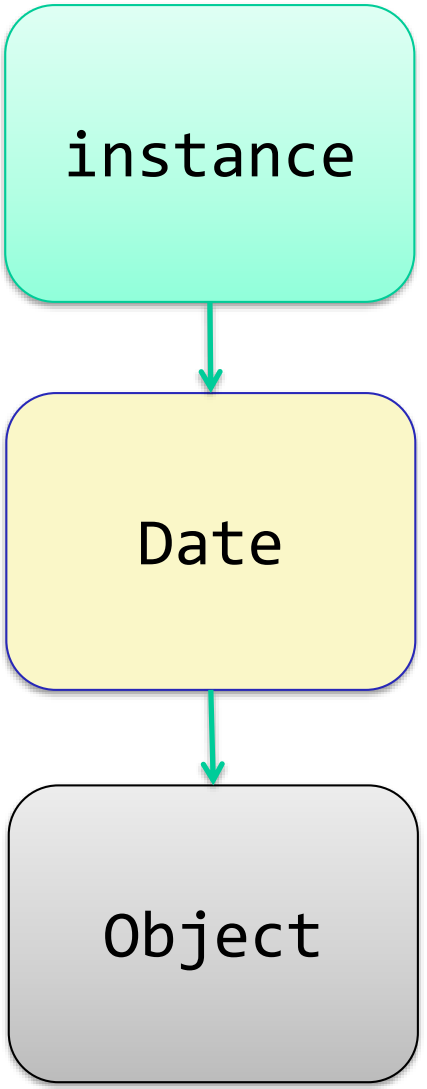
Here this means the array

The Prototype Chain

- Objects in JavaScript can have only a single prototype
  - Their prototype also has a prototype, etc…
  - This is called the **prototype chain**
- When a property is called on an object
  - This object is searched for the property
  - If the object does not contain such property, its prototype is checked for the property, etc…
  - If a null prototype is reached, the result is undefined

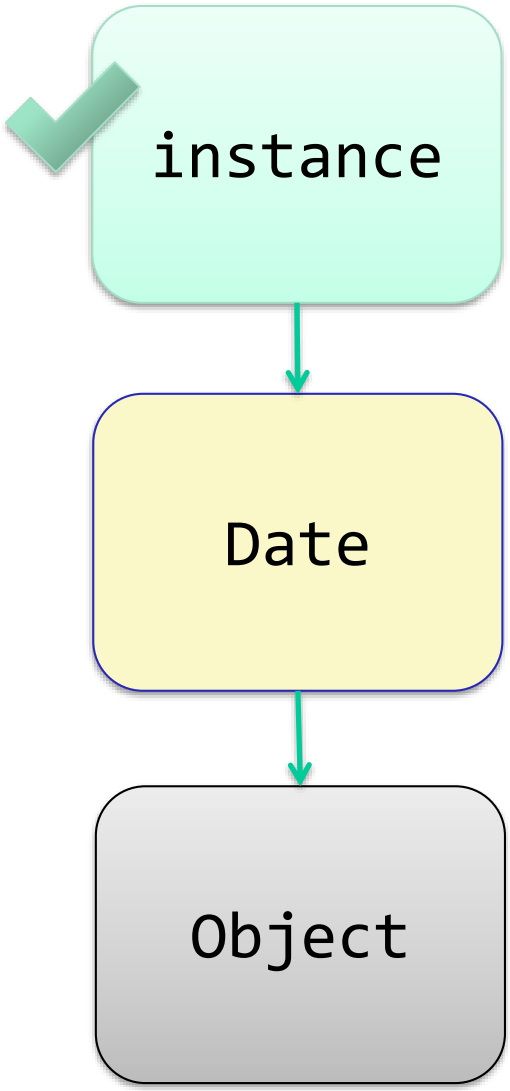
Property lookup chain

```
1 var instance = new Date();
2 instance.foo = function() { alert(“bar”); };
3
4 instance.foo();
5 instance.getTime();
6 instance.hasOwnProperty(“foo”);
```



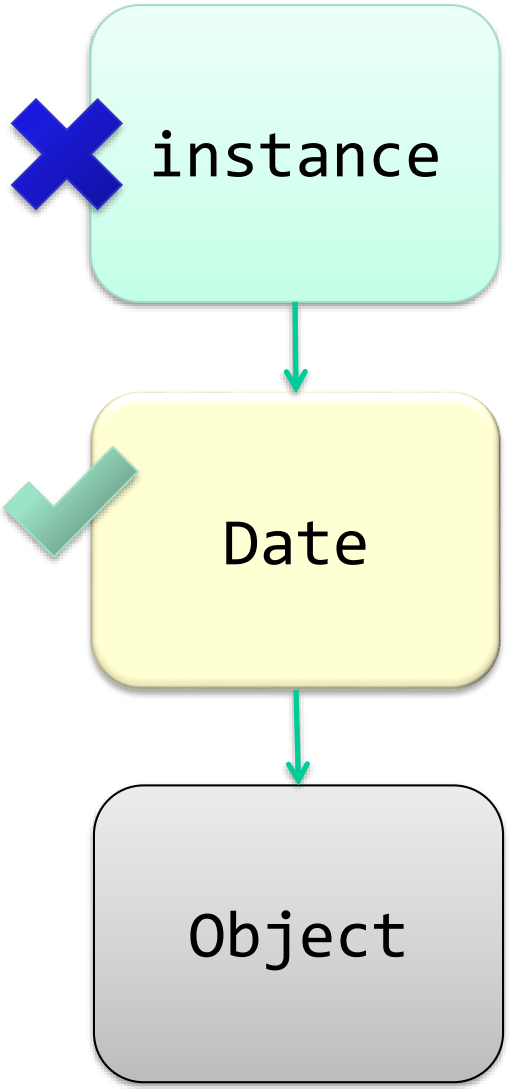
Property lookup chain (look up instance.foo)

```
1  var instance = new Date();
2  instance.foo = function() { alert("bar"); };
3
4  instance.foo();
5  instance.getTime();
6  instance.hasOwnProperty("foo");
7
8
9
10
11
12
```



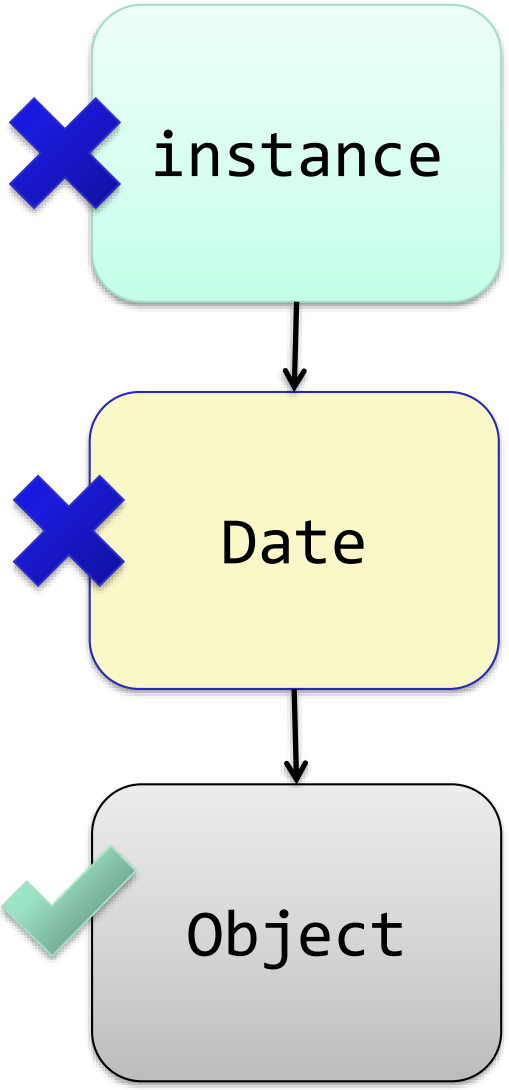
Property lookup chain (lookup instance.getTime)

```
1 var instance = new Date();
2 instance.foo = function() { alert("bar"); };
3
4 instance.foo();
5 instance.getTime();
6 instance.hasOwnProperty("foo");
```



Property lookup chain (look up instance.hasOwnProperty)

```
1 var instance = new Date();
2 instance.foo = function() { alert("bar"); };
3
4 instance.foo();
5 instance.getTime();
6 instance.hasOwnProperty("foo");
```



Modules

Node.js Modules

- An elegant way of encapsulating and reusing code
- Node has a simple module loading system
  - Files and modules are in one-to-one correspondence

The variable **PI** is **private** to circle.js

circle.js

```
let PI = Math.PI;
exports.area = function (r) {return PI * r * r;};
exports.circumference = function (r) {
    return 2 * PI * r;
};
```

index.js

```
let circle = require('./circle.js');
console.log('The area of radius 4: '+
                                circle.area(4));
```

NPM
Node Package Manager

https://www.npmjs.com/

253,375
total packages

165,100,502
downloads in the last day

919,014,128
downloads in the last week

3,877,933,925
downloads in the last month

Node Package Management (NPM)

- npm is used to download Node.js packages. First,

  `npm init`

  can be used to initialize an *package.json* file to define the **project dependencies**

```
$ npm init
//enter package details
name: "NPM demos"
version: 0.0.1
description: "Demos for the NPM package management"
entry point: main.js
test command: test
git repository: http://github.com/user/repository-name
keywords: npm, package management
author: ae@qu.edu.qa
license: MIT
```

Node Package Management (NPM)

- Installing modules

```
$ npm install package-name [--save]
```

  - Installs a package and adds dependency in *package.json*

```
npm install angular2 systemjs --save
```

- Do not push the downloaded packages to github by adding *node_modules/* to .gitignore file

- When getting a project before running it do:

```
$ npm install
```

  - Installs all missing packages from *package.json*

ES 2015 Modules

- ES2015 introduced modules that enables us to write modular code.
  - Each file decides what to export from its module
  - ES2015 modules are mainly use for client-side scripts
- Export the objects you want from a module:

```
// Car.js
export class Car { ... }
export class Convertible extends Car { ... }
```

- Use the module in another file:

```
// App.js
import {Car, Convertible} from 'Car';
let bmw = new Car();
let cabrio = new Convertible();
```

Resources

- Learn ES2015

https://babeljs.io/docs/learn-es2015/

- Best ES 2015 eBook

http://exploringjs.com/es6/

- Best ES 2015 Learning Resources

https://github.com/ericdouglas/ES2015-Learning

- More Examples

http://www.es6fiddle.net/