JavaScript: Objects, Methods, Prototypes

Lecture 14
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
  var myCar = {
    make: "Acura",
    year: 1996,
    plate: "NKR462"
  };
  ```
- To access/modify an object's properties:
  ```javascript
  myCar.make = "Ford"
  myCar["year"] = 2006;
  var str = "ate";
  myCar["pl"+str] = "NKR463";
  ```
Arrays vs Associative Arrays

- **Arrays**
  - Elements: 4, "hi", 3.14, true, false
  - Access: 0, 1, 2, 3

- **Associative Arrays**
  - Elements: age, greeting, doors, pi
  - Access: 0, 1, 2
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"}
  ```
Testing Presence of Key

- **Boolean operator:** *in*
  
  `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
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...)
  - An object, an array, or a 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

```python
myCar

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

honk()

play(sound);
return 0;
```
Keyword "this" in Functions

- Recall *distinguished parameter*
  
  ```javascript
  x.f(y,z); //x is the distinguished param.
  ```

- Inside a function, keyword "this"
  
  ```javascript
  function report() {
    return this.plate + this.year;
  }
  ```

- At run-time, "this" is set to the *distinguished parameter* 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

```java
myCar

register()

return this.plate + this.year;

plate "NKR462"

year 1996

report()

yourCar

info()

plate 340

year 2013
```
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

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

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

circle

- centerX: 10
- centerY: 12
- radius: 2.45

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()
```

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

```javascript
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 ...
```

```javascript
return Math.PI * this.radius * this.radius
```
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 (ie 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"
- doors: 4
- pi: 3.14
- age: 4
- toString()
- hasOwnProperty()
- push()
- pop()
- etc...

Example:

```javascript
var obj = { greeting: "hi", doors: 4, pi: 3.14, age: 4, greeting: function() { console.log(this.greeting); });
console.log(obj.greeting()); // "hi"
```
Class-Based Inheritance

interfaces

classes

objects

static

extends

implements

extends

static

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 "brown" (prop changed)
  dog.color = "green";
  //pet.color is still "brown" (hiding)
  ```
Delegation to Prototype

dog

"Rex"

3

"green"

pet

"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 (ie `[[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
function Circle() {
  this.centerX = x;
  this.centerY = y;
  ... Etc ...
}
```

```javascript
Circle.prototype = {
  centerX: 10,
  centerY: 12,
  radius: 2.45,
  area: function() {
    return Math.PI * this.radius * this.radius;
  }
};
```
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;
```
Idiom: Methods in Prototype

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;
}

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

// sets prototype to new anonymous object
Idiom: Methods in Prototype

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

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

//better: extends existing prototype
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

```javascript
new Dog()

name "Rex"

new Animal()

creator

prototype

Dog()

Animal()
```
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
Objects are Everywhere

- Global variables in JavaScript are a lie
- Implicitly part of some global object, provided by execution environment
  - See FireBug DOM view
  - In JSFiddle:
    ```javascript
    document.write(this + "<br/>");
    for (p in this) {
        document.write(p + "": " + this[p] + "<br/>");
    }
    ```
Window Object

- For JavaScript running in a browser, implicit global object is window
- Many properties, including
  - location (url of displayed document)
  - status (text in status bar of browser)
  - history
  - innerHeight, innerWidth
  - alert(), prompt()
  - document (tree of displayed document)
Document is a Tree

```html
<html lang="en">
  <head>
    <title>Something Short and Sweet</title>
    <meta charset="utf-8">
  </head>
  <body>
    <p>Hello <a href="planet.html">World</a>!
    <img src="pic.png" alt="a globe">
  </body>
</html>
```
DOM: "Document Object Model"

- DOM is a language-neutral API for working with HTML (and XML) documents
  - Different programming languages have different bindings to this API
  - But very closely linked to JavaScript

- In JavaScript, tree nodes → objects
  - A tree element with attributes
    `<input type="text" name="address">`
  - A JavaScript object with many properties
    ```
    { 
      tagName: "INPUT",
      type: "text",
      name: "address", /* lots more*/
    }
    ```
DOM History

- Ad hoc DOM existed from the beginning of JavaScript
  - Core purpose of client-side execution: Enable user interaction with the document
  - Need a connection between programming language (JavaScript) and the document
- DOM 1 specification (W3C) in '98
  - Standardized mapping tree→objects and functions for modifying the tree
- DOM 2 ('00): added styles and event handling
- DOM 3 ('04): fancier tree traversal & indexing schemes
Simplest Mapping

- **window**
  - **document**
    - **write**: outputs text to document body
    - **forms**: array of forms in a page
      - **elements[]**: array of widgets in a form
    - **anchors**: all the anchors in document
    - **links**: all the links in the document
    - **getElementById(string)**: finds a node
    - etc...
Document is a Tree

head
  --title
    charset: utf-8
  --meta
    charset: utf-8

body
  --p
    a
      href: planet.html
    br
      img
        src: pic.png
        alt: a globe
  Hello
    Something Short and Sweet
  World
Node is a JavaScript Object

- **Properties**
  - `parentNode`, `childNodes`, `firstChild`, `lastChild`, `nextSibling`, `previousSibling`
  - `innerHTML`
  - `tagName`
    - XML lower case ("a"), HTML upper case ("A")
  - `attributes`, `name`, `id`, `class`
  - `style`
    - Hyphenated properties in CSS (background-color)
    - Become camelCase in JavaScript (backgroundColor)

- **Methods**
  - `appendChild(node)`, `removeChild(node)`, `insertBefore(node)`
  - `hasAttribute(attr)`, `removeAttribute(attr)`, `getAttribute(attr)`, `setAttribute(attr)`
  - `getElementsByTagName(name)`
How to Find a Node in Tree

1. Hard coding with "flat" techniques
   - Eg children arrays
     \[\text{document.forms[0].elements[0]}\]
   - Downside: too brittle
   - If the document structure changes a little, everything breaks

2. Using an element's name attribute
   - In HTML:
     \[
     \text{<form name="address"> ...} \\
     \text{<input name="zip"... /> </form>}
     \]
   - In JavaScript:
     \[
     \text{document.address.zip}
     \]
   - Downside: direct path still hard coded
How to Find a Node in Tree

3. To get a unique element: document method getElementById
   - In HTML
     `<td id="shipping">...</td>`
   - In JavaScript
     `document.getElementById("shipping")`
   - Downside: every element you want to find needs unique ID

4. Combination: element ID for form, arrays for options in selection element
Example

```html
<form id="wheels">
  <input type="checkbox" name="vehicles" value="car" /> Car
  <input type="checkbox" name="vehicles" value="truck" /> Truck
  <input type="checkbox" name="vehicles" value="bike" /> Bike
</form>

var numChecked = 0;
var elt = document.getElementById("wheels");
for (i = 0; i < elt.vehicles.length; i++) {
  if (elt.vehicles[i].checked)
    numChecked++;
}
```
Interactive Documents

- To make a document interactive, you need:
  - Widgets (ie HTML elements)
    - Buttons, windows, menus, etc.
  - Events
    - Mouse clicked, window closed, button clicked, etc.
  - Event listeners
    - Listen (ie wait) for events to be triggered, and then perform actions to handle them
Events Drive the Flow of Control

- This style is *event driven* programming

- Event handling occurs as a loop:
  - Program is idle
  - User performs an action
    - Eg moves the mouse, clicks a button, types in a text box, selects an item from menu, ...
  - This action generates an event (object)
  - That event is sent to the program, which responds
    - Code executes, could update document
  - Program returns to being idle
Handling Events Mechanism

- Three parts of the event-handling mechanism
  - *Event source*: the widget with which the user interacts
  - *Event object*: encapsulated information about the occurred event
  - *Event listener*: a function that is called when an event occurs, and responds to the event
Programmer Tasks

- Define an event handler
  - Any function can be an event handler
  - Often need information about the triggering event in order to know what response is needed
- Register handler with source element
- Detect event and invoke handler
  - Ha! Just kidding, you do NOT do this
Simple Example: Color Swaps

<p>This page illustrates changing colors</p>
<form>
  <p>
    <label> background: 
      <input type="text" name="back" size="10"
          onchange="foo('bg', this.value)" />
    </label> <br />
  </p>
  <label> foreground: 
    <input type="text" name="fore" size="10"
          onchange="foo('fg', this.value)" />
  </label>
</form>
function foo(place, color) {
    if (place === "bg") {
        document.body.style.backgroundColor = color;
    } else {
        document.body.style.color = color;
    }
}
Event Propagation

- Elements are nested in tree
- When an event occurs, which element's handler(s) is(are) notified?
- First, *propagation path* is calculated: from root to smallest element
- Then event dispatch occurs in 3 phases
  1. Capture (going *down* the path)
  2. Target (smallest element)
  3. Bubble (going *up* the path, reverse of 1)
http://www.w3.org/TR/DOM-Level-3-Events/
Bubbling Up

- Usually, handling is done in phase 2 and 3
- Example: mouse click on hyperlink
  - Handler for `<a>` element displays a pop-up ("Are you sure you want to leave?")
  - Once that is dismissed, event flows up to enclosing `<p>` element, then `<div>` then... etc until it arrives at root element of DOM
  - This root element (ie window) has a handler that loads the new document in the current window
Programmer Tasks

- Define a handler
  - Easy, any function will do

- Register handler
  - Multiple ways to link (HTML) tree elements with (JavaScript) functions

- Be triggered by the event
  - Ha! Still kidding

- Get information about triggering event
  - Multiple (incompatible) ways for handler to get the event object
Registering an Event Handler

- Three techniques, ordered from:
  - Oldest (most brittle, most universal) to
  - Newest (most general, least standard)

1. Inline (link in HTML itself)
   `<a href="page.html" onclick="foo()">...</a>

2. Direct (link in JavaScript)
   `var e = ...//find source element in tree
   e.onclick = foo;`

3. Chained (In JavaScript, browser differences)
   `var e = ...//find source element in tree
   e.addEventListener("click", foo, false);`
Inline Registration (pre DOM)

- HTML attributes, vary by element type
  - For window: onload, onresize, onunload, ...
  - Forms & elements: onchange, onblur, onfocus, onsubmit, ...
  - Mouse events: onclick, onmouseover, onmouseout, ...
  - Keyboard events: onkeypress, onkeyup, ...
- The *value* of these attributes is JavaScript code to be executed
  - Normally just a function invocation
- Example
  `<a href="page.html" onclick="foo()">...
- Advantage: Quick, easy, universal
- Disadvantage: mixes code with content
Direct Registration (DOM 1)

- Use properties of DOM element objects
  - onchange, onblur, onfocus,...
  - onclick, onmouseover, onmouseout,...
  - onkeypress, onkeyup,...
- Set this property to appropriate handler
  ```javascript
  var e = ... //find source element in tree
  e.onclick = foo;
  ```
- Note: no parentheses!
  ```javascript
  e.onclick = foo(); //what does this do?
  ```
- Disadvantage? No arguments to handler
  - Not a problem, handler gets event object
- Real disadvantage: 1 handler/element
Example

```javascript
var x = document.getElementsByTagName("div");
for (var i = 0; i < x.length; i++) {
    x[i].onmouseover = function () {
        this.style.backgroundColor = "red"
    }
    x[i].onmouseout = function () {
        this.style.backgroundColor = "blue"
    }
}
```
Chained Registration (DOM 2)

- Each element has a *collection* of handlers
- Add/remove handler to this collection
  ```javascript
  var e = ... // find source element in tree
  e.addEventListener("click", foo, false);
  ```
- Note: no "on" in event names (ie "click")
- Third parameter: true for capture phase
- Disadvantage: browser incompatibilities
  ```javascript
  e.addEventListener()  // FF, Webkit, IE9+
  e.attachEvent()       // IE5-8
  ```
- Many browser compatibility issues with DOM and events
- Solution: Libraries
  - Eg jQuery, Dojo, Prototype, YUI, MooTools,...
Example

```javascript
var x = document.getElementsByTagName("div");
for (var i = 0; i < x.length; i++) {
    x[i].addEventListener("click",
        function () {
            this.act = this.act || false;
            this.act = !this.act;
            this.style.backgroundColor = (this.act ? "red" : "gray");
        },
        false);
}
```
Task: Getting Event Object

- Most browsers: parameter to handler
  function myHandler(event)
- IE: event is property of window
- Common old-school idiom:
  function myHandler(event) {
    event = event || window.event;
    ... etc ...
- Again, libraries are the most robust way to deal with these issues
Summary

- **DOM: Document Object Model**
  - Programmatic way to use document tree
  - Get, create, delete, and modify nodes

- **Event-driven programming**
  - Source: element in HTML (a node in DOM)
  - Handler: JavaScript function
  - Registration: in-line, direct, chained
  - Event is available to handler for inspection