



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 an 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., number, string, object ...) 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

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

Object Literal using JSON

Create an Object Literal using JSON

(JavaScript Object Notation)

```
const 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. Then add properties and methods

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


Get, set and delete

- **get**
object.name
- **set**
object.name = value;
- **delete**
delete object.name

JSON.stringify and JSON.parse

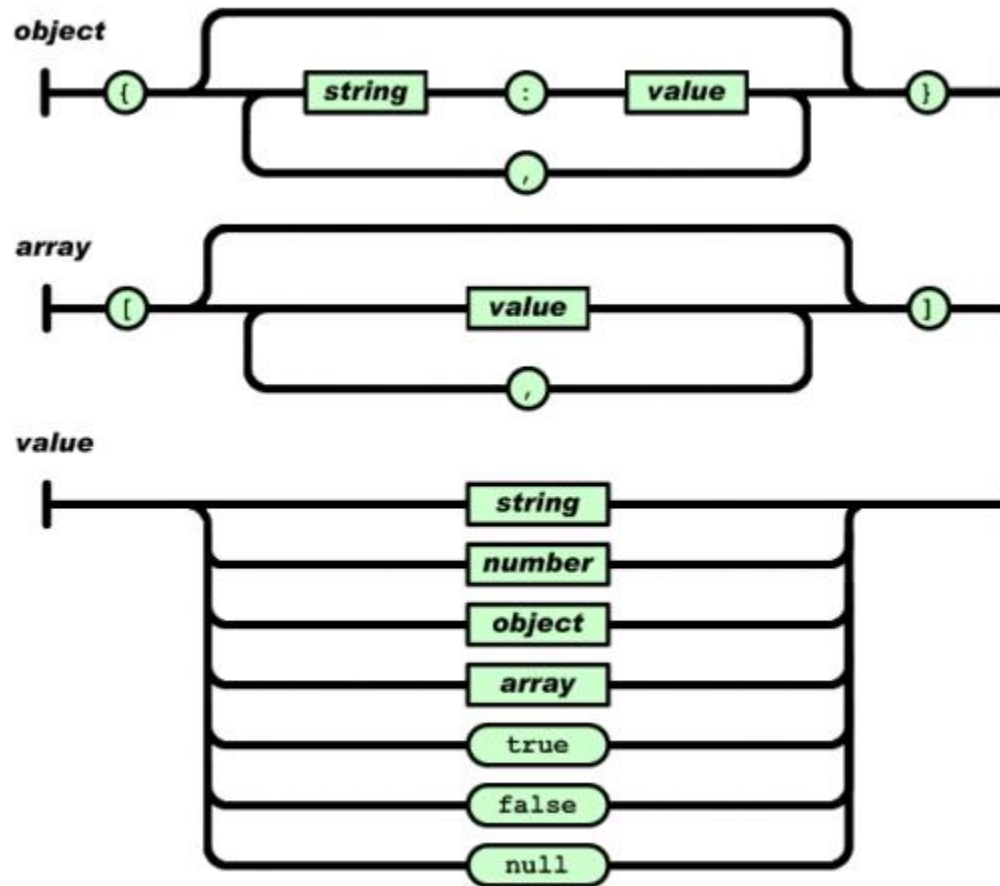
/ Serialise the object to a string in JSON format
-- only properties get serialised */*

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

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

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

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



JSON Data Format

- **JSON** is a very popular **lightweight data format** to transform an object to a **text** form to ease storing and transporting data
- **JSON** class could be used to transform an object to json or transform a json string to an object

Transform an instance of Surah class to a JSON string:

```
const fatiha = {id: 1, name: "الفاتحة",  
englishName: "Al-Fatiha", ayaCount: 7, type: "Meccan"}  
const surahJson = JSON.stringify(fatiha)
```

// Converting a json string to an object

```
const surah = JSON.parse(surahJson)
```



```
{  
  "id": 1,  
  "name": "الفاتحة",  
  "englishName": "Al-Fatiha",  
  "ayaCount": 7,  
  "type": "Meccan"  
}
```

Surah
<ul style="list-style-type: none">id: intname: StringenglishName: StringayaCount: inttype: String

Destructuring Object

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

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

- `const { nestedObjectProp: { identifier } } = expression;` same as `const identifier = expression.nestedObjectProp.identifier;`
- 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}`;  
    }  
}
```

```
const 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;  
    }  
}  
  
const circle = new Circle(3.5);  
  
//Add getArea method to the class at runtime  
Circle.prototype.getArea = function () {  
    return Math.PI * this.radius * 2;  
}  
  
const 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() {  
    const max = Math.max(...this);  
    return max;  
}
```

Here **this** means the array

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

Private Attributes

- Private attributes can only be accessed within the class. They are prefixed with **#**

```
class User {  
  // Random number between 0 and 100  
  #randomPrefix = Math.floor(Math.random() * 100);  
  #id = `${this.#randomPrefix}${new Date().getFullYear()}`;  
  constructor(name) {  
    this.name = name;  
  }  
  get userId() {  
    return this.#id;  
  }  
}
```

```
const user1 = new User("Juha Dahak");  
console.log(user1.userId, user1.name);  
// Accessing a private attribute causes a syntax error  
console.log(user1.#id);
```

Prototypal Inheritance

Prototypal Inheritance

- Prototypal Inheritance (aka Object-Based Inheritance) enables creating an object from another object
 - Instead of creating classes, you **make prototype object**, and then use **Object.create(..)** or **Object.setPrototypeOf(..)** 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

```
const cat = { legs : 4, eyes: 2 };  
const myCat = { name: 'Garfield' };  
Object.setPrototypeOf(myCat, cat);  
// Or const myCat = Object.create(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 (i.e., prototype object)
- Create new instances from that object
 - Resulting object **maintains an explicit** link (**delegation** pointer) to its prototype
 - JavaScript runtime dispatches the correct method or finds 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**

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

//We clone movie 1 and override the episode property

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

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

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

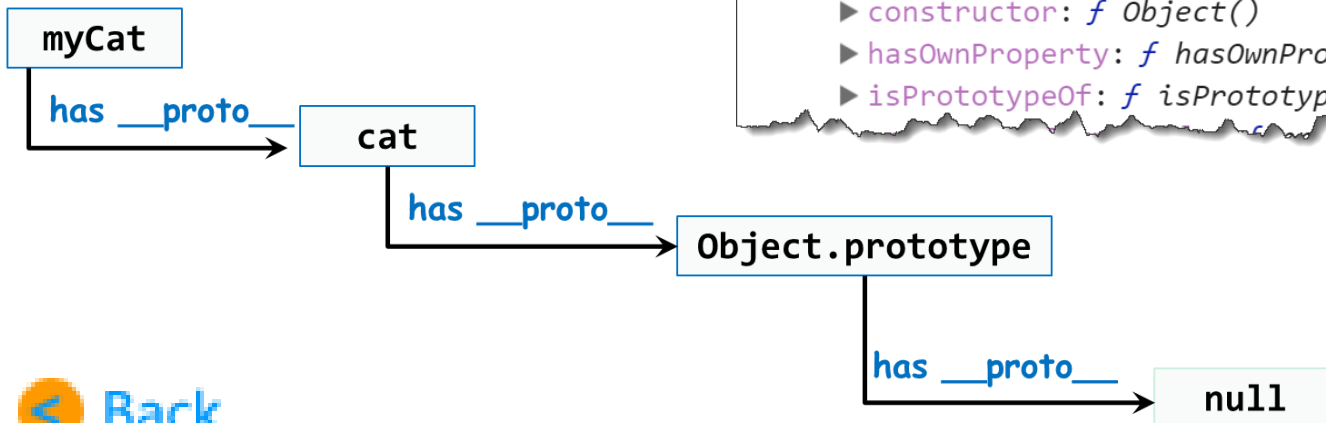
```
console.log('\n');
```

```
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

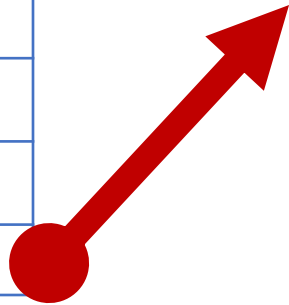
- **Prototype Chain** is the mechanism used for inheritance in JavaScript
 - Establish behavior-sharing between objects using delegation pointers (called Prototype Chain)
- Every object has a an internal **__proto__** property **pointing** to another object
 - `Object.prototype.__proto__` equals null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

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

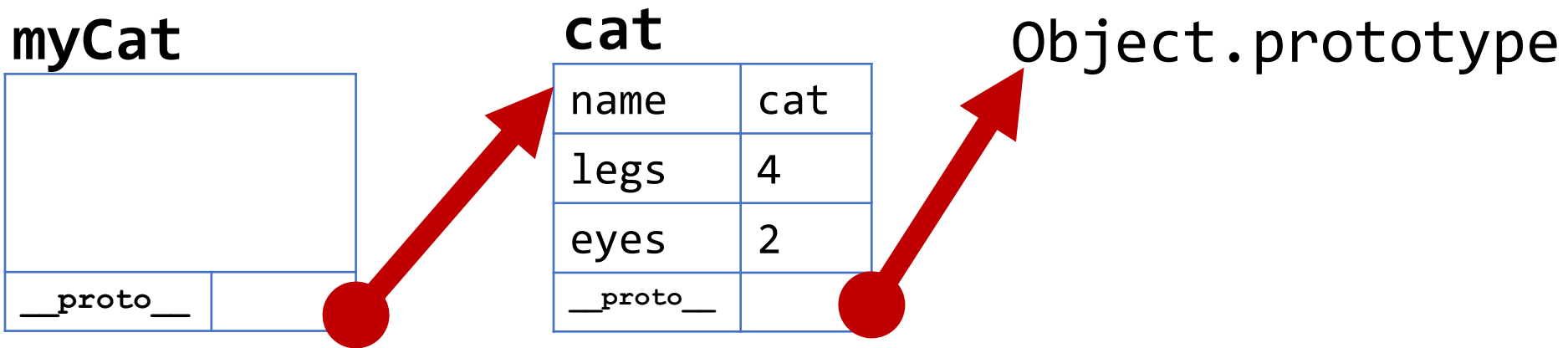
cat

name	cat
legs	4
eyes	2
__proto__	

Object.prototype



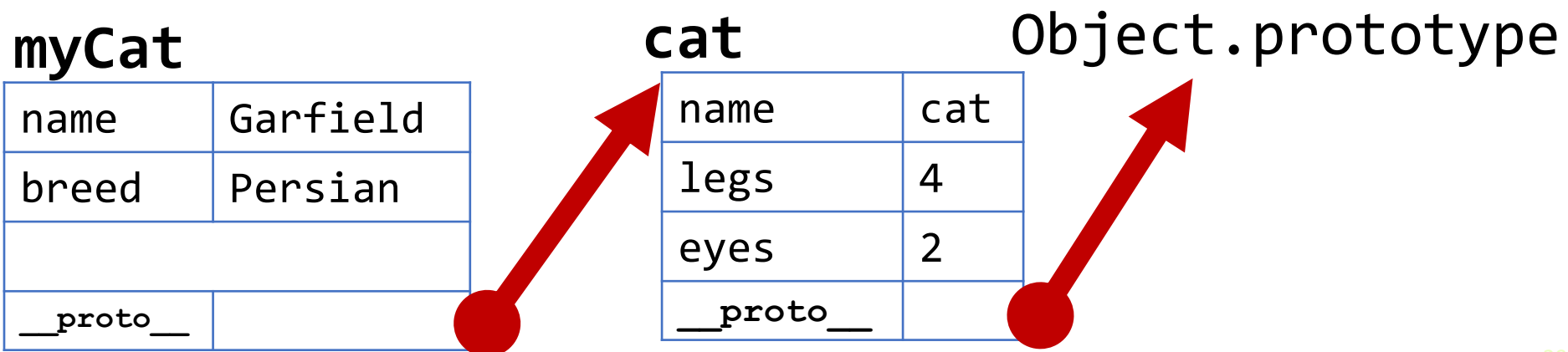
```
const cat = {  
  name : 'cat',  
  legs : 4,  
  eyes : 2  
};  
const myCat = Object.create(cat);
```



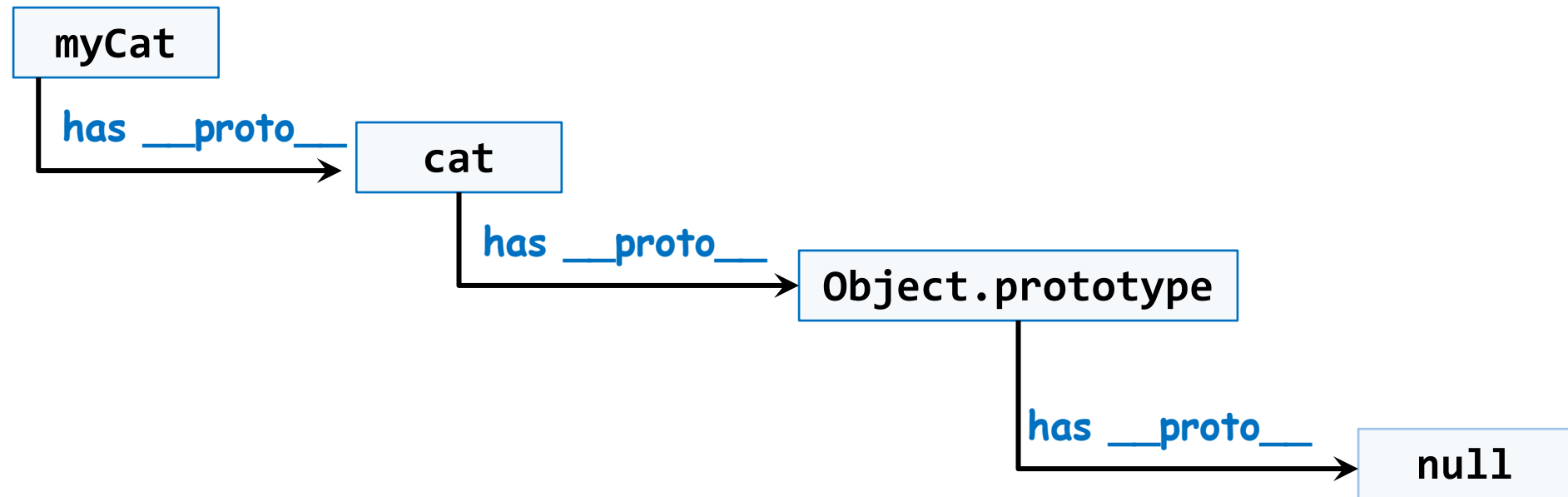
```
const cat = {  
  name : 'cat',  
  legs : 4,  
  eyes : 2  
};
```

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).

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



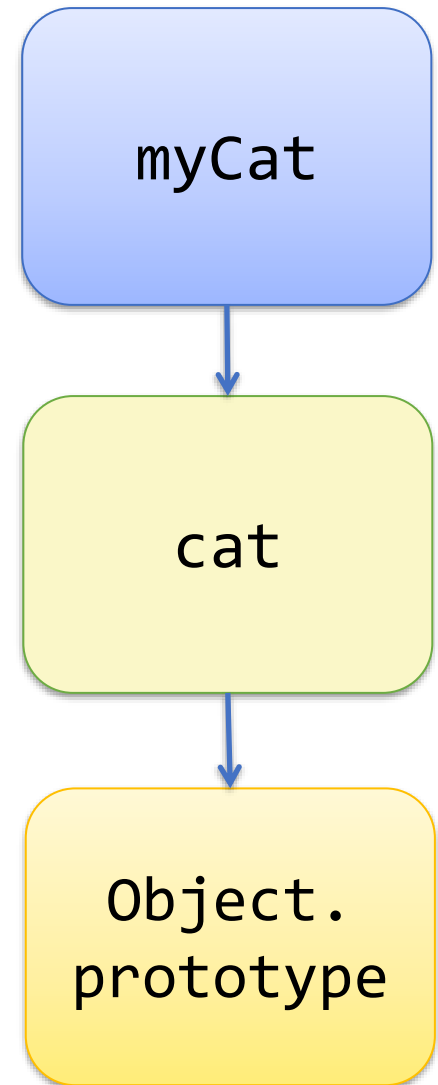
Prototype Chain example



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

Prototype Chain

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



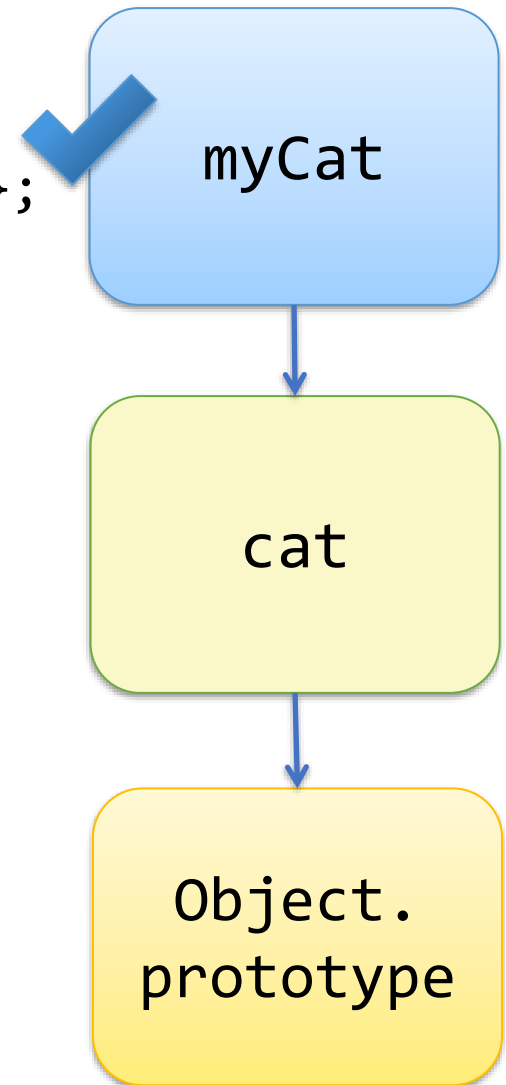
Prototype Chain (lookup myCat.name)

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



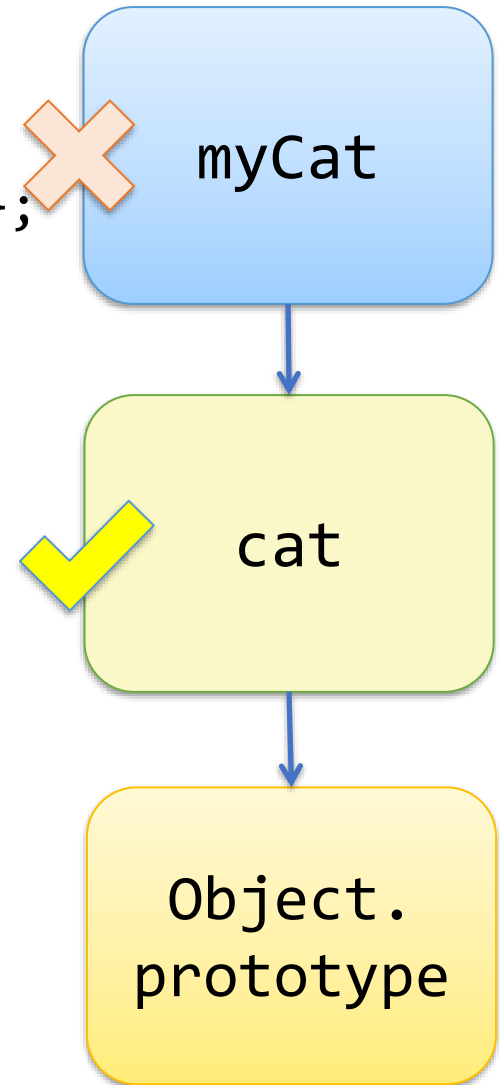
Prototype Chain (lookup `myCat.legs`)

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



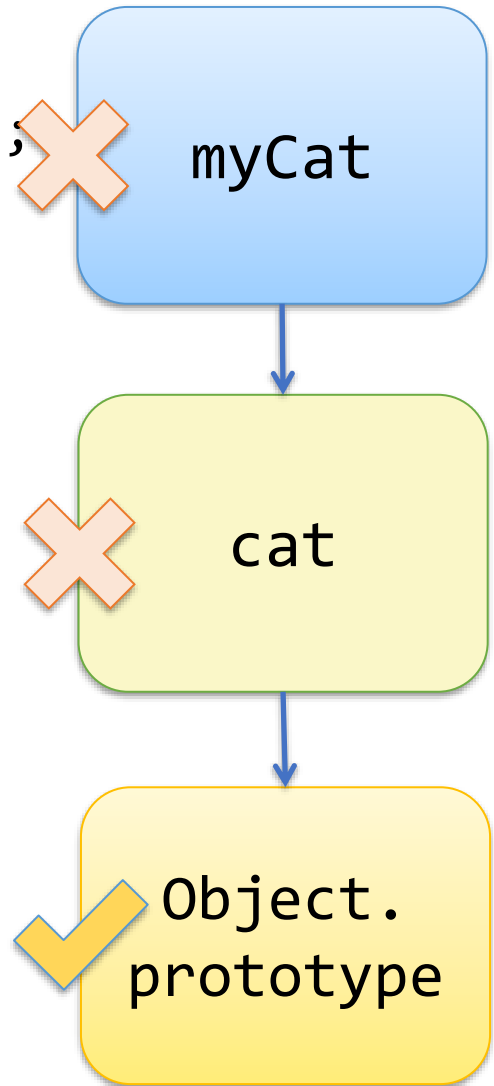
Prototype Chain (lookup myCat.hasOwnProperty)

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

A module = reusable snippets of functionality (functions, classes, objects, variables, constants) that you can include in your application

JavaScript Modules

- JavaScript modules are supported by browsers and Node.js
 - Modules allow reusing code stored in different .js files
 - For Node.js need to add "type": "module" to *packages.json*

- **Export** the items from a module (named export):

```
// lib.js
```

```
export const add = (x, y) => x + y;
```

```
export const multiply = (x, y) => x * y;
```

- **Import** the desired module items in another file:

```
// app.js
```

```
import {add, multiply} from './lib.js';
```

```
add(2, 3);
```

```
multiply(2, 3);
```

named export vs. default export

- JavaScript provides two ways to export items (a variable, a function, a class, an object) from a file: **named** export and **default** export
- Named exports allows several exports per file
 - The name of imports must be the same as the name of exports
- Only one **default (unnamed) export** per file is allowed
 - Specify a name when importing a default module

```
// calculator.js
class Calculator {
  add = (x, y) => x + y;
  subtract = (x, y) => x - y;
}
export default new Calculator();
```

```
// app.js
import calculator from './calculator.js';
```

Module Export and Import

- Alternatively, a single export statement can be used
- **import** is then used to pull items from a module into another script:

```
// lib.js
const PI = 3.1415926;

function sum(...args) {
  log('sum', args);
  return args.reduce((num, tot) => tot + num);
}

function mult(...args) {
  log('mult', args);
  return args.reduce((num, tot) => tot * num);
}

// private function
function log(...msg) {
  console.log(...msg);
}

// A single export statement
export { PI, sum, mult };
```

```
// main.js
//One items can be imported
import { sum } from './lib.js';
console.log( sum(1,2,3,4) );
```

```
//Multiple items can be imported at
one time:
import { sum, mult } from './lib.js';
console.log( sum(1,2,3,4) );
console.log( mult(1,2,3,4) );
```

```
// All public items can be imported by
providing a namespace:
import * as lib from './lib.js';
```

```
console.log( lib.PI );
console.log( lib.add(1,2,3,4) );
```

Old Modules Syntax



- Old Modules syntax used by Node.js

circle.js

```
//Export 2 functions to make them 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');
```

Built-in Modules

- Node.js has a set of built-in modules which you can use without any further installation
 - https://www.w3schools.com/nodejs/ref_modules.asp
- To include a module, use the import statement with the name of the module

```
import path from 'path';  
import fs from 'fs';
```

```
const currentPath = path.resolve();  
console.log(`Files in current path: ${currentPath}`);  
fs.readdir(currentPath, (err, files) => {  
    files.forEach(file => {  
        console.log(file);  
    })  
})
```


Node Package Management (NPM)

- <https://npmjs.com> is a huge npm repository to publish and download JavaScript modules
 - **npm** is used to download packages
 - 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 fs-extra
```

```
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 eBook

<https://exploringjs.com/impatient-js/toc.html>

- Code Camp

<https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/>