JavaScript: Objects, Methods, Prototypes

Lecture 25
What is an Object?

- **Property**: a key/value pair
  - aka name/value pair
- **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;
  let str = "ate";
  let pl = "pl" + str;
  myCar[pl] == "NKR463"; //=> true
  ```
Object Properties

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

myCar
Arrays vs Associative Arrays

- Arrays:
  - Indices: 0, 1, 2, 3
  - Values: 4, "hi", 3.14, true, false, 0, 1, 2

- Associative Arrays:
  - Keys: age, greeting, doors, pi
  - Values: 4, "hi", 3.14, true, false
Dynamic Size, Just Like Arrays

- Objects can grow
  ```javascript
  myCar.state = "OH"; // 4 properties
  let myBus = {}; // adds a prop
  myBus.driver = true;
  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 = { make: "Ford", year: 2006, plate: "NKR463", state: "OH"; }
myCar.state = "OH";
```
Object Properties

```javascript
myCar

delete myCar.plate;
```

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

- **Boolean operator:** `in`

  \[
  \text{propertyName in object}
  \]

- 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) {
      ...
  }
  ```

- Notice `[ ]` to access each property
  ```javascript
  for (let p in myCar) {
      document.write(`\$\{p\}: \$\{myCar[p]\}\`);
  }
  ```

- Loop over iterable (eg array) with *for...of*
  ```javascript
  for (let elt of roster) {
      document.write(`\$\{elt\}\`);
  }
  ```
Destructuring Assignment

- Objects can have many properties, and many levels of nesting
  ```javascript
  const result = someGiantObject();
  // only want 2 of result's properties
  report(result.car);
  combine(result.car, result.bus);
  ```

- Alternative: destructuring assignment
  ```javascript
  let {car, bus} = someGiantObject();
  report(car);
  combine(car, bus);
  let {car: c, bus: b} = someGiantObject();
  combine(c, b);
  ```
  - Eliminates unneeded result variable
  - Simplifies access to properties of interest
Methods

- The value of a property can be:
  - A primitive (boolean, number, string, null...)
  - A reference (object, array, function)

```javascript
let temp = function(sound) {
    play(sound);
    return 0;
}
myCar.honk = temp;
```

- More succinctly:

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

```javascript
let myCar = {
    make: "Acura",
    year: 1996,
    plate: "NKR462",
    honk: function(sound) {
        play(sound);
        return 0;
    }
};
```
Example: Method (with Sugar)

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

myCar

make: "Acura"
year: 1996
plate: "NKR462"

honk

play(sound);
return 0;
Keyword “this” in Functions

- Recall distinguished formal parameter
  ```javascript
  x.f(y, z); // x is the distinguished argmt.
  ```
- Inside a function, keyword “this”
  ```javascript
  function report() {
    return this.plate + this.year;
  }
  ```
- At run-time, “this” is the distinguished argument of the invocation
  ```javascript
  myCar = { plate: "NKR462", year: 1996 };
  yourCar = { plate: 340, year: 2013 };
  myCar.register = report;
  yourCar.info = report;
  myCar.register(); // => "NKR4621996"
  yourCar.info();  // => 2353
- Aside: arrow functions work differently
  - Do not have their own this, use enclosing lexical scope
Object Properties

myCar

plate  "NKR462"

year  1996

register

report

yourCar

plate  340

year  2013

info

```
return this.plate + this.year;
```
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
  }
  let 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;
    }
}
let c = new Circle(10, 12, 2.45);
```
Creating a Circle Object

```javascript
let c = new Circle(10, 12, 2.45);
this.centerX = x;
this.centerY = y;
... Etc ...
```
Creating a Circle Object

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

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

let c = new Circle(10, 12, 2.45);

circle

- centerX: 10
- centerY: 12
- radius: 2.45
- area: return Math.PI * this.radius * this.radius

... Etc ...

Circle
Creating a Circle Object

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

```javascript
Circle

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

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

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

```javascript
let c = new Circle(10, 12, 2.45);
this.centerX = x;
this.centerY = y;
... Etc ...

return Math.PI * this.radius * this.radius
```
Creating Many Circle Objects

```javascript
for (let i = 0; i < 1000; i++) {
    new Circle(0, 0, i);
}
```

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

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

How many of these?
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
Class-Based Inheritance
Prototype: Get vs Set of Property

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

- Assume `pet` is `dog`'s prototype
Delegation to Prototype

- dog
  - name: "Rex"
  - age: 3
- pet
  - color: "blue"
Prototype: Get vs Set of Property

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

- Assume `pet` is `dog`'s prototype
  ```javascript
  // dog.name == ?
  // dog.color == ?
  pet.color = "brown";
  // dog.color is ?
  dog.color = "green";
  // dog.color is ?
  // pet.color is ?
  ```
Delegation to Prototype

```javascript
// get follows prototype chain

dog.color == ?
dog.color
```

Diagram:
- Dog: name = "Rex", age = 3
- Pet: color = "blue"
Delegation to Prototype

```
pet.color = "brown";
// set changes object
```

```
dog.color == ?
// get follows prototype chain
```
Delegation to Prototype

dog

pet

dog.color = "green";
// set changes object!

dog.color == ?
// get follows prototype chain
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

pet

name
"Rex"

age
3

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
  let 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

Circle

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

c = new Circle()
Prototypes And Constructors

c = new Circle()

Circle

prototype

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

constructor

area

Computer Science and Engineering ▶ The Ohio State University
Prototypes And Constructors

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

Properties:
- `centerX` = 10
- `centerY` = 12
- `radius` = 2.45

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

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

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

// bad: method is added to object itself
```
Method is in Object

```javascript
r = new Dog();

name: "Rex",
age: 6,
bark: 

Dog.prototype

return this.name + "says" + sound;

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

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

// good: add method to prototype
Idiom: Methods in Prototype

class Dog {
    constructor(n, a) {
        this.name = n;
        this.age = a;
    }

    bark(sound) {
        return this.name + "says" + sound;
    }
}

// best: ES6 class syntax
Methods in Prototype

```
Dog.prototype

r = new Dog()

Dog.prototype

name: "Rex"

age: 6

constructor:
this.name = n;
this.age = a;

bark
prototype

return this.name + " says " + sound;
```
class Dog {
    name = "Fur"; // property of object
    age;

    constructor(n, a) {
        this.name = n;
        this.age = a;
    }

    bark(sound) {
        return this.name + " says " + sound;
    }
}

class Dog {
    name: "Fur"; // property is in prototype!
    age: 0;

    constructor(n, a) {
        this.name = n; // hides prototype property
        this.age = a;
    }

    bark(sound) {
        return this.name + " says " + sound;
    }
}
Class Properties

```
r = new Dog();

Dog.prototype

   // constructor
   this.name = x;
   this.age = a;

   // prototype
   bark
   return this.name + " says " + sound;
```
Meaning of `r instanceof Dog`

```
r = new Dog()

prototype

Dog.prototype

constructor

return this.name + "says" + sound;

r.__proto__.constructor == Dog
```

```
dog = new Dog()

name: "Rex"
age: 6

this.name = x;
this.age = a;
```
Idiom: Classical Inheritance

```javascript
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

```javascript
r = new Dog();

new Animal() // Dog.prototype

Animal.prototype

Dog.prototype

Animal

Dog

new Animal();

r = new Dog();

name: "Rex"

constructor

prototype

constructor

prototype
```
Prototype Chains

- `instanceOf` is checked transitively up the prototype chain
  - `r instanceof Dog` //=> true
  - `r instanceof Animal` //=> true
  - `r instanceof Object` //=> true

- Q: Identify in the previous diagram
  - `r.__proto__.__proto__.constructor`
  - `Dog.prototype.__proto__.constructor`
  - `protype.prototype.constructor.prototype`
To Ponder

JavaScript Object Layout [Hursh Jain/mollypages.org]
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

- Any function can be a constructor

- Prototypes are "parent" objects
  - Delegation up the chain of prototypes
  - Prototype is determined by constructor
  - Prototypes can be modified