



OOP Using JavaScript

Outline

- JavaScript OOP
 - Object Literal using JSON
 - Class-based OOP
 - Prototypal Inheritance
- Prototype Chain
- Modules

JavaScript OOP

Properties & Methods

JavaScript OOP

- JavaScript object is a dynamic collection of **properties**
- An **object property** is association between a **key** and a **value**.
 - **Key** is a string that is unique within that object.
 - **Value** can be either:
 - a **data** (e.g., a number or a string) or
 - a **method** (i.e., function)
- 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 notation
- **Instantiate a Class**: create a class then instantiate objects from the class
- **Create an object based on another object**:
prototype-based programming
 - Make a prototype object then make new instances from it (objects inherit from objects)
 - Augment the new instances with new properties and methods

```
let cat = { legs : 4, eyes: 2 };  
let myCat = Object.create(cat);  
myCat.name = 'Garfield';
```

Object Literal using JSON

Create an Object Literal using JSON

```
let person = {  
  firstName: 'Samir',  
  lastName: 'Saghir',  
  height: 54,  
  getName () {  
    return `${this.firstName} ${this.lastName}`;  
  }  
};
```

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

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

Creating an object using {}

- Another way to create an object is to simply assigning {} to the variable

```
let joha = {}; //or new Object();  
joha.name = "Juha Nasreddin";  
joha.age = 28;  
  
joha.toString = function() {  
    return `Name: ${this.name} Age: ${this.age}`;  
};
```

```
//Creating an object using variables  
let name = 'Samir Saghir'; age = 25;  
let person = {name, age};
```


Get, set and delete

- **get**

`object.name`

- **set**

`object.name = value;`

- **delete**

`delete object.name`

JSON.stringify and JSON.parse

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

```
let jsonString = JSON.stringify(person);  
console.log(jsonString);
```

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

```
let 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 = {  
  firstname: 'Ali', lastname: 'Faleh', age: 18, gpa: 3.6,  
  address: {  
    city: 'Doha',  
    street: 'University St'  
  }  
}  
  
let { firstname, lastname, address: {city}, ...otherDetails } = person
```

Rest operator (...) assigns the remaining properties to the *otherDetails* variable

Class-based OOP

Class-based OOP

- Class-based OOP uses classes

```
class Person {  
  constructor(firstname, lastname){  
    this.firstname = firstname;  
    this.lastname = lastname;  
  }  
  
  get fullname() {  
    return `${this.firstname} ${this.lastname}`;  
  }  
  
  set fullname(fullname) {  
    [this.firstname, this.lastname] = fullname.split(" ");  
  }  
  
  greet() {  
    return `Hello, my name is ${this.fullname}`;  
  }  
}
```

Constructor of the class

Getter, defines a
computed property

Method

Class-based Inheritance

- A class can extend another one

```
class Student extends Person {  
    constructor(firstname, lastname, gpa){  
        super(firstname, lastname);  
        this.gpa = gpa;  
    }  
    greet() {  
        return `${super.greet()}. My gpa is ${this.gpa}`;  
    }  
}
```

```
let student1 = new Student("Ali", "Faleh", 3.5);  
//Change the first name and last name  
student1.fullname = "Ahmed Saleh";  
console.log(student1.greet());
```

Prototype property can be used to extend a class

- Classes has a special property called **prototype**
- It can be used to add properties / methods to a class
 - Change reflected on all instances of the class

```
class Circle {  
    constructor(r) {  
        this.radius = r;  
    }  
}  
  
let circle = new Circle(3.5);  
  
//Add getArea method to the class at runtime  
Circle.prototype.getArea = function () {  
    return Math.PI * this.radius * 2;  
}  
  
let area = circle.getArea();  
console.log(area); // 21.9
```

Using **prototype** property to Add Functionality even to Build-in Classes

- Dynamically add a function to a built-in class using the **prototype** property:

Attaching a method to the Array class

```
Array.prototype.getMax = function() {  
    let max = Math.max(...this);  
    return max;  
}
```

Here **this** means the array

```
let numbers = [9, 1, 11, 3, 4];  
let max = numbers.getMax();
```


Prototypal Inheritance

Prototypal Inheritance

- Prototypal Inheritance (aka Object-Based Inheritance) enables creating an object from other object
 - Instead of creating classes, you **make prototype object**, and then use **Object.setPrototypeOf(..)** or **Object.create(..)** to make new instances that inherit from 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!**

Example

```
let cat = { legs : 4, eyes: 2 };
```

```
let myCat = { name: 'Garfield' };
```

```
Object.setPrototypeOf(myCat, cat);
```

```
myCat.breed = 'Persian';
```

```
console.log( `${myCat.name} is a ${myCat.breed}  
cat with ${myCat.legs} legs  
and ${myCat.eyes} eyes` );
```

Prototypal Inheritance

- Make an object that you like (i.e., prototype object)
- Create new instances from that object
 - Resulting object **maintains an explicit** link (**delegation** pointer) to its prototype
 - JavaScript runtime is capable of dispatching the correct method or finding the value of a property by simply following a series of delegation pointers (i.e., Prototype Chain) until a match is found
- Changes in the prototype are visible to the new instances
- New objects can add their own custom properties and methods

The spread operator (...)

- The spread operator (...) is used to merge one or more objects to a target object while **replacing** values of properties with matching names
 - Used for cloning => no inheritance
- Alternative way is to use **Object.assign**

```
let movie1 = {  
  name: 'Star Wars',  
  episode: 7  
};
```

//We clone movie 1 and override the episode property

```
let movie2 = {...movie1, episode: 8, rating: 5};
```

//Another way of doing the same using Object.assign

```
//let movie2 = Object.assign({}, movie1, { episode: 8, rating: 5});
```

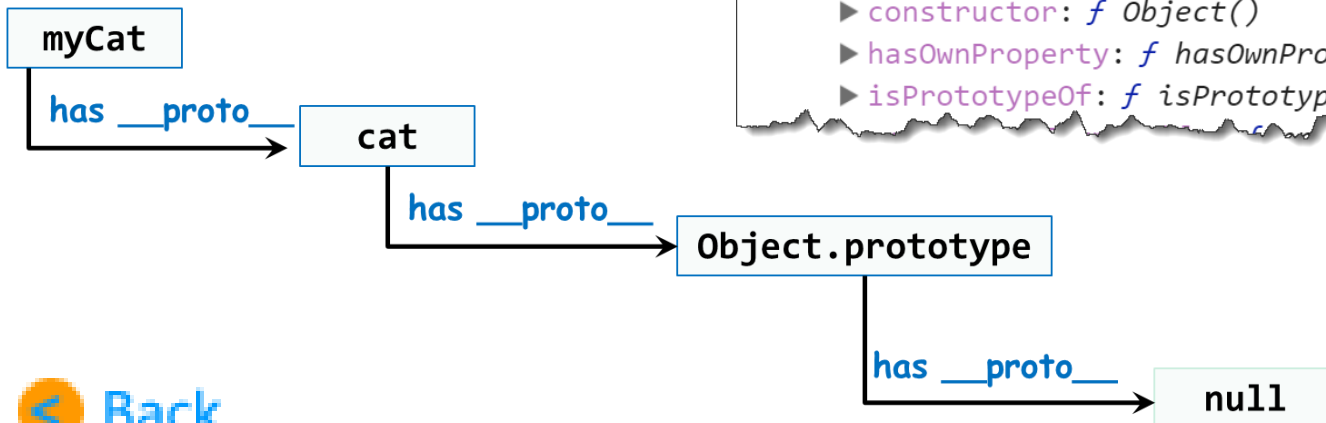
```
console.log('\n');
```

```
console.log(movie1.name, "movie1.episode: ", movie1.episode); // writes 7
```

```
console.log(movie2.name, "movie2.episode: ", movie2.episode); // writes 8
```

Prototype Chain

```
▼ {name: "Garfield", breed: "Persian"} ⓘ  
  breed: "Persian"  
  name: "Garfield"  
  ▼ __proto__:  
    eyes: 2  
    legs: 4  
    tail: 1  
    ▼ __proto__:  
      ▶ constructor: f Object()  
      ▶ hasOwnProperty: f hasOwnProperty()  
      ▶ isPrototypeOf: f isPrototypeOf()
```



Prototype Chain

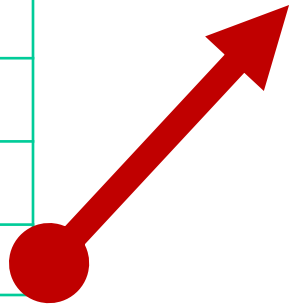
- **Prototype Chain** is the mechanism used for inheritance in JavaScript
 - Establish behavior-sharing between objects using delegation pointers (called Prototype Chain)
- Every object has a an internal **__proto__** property **pointing** to another object
 - **Object.prototype.__proto__** equals null
- It can be accessed using **Object.getPrototypeOf(obj)** method

```
let cat = {  
  name : 'cat',  
  legs : 4,  
  eyes : 2  
};
```

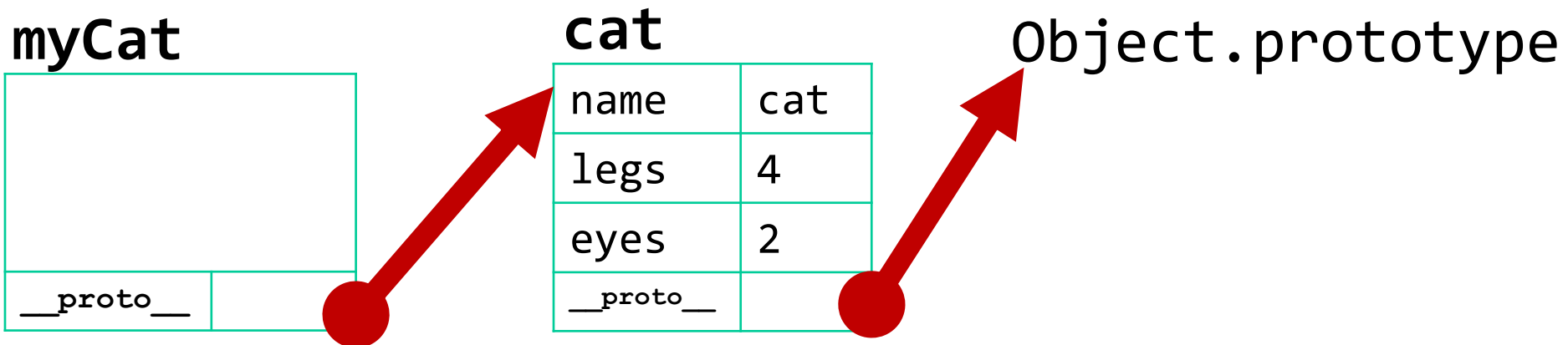
cat

name	cat
legs	4
eyes	2
__proto__	

Object.prototype



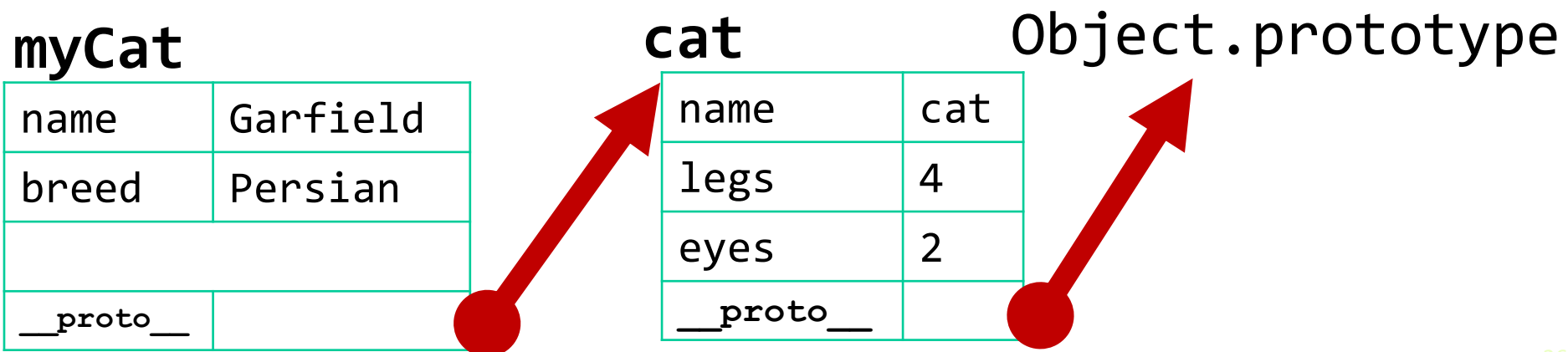

```
let cat = {  
  name : 'cat',  
  legs : 4,  
  eyes : 2  
};  
let myCat = Object.create(cat);
```



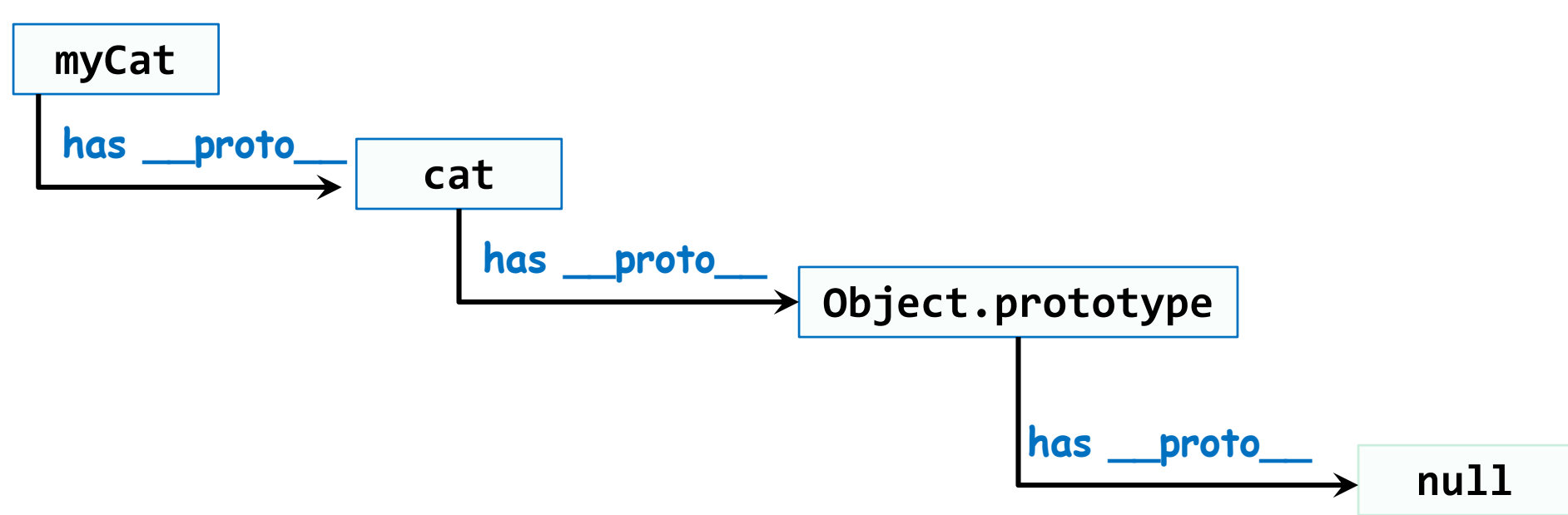
```
let cat = {  
  name : 'cat',  
  legs : 4,  
  eyes : 2  
};
```

Changes to a child object are always recorded in the child object itself and never in its prototype (i.e. the child's value **shadows** the prototype's value rather than changing it).

```
let myCat = Object.create(cat);  
myCat.name = 'Garfield';  
myCat.breed = 'Persian';
```



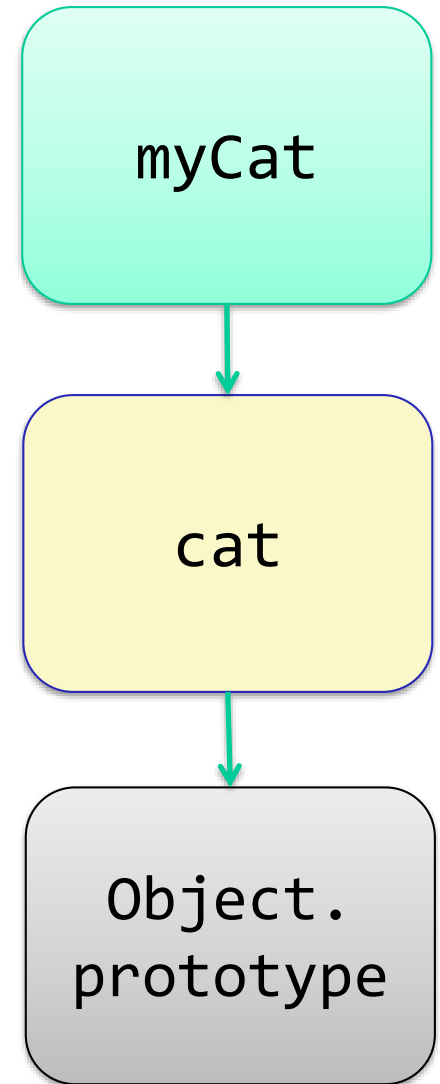
Prototype Chain example



`__proto__` is the actual object that is used in **the lookup the chain** to resolve methods

Prototype Chain

```
let cat = {  
  name : 'cat',  
  legs : 4,  
  eyes : 2  
};  
let myCat = Object.create(cat);  
myCat.name = 'Garfield';  
myCat.breed = 'Persian';
```



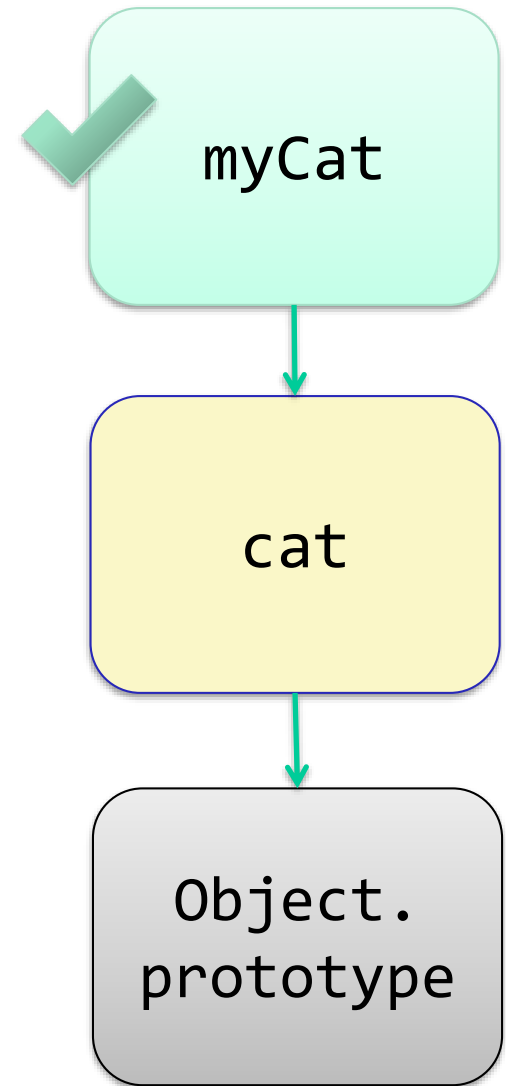
Prototype Chain (lookup myCat.name)

```
let cat = { name: 'cat', legs : 4, eyes: 2 };  
let myCat = { name: 'Garfield' };  
Object.setPrototypeOf(myCat, cat);  
myCat.name = 'Garfield';  
myCat.breed = 'Persian';
```

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

```
console.log(myCat.legs);
```

```
console.log(myCat.hasOwnProperty('eyes'));
```



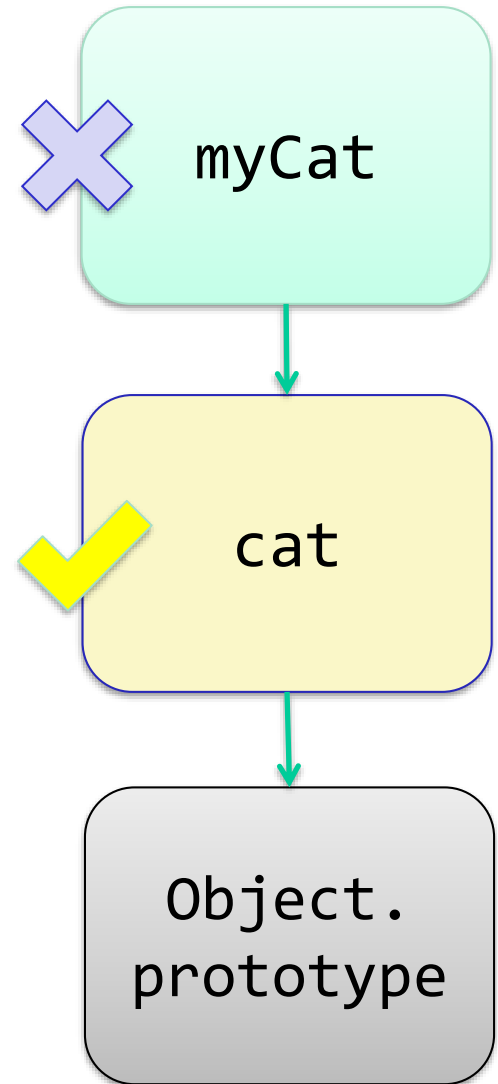
Prototype Chain (lookup myCat.legs)

```
let cat = { name: 'cat', legs : 4, eyes: 2 };  
let myCat = { name: 'Garfield' };  
Object.setPrototypeOf(myCat, cat);  
myCat.name = 'Garfield';  
myCat.breed = 'Persian';
```

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

```
console.log(myCat.legs);
```

```
console.log(myCat.hasOwnProperty('eyes'));
```



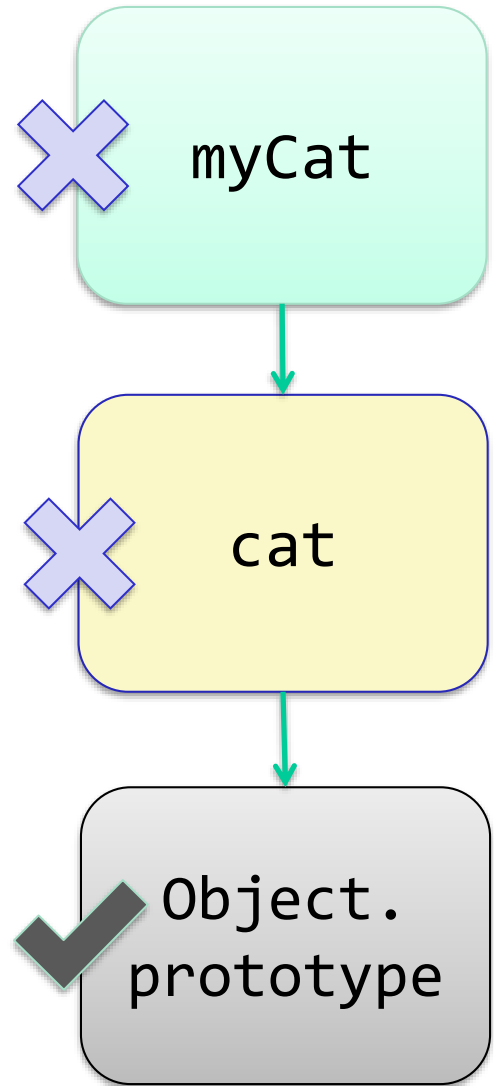
Prototype Chain (lookup myCat. hasOwnProperty)

```
let cat = { name: 'cat', legs : 4, eyes: 2 };  
let myCat = { name: 'Garfield' };  
Object.setPrototypeOf(myCat, cat);  
myCat.name = 'Garfield';  
myCat.breed = 'Persian';
```

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

```
console.log(myCat.legs);
```

```
console.log(myCat.hasOwnProperty('eyes'));
```



Modules

CommonJS Modules

- Modules allow reusing code stored in different .js files
- CommonJS Modules implemented by Node.js for synchronous module loading system (files correspond to modules)

circle.js

```
//Export 2 functions to make functions available in other files  
exports.area = r => Math.PI * r ** 2;  
exports.circumference = r => 2 * Math.PI * r;
```

app.js

```
const circle = require('./circle');  
console.log(`The area of radius 4: ${circle.area(4)}`);
```

calculator.js

```
class Calculator {  
  ...  
}  
module.exports = new Calculator();
```

app.js

```
const calculator = require('./calculator');
```

ES6 Modules

- ES6 introduced new modules syntax
 - ES6 modules are supported by most browsers
 - Node.js has an initial support (using node `--experimental-modules` flag)
- 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();
```

Node Package Management (NPM)

- npm is used to download Node.js packages from <https://npmjs.com>
 - First, **npm init** can be used to initialize a *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)

- ◆ Install a package and adds dependency in *package.json* using **npm install package-name**

```
npm install axios
```

```
npm install mocha -D
```

// -D for installing dev dependencies (not needed in production)

- ◆ Do not push the downloaded packages to GitHub by adding *node_modules/* to *.gitignore* file
- ◆ When cloning a project from GitHub before running it do:

```
$ npm install
```

=> Installs all missing packages from *package.json*

Resources

- Best JavaScript eBooks

<http://exploringjs.com/es6/>

<http://exploringjs.com/es2016-es2017/>

<http://exploringjs.com/es2018-es2019/>

- More Resources

<https://github.com/ericdouglas/ES6-Learning>