JavaScript: Objects, Methods, Prototypes

Lecture 24
What is an Object?

- **Property**: a key/value pair (aka “name”/value)
- **Object**: a partial map of properties
  - Keys must be unique
- Creating an object, literal notation
  ```javascript
  ```
- To access/modify an object's properties:
  ```javascript
  myCar.make = "Ford";  //cf. Ruby
  myCar["year"] = 2006;
  var str = "ate";
  myCar["pl" + str] == "NKR463";
  ```
Object Properties

myCar

- make: "Ford"
- year: 2006
- plate: "NKR463"
Arrays vs Associative Arrays

- Arrays: store elements in a fixed order and can be accessed by their index.

Example:
- Array: [4, "hi", 3.14, true, false]
- Associative Array: {
  "age": 4,
  "greeting": "hi",
  "doors": true,
  "pi": 3.14
}

- Arrays can be more efficient for simple numerical operations.
- Associative Arrays are better for complex key-value scenarios.
Dynamic Size, Just Like Arrays

- Objects can grow
  
  ```javascript
  myCar.state = "OH"; // 4 properties
  var myBus = {};
  myBus.driver = true; // adds a prop
  myBus.windows = [2, 2, 2, 2];
  ```

- Objects can shrink

  ```javascript
  delete myCar.plate;
  // myCar is now {make: "Ford",
  // year: 2006, state: "OH"}
  ```
Object Properties

myCar

- make: "Ford"
- year: 2006
- plate: "NKR463"
Object Properties

```javascript
myCar = {};

myCar.make = "Ford";
myCar.year = 2006;
myCar.plate = "NKR463";
myCar.state = "OH";
```
Object Properties

myCar

delete myCar.plate;

make: "Ford"
year: 2006
state: "OH"
Testing Presence of Key

- Boolean operator: \textit{in} \newline \textit{propertyName in object}

- Evaluates to true iff object has the indicated property key
  
  "make" in myCar //=> true

  "speedometer" in myCar //=> false

  "OH" in myCar //=> false

- Property names are strings
Iterating Over Properties

- Iterate using `for...in` syntax
  ```javascript
  for (property in object) {
    ...object[property]...
  }
  ```

- Notice `[]` to access each property
  ```javascript
  for (p in myCar) {
    document.write(p +"": " + myCar[p]);
  }
  ```
Methods

- The value of a property can be:
  - A primitive (boolean, number, string, null...)
  - A reference (object, array, function)
    ```javascript
    var temp = function(sound) {
        play(sound);
        return 0;
    }
    myCar.honk = temp;
    ```

- More succinctly:
  ```javascript
  myCar.honk = function(sound) {
      play(sound);
      return 0;
  }
  ```
Example: Method

```javascript
var myCar = {
    make: "Acura",
    year: 1996,
    plate: "NKR462",
    honk: function(sound) {
        play(sound);
        return 0;
    }
};
```
Object Properties

```c
myCar = {
    make: "Acura",
    year: 1996,
    plate: "NKR462",
    honk: play(sound);
    return 0;
};
```
Keyword “this” in Functions

- Recall distinguished formal parameter
  \[ x.f(y, z); \ \text{//} x \text{ is the distinguished argmt.} \]
- Inside a function, keyword “this”
  ```javascript
  function report() {
    return this.plate + this.year;
  }
  ```
- At run-time, “this” is set to the distinguished argument of invocation
  ```javascript
  myCar = {plate: "NKR462", year: 1996};
  yourCar = {plate: 340, year: 2013};
  myCar.register = report;
  yourCar.info = report;
  myCar.register(); \ //=> "NKR4621996"
  yourCar.info(); \ //=> 2353
  ```
Object Properties

```
return this.plate + this.year;
```

```
myCar

plate "NKR462"
year 1996
register
report

yourCar

plate 340
year 2013
info
```
Constructors

- Any function can be a constructor
- When calling a function with “new”:
  1. Make a brand new (empty) object
  2. Call the function, with the new object as the distinguished parameter
  3. Implicitly return the new object to caller
- A “constructor” often adds properties to the new object simply by assigning them
  ```javascript
  function Dog(name) {
      this.name = name;  // adds 1 property
      // no explicit return
  }
  var furBall = new Dog("Rex");
  ```
- Naming convention: Functions intended to be constructors are capitalized
Example

```javascript
function Circle(x, y, radius) {
    this.centerX = x;
    this.centerY = y;
    this.radius = radius;
    this.area = function() {
        return Math.PI * this.radius * this.radius;
    }
}
var c = new Circle(10, 12, 2.45);
```
Creating a Circle Object

```javascript
var c = new Circle(10, 12, 2.45);

this.centerX = x;
this.centerY = y;
... Etc ...
```
Creating a Circle Object

```javascript
var c = new Circle(10, 12, 2.45);
```

Circle

```javascript
this.centerX = x;
this.centerY = y;
... Etc ...
```
Creating a Circle Object

```javascript
var c = new Circle(10, 12, 2.45);
```

```javascript
Circle

this.centerX = x;
this.centerY = y;
... Etc ...

return Math.PI * this.radius * this.radius
```
Creating a Circle Object

```javascript
var c = new Circle(10, 12, 2.45);
```

```javascript
Circle
   this.centerX = x;
   this.centerY = y;
   ... Etc ...

   return Math.PI * this.radius * this.radius
```
Creating a Circle Object

```javascript
var c = new Circle(10, 12, 2.45);

Circle

centerX
10

centerY
12

radius
2.45

area

return Math.PI * this.radius * this.radius
```

... Etc ...

Prototypes

- Every object has a *prototype*
  - A hidden, indirect property ([[Prototype]])

- What is a prototype?
  - Just another object! Like any other!

- When accessing a property (*i.e.* `obj.p`)
  - First look for `p` in `obj`
  - If not found, look for `p` in `obj`'s prototype
  - If not found, look for `p` in that object's prototype!
  - And so on, until reaching the basic system object
Prototype Chaining

```
greeting: "hi"
greeting: doors:
 doors: pi:
 pi: 3.14
age: 4
toString: hasOwnProperty: push: pop: etc...
```

```
0: true
1: true
2: false
```
Class-Based Inheritance

- Interfaces
- Classes
- Objects

Relationships:
- Extends
- Implements
- Instantiates
Example

- Consider two objects
  ```javascript
  var dog = {name: "Rex", age: 3};
  var pet = {color: "blue"};
  ```

- Assume `pet` is `dog`'s prototype
  ```javascript
  // dog.name is "Rex"
  // dog.color is "blue" (follow chain)
  pet.color = "brown";
  // dog.color is still "brown" (prop changed)
  dog.color = "green";
  // pet.color is still "brown" (hiding)
  ```
Delegation to Prototype

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Rex&quot;</td>
<td>3</td>
<td>&quot;green&quot;</td>
</tr>
<tr>
<td>&quot;brown&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Diagram:
- Dog:
  - Name: "Rex"
  - Age: 3
  - Color: "green"

- Pet:
  - Color: "brown"
Prototypes Are Dynamic Too

- Prototypes can add/remove properties
- Changes are felt by all children

```javascript
// dog is {name: "Rex", age: 3}
// dog.mood & pet.mood are undefined
pet.mood = "happy"; // add to pet
// dog.mood is now "happy" too

pet.bark = function() {
    return this.name + " is " + this.mood;
}

dog.bark(); //=> "Rex is happy"
pet.bark(); //=> "undefined is happy"
```
Delegation to Prototype

dog

name
"Rex"
age
3

pet

color
"brown"
mood
"happy"
bark

return this.name + " is " + this.mood;
dog.bark();
pet.bark();
Connecting Objects & Prototypes

- How does an object get a prototype?
  ```javascript
  var c = new Circle();
  ```
- Answer
  1. Every function has a prototype `property`
     - Do not confuse with hidden `[[Prototype]]`!
  2. Object's prototype `link`—`[[Prototype]]`—is set to the function's prototype `property`
- When a function `Foo` is used as a constructor, *i.e.* `new Foo()`, the value of `Foo`'s prototype property is the prototype object of the created object
Prototypes And Constructors

```javascript
this.centerX = x;
this.centerY = y;
... Etc ...
```
Prototypes And Constructors

```java
Circle c = new Circle();
```

```java
this.centerX = x;
this.centerY = y; ...
... Etc ...
```
Prototypes And Constructors

c = new Circle()

Circle

prototype

this.centerX = x;
this.centerY = y;
... Etc ...

constructor

area
Prototypes And Constructors

```java
Circle c = new Circle();
```

```java
this.centerX = x;
this.centerY = y;
... Etc ...
```
Idiom: Methods in Prototype

```javascript
function Dog(n, a) {
    this.name = n;
    this.age = a;
}

var canine = {
    bark: function (sound) {
        return this.name + "says" + sound;
    }
};

Dog.prototype = canine;
```
function Dog(n, a) {
    this.name = n;
    this.age = a;
}

var canine = {
    bark: function (sound) {
        return this.name + "says" + sound;
    }
};

Dog.prototype = canine;
Idiom: Methods in Prototype

```javascript
function Dog(n, a) {
    this.name = n;
    this.age = a;
}

Dog.prototype = {
    bark: function (sound) {
        return this.name + " says " + sound;
    }
};

// set prototype to new anonymous object
```
Idiom: Methods in Prototype

```javascript
function Dog(n, a) {
    this.name = n;
    this.age = a;
}

Dog.prototype.bark = function (sound) {
    return this.name + " says " + sound;
}

// better: extend existing prototype
```
Methods in Prototype

```
r = new Dog()  
```
Idiom: Classical Inheritance

function Animal() { ... };
function Dog() { ... };

Dog.prototype = new Animal();
    // create prototype for future dogs

Dog.prototype.constructor = Dog;
    // set prototype's constructor
    // properly (ie should point to Dog())
Setting up Prototype Chains

```
new Dog()

name  "Rex"

new Animal()

constructor

Dog

Animal

prototype

constructor

prototype
```
Summary

- Objects as associative arrays
  - Partial maps from *keys* to *values*
  - Can dynamically add/remove properties
  - Can iterate over properties
- Method = function-valued property
  - Keyword this for distinguished parameter
- Constructor = any function
- Prototypes are "parent" objects
  - Delegation up the chain of prototypes
  - Prototype is determined by constructor
  - Prototypes can be modified