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)
- 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 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 = { name: 'Garfield' };
Object.setPrototypeOf(myCat, cat);
```

Object Literal using JSON

Back

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

```
var 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
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 gets 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 St'
    }
};

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

Class-based OOP

Back

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());
```

Prototypal Inheritance

< Back

Prototypal Inheritance

- Prototypal Inheritance (aka 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 **Object.setPrototypeOf**(..) or 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!**

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 right piece of data simply by following a series of delegation pointers 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

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`);
```

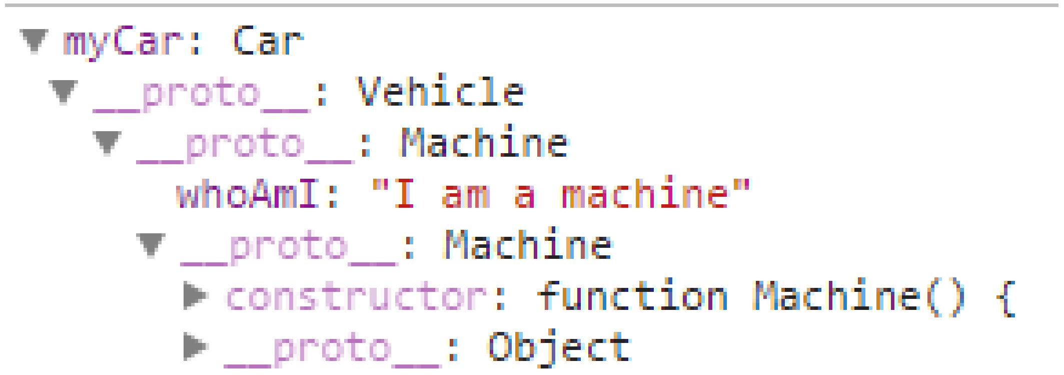
Object.assign() method

- The **Object.assign()** method is used to merge one or more source objects to a target object while **replacing** values of properties with matching names

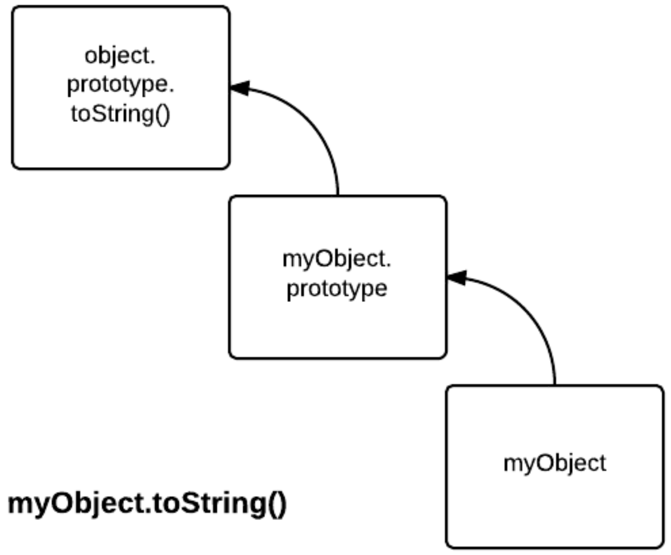
  - Used for cloning => no inheritance

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

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

console.log("movie1.episode: ", movie1.episode); // writes 7
console.log("movie1.episode: ", movie2.episode); // writes 8
```

Prototype Chain



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

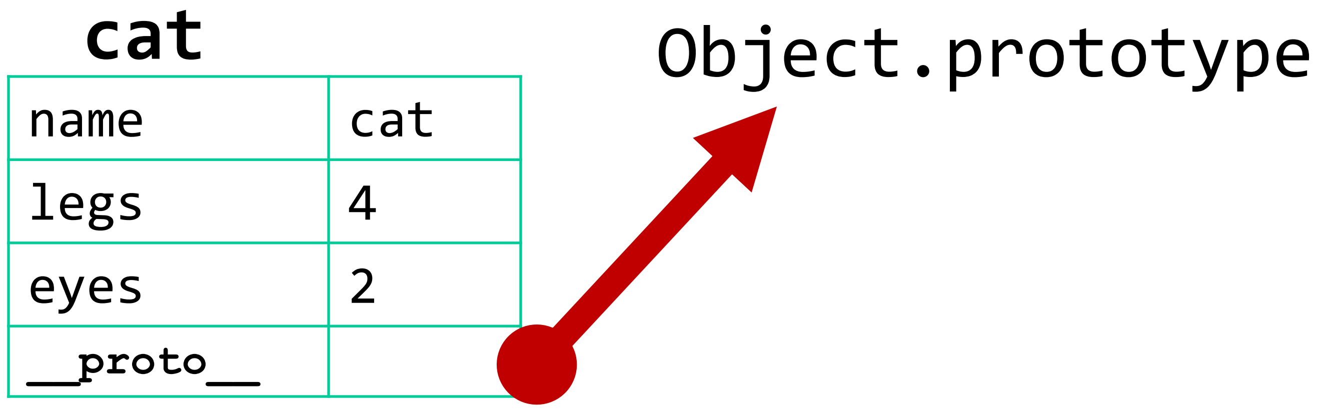


< Back

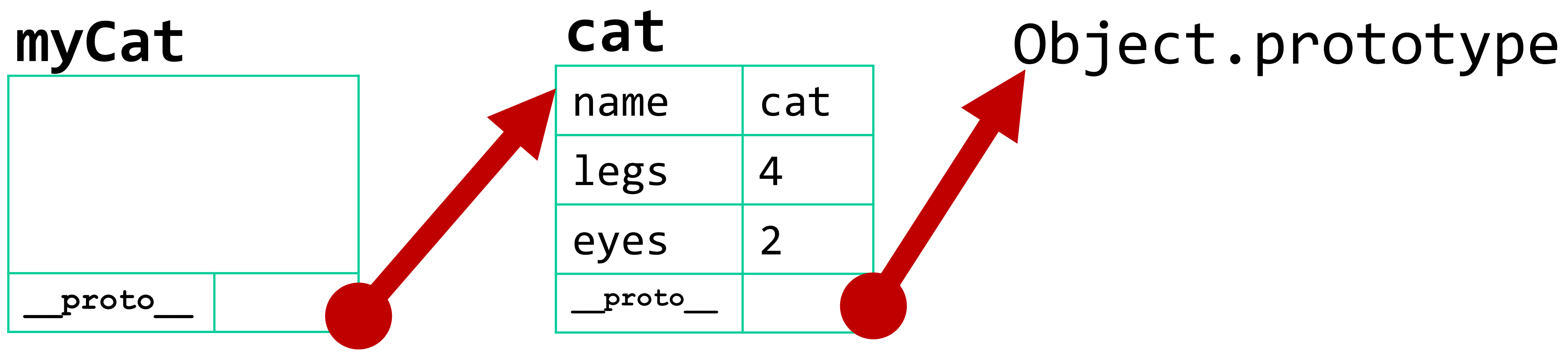
Prototype Chain

- **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
};
```

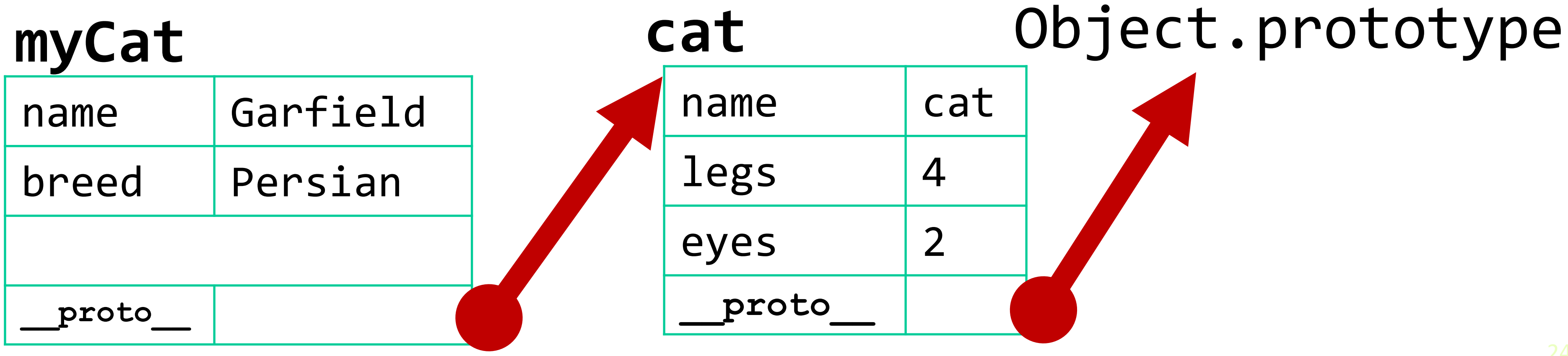


```
let cat = {
      name : 'cat',
      legs : 4,
      eyes : 2
};
let myCat = {};
Object.setPrototypeOf(myCat, cat);
```

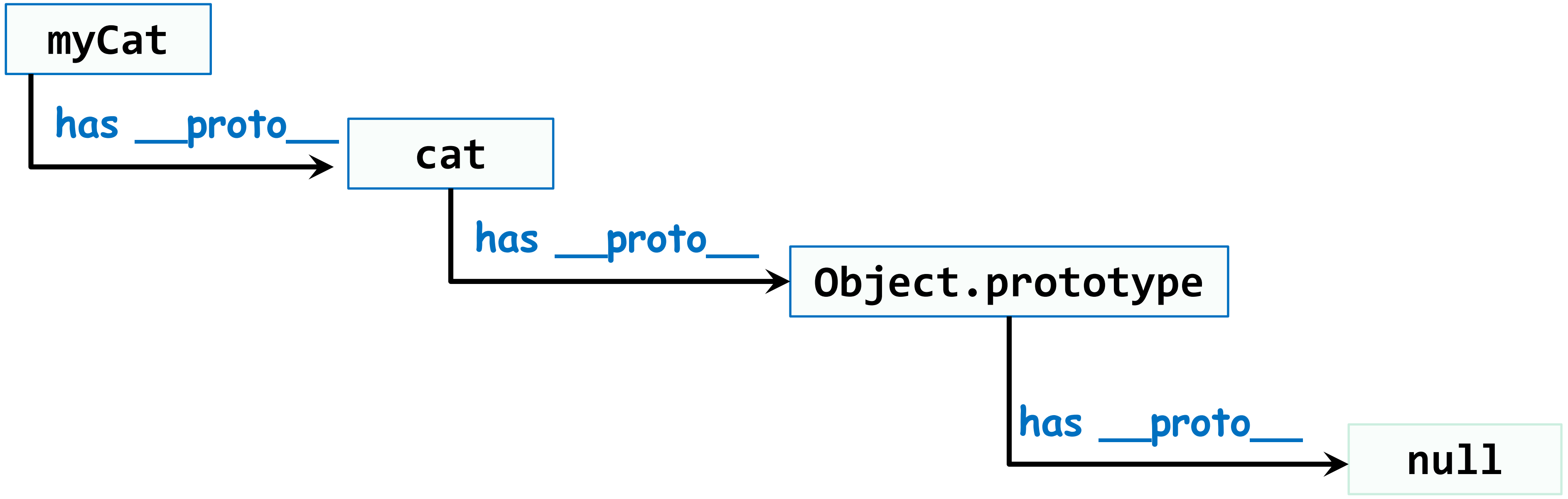


```
let cat = {
        name : 'cat',
        legs : 4,
        eyes : 2
};
let myCat = {};
Object.setPrototypeOf(myCat, cat);
myCat.name  = 'Garfield';
myCat.breed = 'Persian';
```

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).



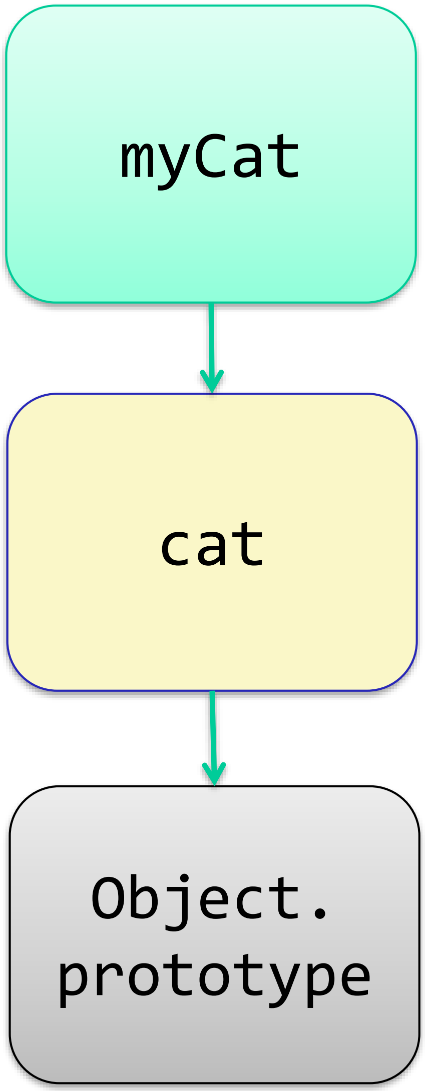
Prototype Chain example



__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.setPrototypeOf(myCat, cat);
myCat.name  = 'Garfield';
myCat.breed = 'Persian';
```



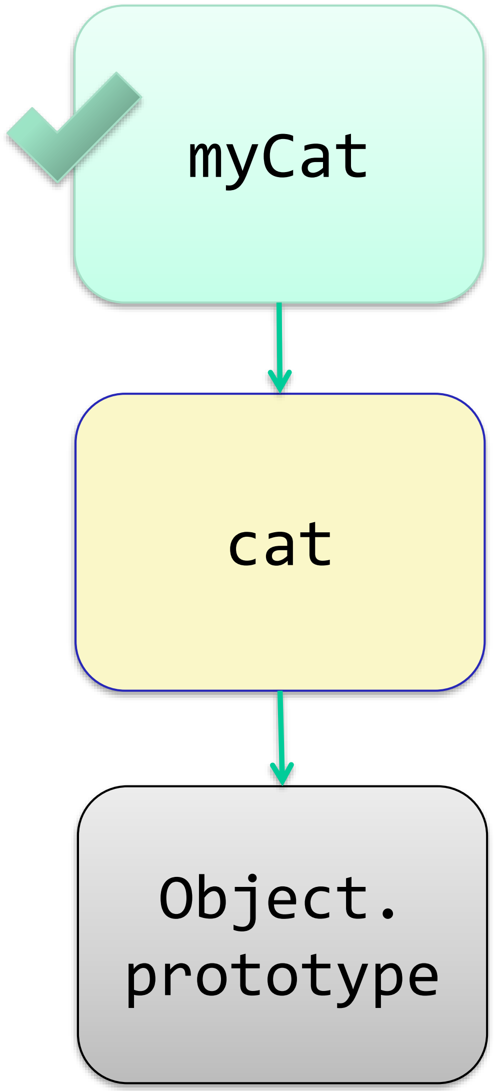
Prototype Chain (lookup myCat.name)

```
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);
```

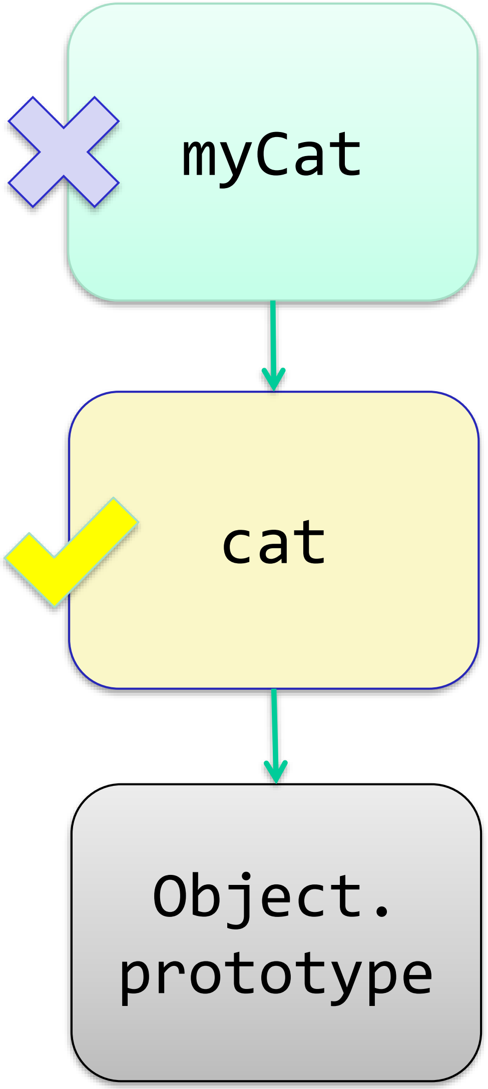
```
console.log(myCat.hasOwnProperty('eyes'));
```



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';
```

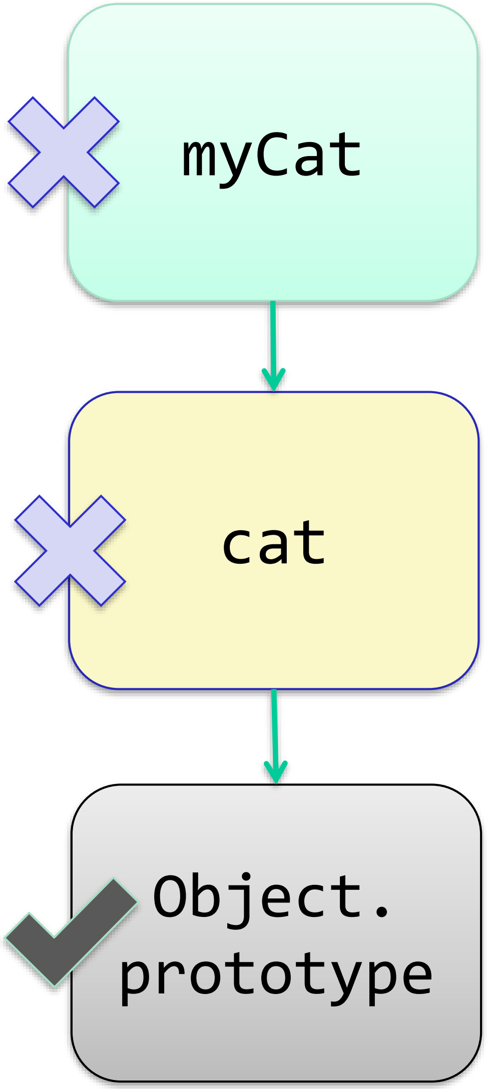
```
 console.log(myCat.name);

 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'));
```



Prototype can be used to extend classes

- **<u>Classes</u>** has a special property called **prototype**
- It can be used to add properties / methods to a class
  - Reflected on all instances of the class
  - Simply reference the **prototype** property on the class before adding the property

See *6.class-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 property to Add Functionality to Build-in Classes

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

```
//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

Modules

Back

CommonJS Modules

- Modules are elegant way of encapsulating and reusing code
- 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
  - Each file decides what to **export** from its module
  - 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();
```

Module Bundling

The problem:

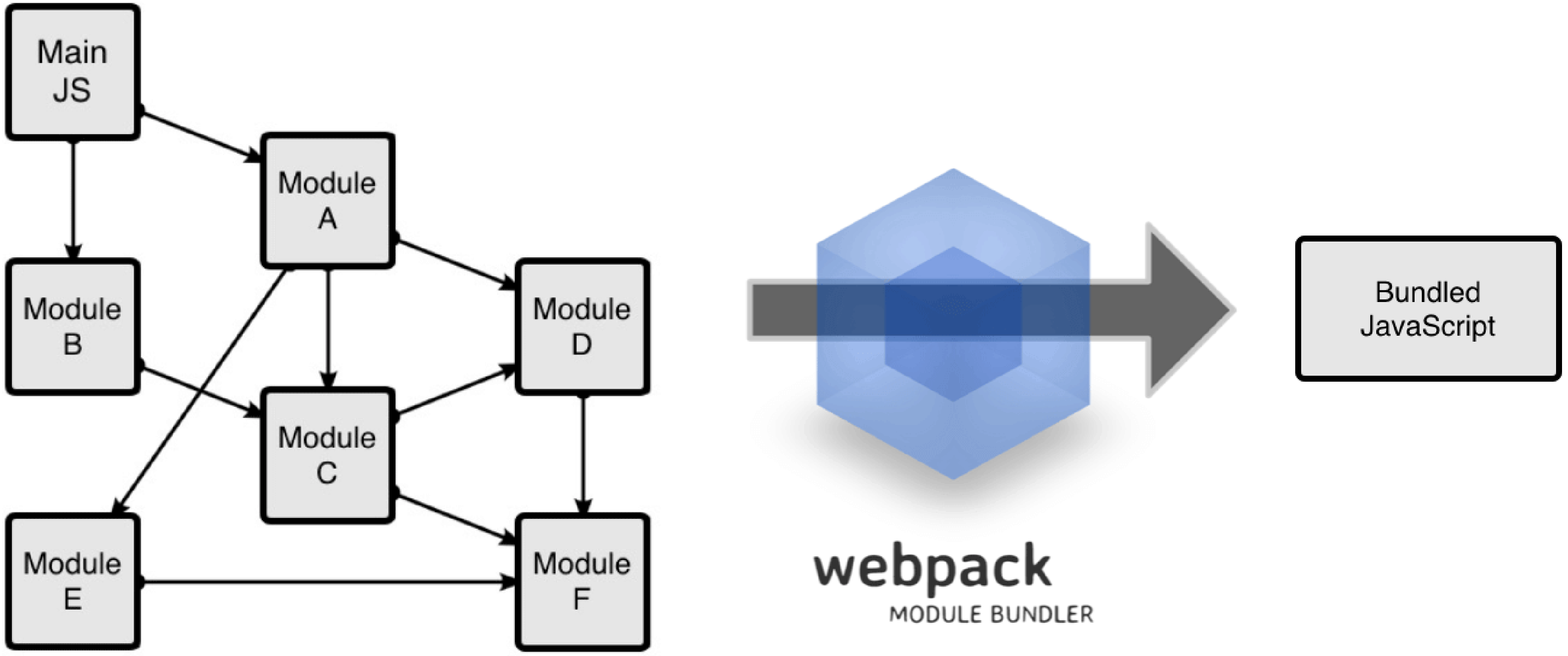
- Code complexity grows as the site gets bigger
- Developers prefer to split up code into different modules
- Deployment requires optimized code in just one or a few HTTP calls

Solution:

- Java Script Module Bundlers such as Browserify and **Webpack**

Webpack

- Webpack Is a module bundler. It takes modules with dependencies and generates static assets representing those modules



Modules with dependencies

Configuration based

- Run **webpack** on the command-line to create bundle.js

```
const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
const path = require('path');

module.exports = {
    entry: './app.js',
    output: {
        filename: 'bundle.js',
        path: path.resolve(__dirname, 'dist')
    },
    plugins: [
        new UglifyJSPlugin()
    ]
}
```

3 main things webpack needs to know

1. the starting point of your application
2. which transformations to make on your code (e.g., uglify)
3. where it should save the new transformed code

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 node-fetch
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

- Learn ES2015

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

- Best JavaScript eBooks

http://exploringjs.com/es6/

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

- More Resources

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