



OOP Using JavaScript

Some of the slides are based on JavaScript course from Telerik
<http://downloads.academy.telerik.com/svn/school-academy/>

Outline

- JavaScript OOP
 - Object Literal using JSON
 - Classical OOP
 - Prototype-oriented OOP
- Inheritance

JavaScript OOP

Properties & Methods

JavaScript OOP

- JavaScript is designed on a simple object-based paradigm
 - An **object** is a collection of properties
- An object property is association between a name and a value
 - A value of property can be either a **variable** (e.g., a number or a string) or a **method** (function)
- Lots of predefined objects available in JS
 - Math, document, window, etc...
- Objects can be created by the developer

JS Objects

- Objects are containers for
 - Properties - Data
 - Methods - Behaviors
- In JS an object is a dynamic collection of properties
 - Every property has a key string that is unique within that 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
 - **Classical OOP**: create a class then instantiate objects from the class
 - **Prototype-oriented OOP**: create objects from other objects
 - Creates new copies of objects from an existing object (the **prototype**)
 - Code reuse done by **cloning**
- e.g, `var fiona = Object.create(shrek);`

Object Literal using JSON

Create an Object Literal using JSON

```
var person = {  
  firstName: 'Samir',  
  lastName: 'Saghir',  
  height: 54,  
  name : function() {  
    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.grade = 8;  
  
joha.toString = function() {  
    return 'Name: ' + this.name + ', Age: '  
        + this.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>

Classical OOP

Defining a Class with Constructors

- A new JavaScript class is defined by creating a function (serving as constructor)
 - Functions play the role of object constructors
 - Create/initiate object by calling the function with the **"new"** keyword
 - Function constructors can take parameters to give instances different state

```
function Student(name, grade)
{
    this.name = name;
    this.grade = grade;
}
var juha = new Student("Juha Nasreddin", 8);
var abbas = new Student("Abbas Ibn Firnas", 9 );
var samir = new Student("Samir Saghir", 6);
```

Class Members

- Use the keyword **this** to attach properties to object
- Property values can be either variables or functions (i.e., methods)
- **this** contains the instance of the object that is initialized with the function constructor

```
function Person( name, age ) {  
  this.name = name;  
  this.age = age;  
  this.sayHello = function() {  
    return "My name is " + this.name +  
      " and I am " + this.age + " years old";  
  }  
}  
var maria = new Person("Maria",18);  
maria.sayHello();
```

Here **this** means the
Person object

this

- **this** is available everywhere where there is JavaScript
 - Yet it has a different meaning
- This can have two different values
 - **Global scope** (i.e. window)
 - When outside a object scope
 - **A concrete object**
 - When using the new operator

Prototype-oriented OOP

Prototypal OOP

- JavaScript is **prototype-oriented** programming language
- Prototypal 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 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!**

Prototypal OOP

- ◆ Create an object template then 'clone it' into another object
 - Create an object from another object!

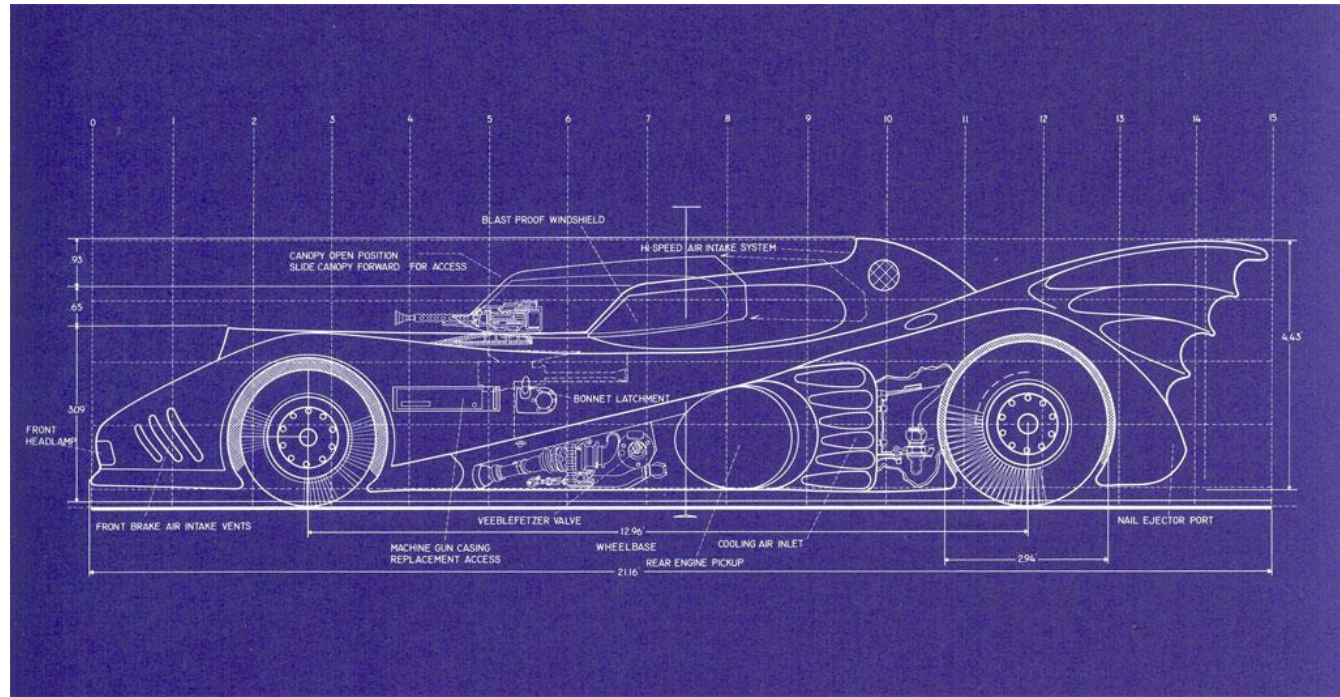
```
var Organism = Object.create(Object.prototype);  
var Animal = Object.create(Organism);  
var Mammal = Object.create(Animal);  
var Dog = Object.create(Mammal);  
var Spot = Object.create(Dog);
```

```
Dog.numLegs = 4;  
Dog.speak = function() {  
    return 'woof, woof!';  
};
```

Demo @ <http://www.youtube.com/watch?v=giJV6bo0LxU>

Inheritance





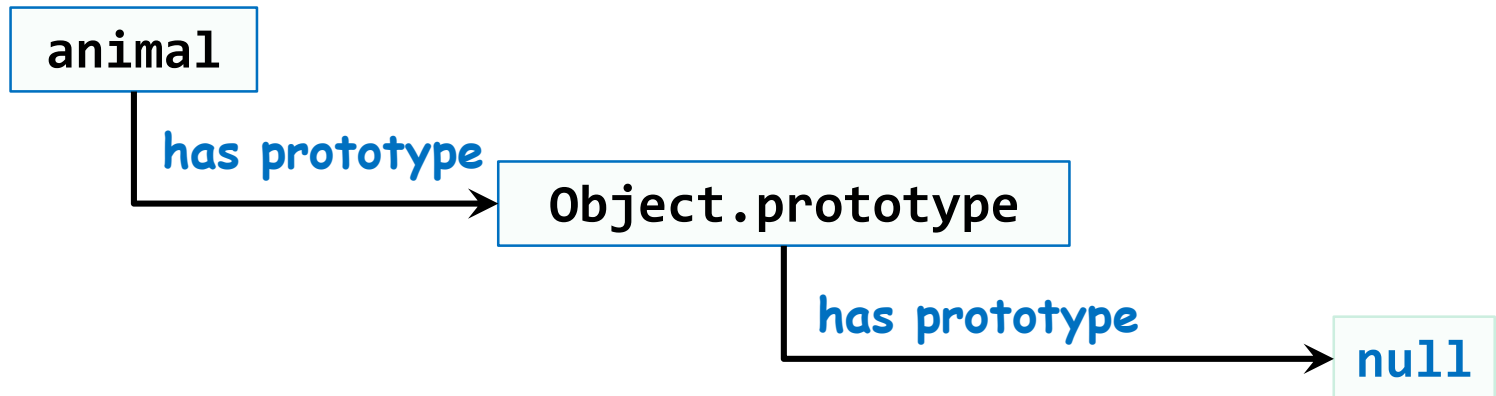
- Every object has a **prototype** property that can be used to:
 - **Extend a class** (i.e., add properties and methods to a class)
 - Implement **inheritance**

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

```
var 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 classes
 - That is reflected on all instances of the class
 - Simply reference the keyword **prototype** on the object before adding the custom property

See [prototype-object.html](#)

```
function Circle() {  
}  
Circle.prototype.pi = 3.14159;  
Circle.prototype.radius = 5;  
Circle.prototype.calculateArea = function () {  
    return this.pi * this.radius * 2;  
}  
var circle = new Circle();  
var area = circle.calculateArea();  
alert(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(){
```

```
  var sum = 0;
```

```
  for(var i in this){
```

```
    if(typeof this[i] === "number"){
```

```
      sum += this[i];
```

```
    }
```

```
  }
```

```
  return sum;
```

```
}
```

```
var numbers = [1,2,3,4,5];
```

```
console.log(numbers.sum()); //logs 15
```

Attaching a method
to the Array class

Here **this** means
the array

Prototype can be use for Inheritance

- ◆ Inheritance is a reuse mechanism
 - E.g., Student class inherits Person class properties and methods.
- ◆ In JavaScript Inheritance is achieved by setting the **prototype of the derived type to an instance of the super type**

```
function Person(fname, lname) {}  
function Student(fname, lname, gpa) {}  
Student.prototype = new Person();
```

- ◆ Now all instances of type Student are also of type Person and have Person functionality

```
var student = new Student("Abbas", "Ibn Firnas", 3.9);
```


Inheritance Example

- To inherit from a class in JavaScript you should set the **prototype** object of the subclass to an instance of the superclass:

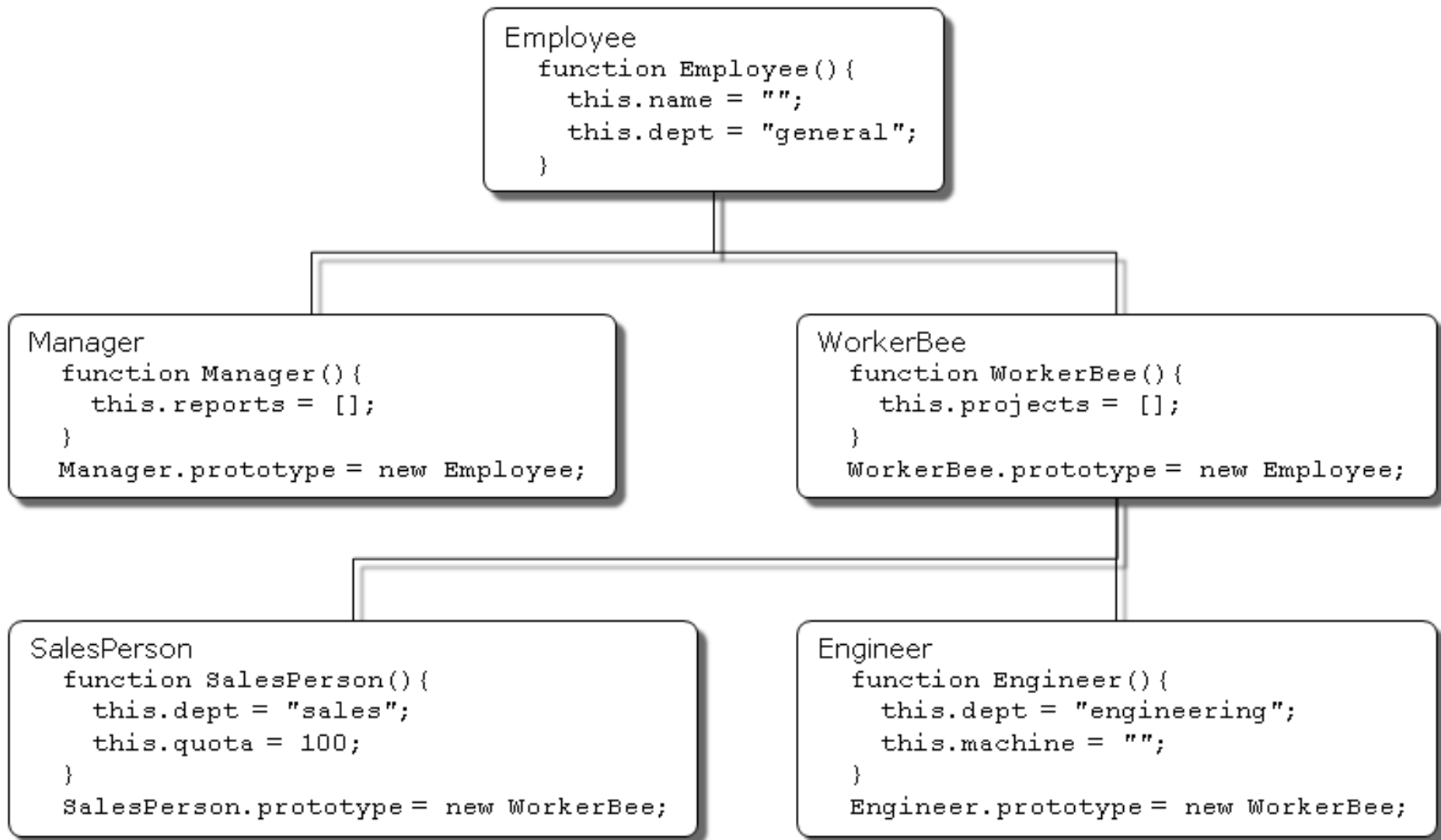
inheritance.html

```
function Person(name) {  
  this.name = name;  
  this.talk = function () {  
    alert("Hi! I am " + this.name);  
  }  
}
```

```
function Student(name, grade) {  
  this.name = name;  
  this.grade = grade;  
}
```

```
Student.prototype = new Person();
```

This way we say that the Student class will have all the functionality of the Person class



- Further details @ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details of the Object Model](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model)

Reusing the parent Constructor

- To reuse the parent constructor you can use **apply** or **call** function

```
function Person(name, age){  
    this.name = name; this.age = age;  
}  
function Student(name, age, grade){  
    Person.apply(this, arguments); //Person.call(this, name, age);  
    this.grade = grade;  
}  
Student.prototype = new Person();  
Student.prototype.constructor = Student;
```

Polymorphism in JavaScript

- **Polymorphism** = ability to take more than one form (objects have more than one type)
 - A class can be used through its parent interface
 - A child class may override some of the behavior of the parent class
 - Refer to ***polymorphism.html*** example for further details

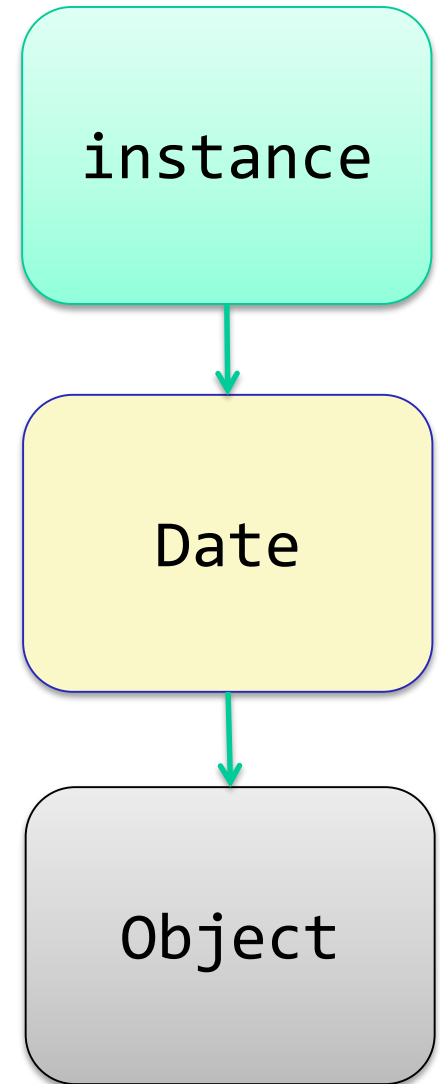
```
Student.prototype = new Person();
Teacher.prototype = new Person();
var people = [new Teacher("Joha", "Math"),
               new Student("Samir", 3.5),
               new Person("Salaheddine")];
for(var index in people) {
    people[index].saySalam();
}
```

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

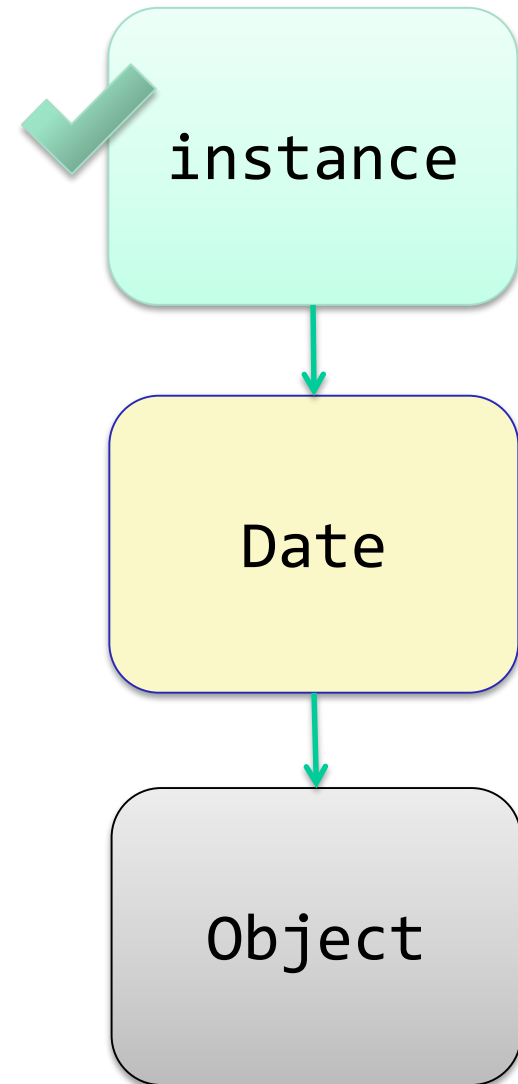
Property lookup chain

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



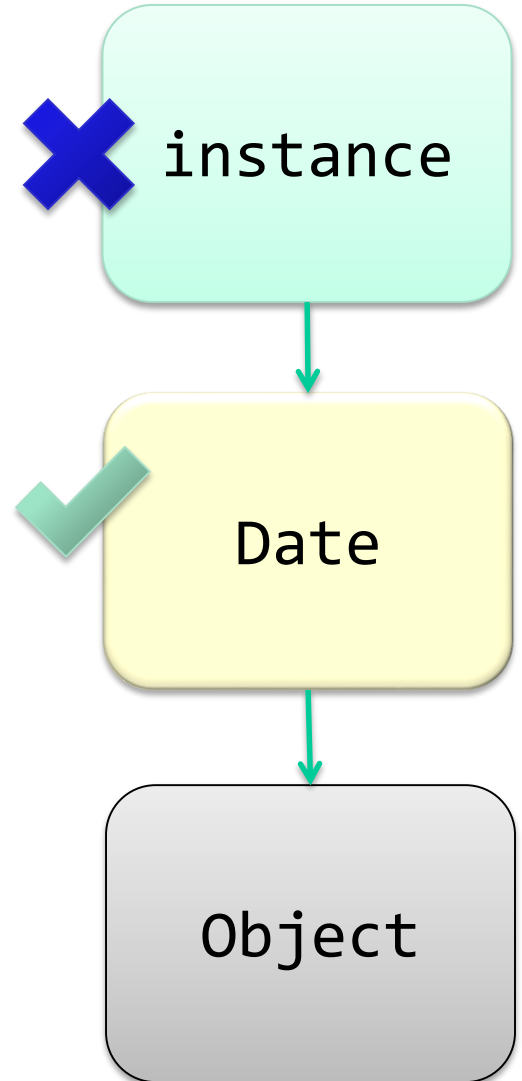
Property lookup chain (look up instance.foo)

```
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
```



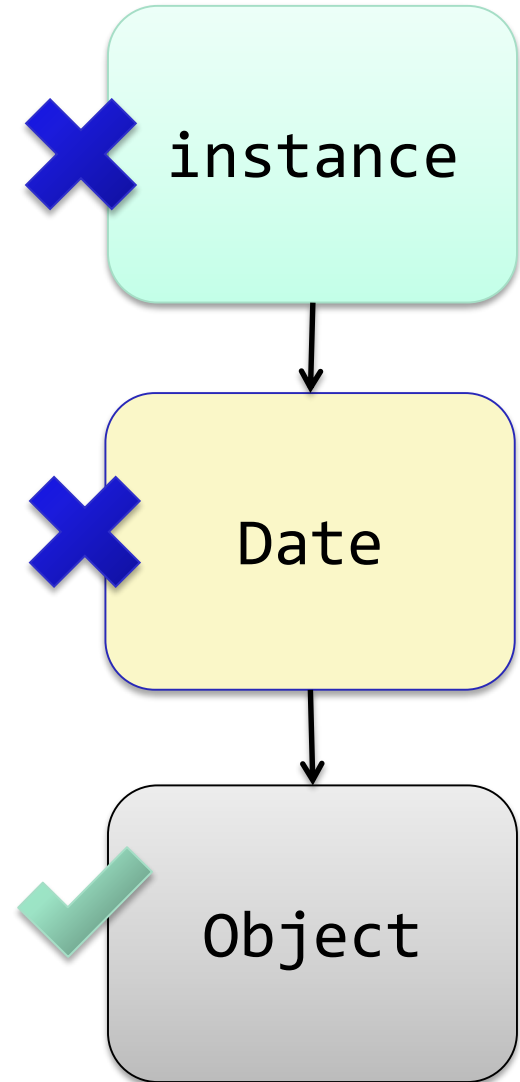
Property lookup chain (lookup instance.getTime)

```
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");
```



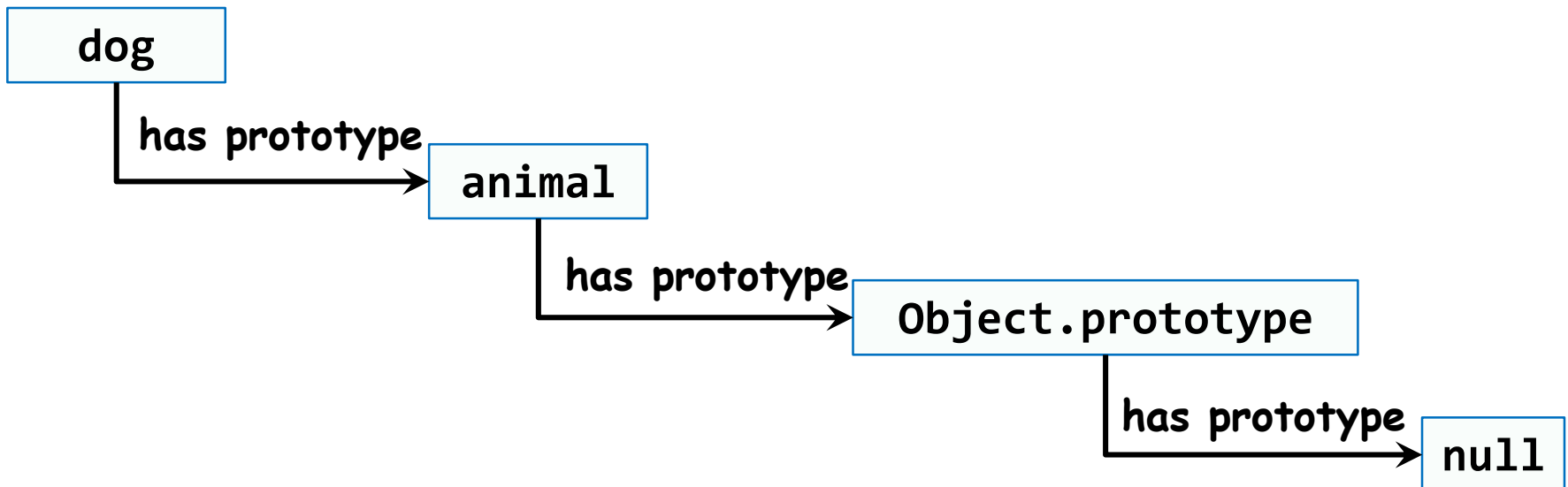
Setting the Prototype of an Object

- Every JavaScript object has a property **__proto__**
 - It can be used to access/change the prototype
- When setting a prototype to an object, the object has access to all the properties of the prototype
 - Through the prototype chain
 - This is actually called Prototypal Inheritance

```
var animal = {  
  makeNoise: function(){  
    console.log('The ' + this.type +  
                ' makes noise "' + this.noise + '"');  
  }  
};  
var dog = { type: 'dog', noise: 'Djaf' };  
dog.__proto__ = animal;  
dog.makeNoise(); //makeNoise() is from the prototype(animal)
```

Setting the Prototype of an Object

```
var animal = {  
  makeNoise: function(){  
    console.log('The ' + this.type +  
      ' makes noise ' + this.noise + '');  
  }  
};  
var dog = { type: 'dog', noise: 'Djaf' };  
dog.__proto__ = animal;  
dog.makeNoise(); // makeNoise() is from the prototype(animal)
```



Setting Prototypes with Object.create()

- ES5 introduces a method for setting the prototype of an object
 - Object.create(object)

```
var animal = {  
  /* properties and methods */  
};  
var dog = Object.create(animal);  
dog.type = 'dog';  
dog.noise = 'djaf';
```

Same as

```
var animal = {  
  /* properties and methods */  
};  
var dog = {  
  type: 'dog',  
  noise: 'djaf'  
};  
dog.__proto__ = animal
```