Announcements

• Problem Set 1 will be ready soon

• Added another example (tasks.html and tasks.css) to the previous lecture notes

• Reading assignment for this slide set:
  • JavaScript tutorial
    • [http://www.w3schools.com/js/default.asp](http://www.w3schools.com/js/default.asp)

• Take a break around 10:15am
JavaScript

- One of 3 languages for all web developers **must** learn:
  - **HTML** to define the content of web pages
  - **CSS** to specify the layout of web pages
  - **JavaScript** to program the behavior of web pages

- Is a scripting (programming) language with
  - dynamic typing and
  - first-class functions

- Used as part of a web browser
  - Handles static content
  - cf. for dynamic content, we will use PHP later
JavaScript (cont.)

- Allows client-side scripts to
  - interact with the user
  - control the browser
  - alter the document content that is displayed
  - cf. server-side scripting will be done using PHP later

- JavaScript files named with .js extension, e.g., demo.js

- Not related to Java at all although it shares the same name
What you need

- A text editor for writing code, e.g., Notepad++ (Windows), Text Wrangler (Macs), Emacs or Sublime Text 2/3 (both)
  - What is your favorite? (I am curious)

- A web browser, e.g., Chrome, Firefox

- Opening a JavaScript interpreter on Chrome
  - OSX: Command+Option+i
  - Windows: F12 or Ctrl+Shift+i
  - Alternatively, you can right click anywhere on any web page and choose “Inspect Element”
  - To type commands that span multiple lines, use Shift+Enter
Some basic concepts

- An **element** in HTML is any tagged item in the page, e.g., `<b>…</b>`, `<h3>…</h3>`, `<table>…</table>`, etc.

- **document** is the HTML DOM (Document Object Model) object that represents a web page
  - It is the owner of all other objects in your web page

- Finding HTML elements: If you want to access objects in an HTML page, you start with the document object, e.g.,
  - `document.getElementById()` to find an element by element id
  - `document.getElementsByTagName()` to find elements by tag name
  - `document.getElementsByClassName()` to find elements by class name

- Changing HTML elements, e.g.,
  - `element.innerHTML = ...` to change the inner HTML of an element
  - `element.setAttribute = ...` to change the attribute of an HTML element
  - etc.
Examples

- See js1.html
- See js2.html, pic_bulbon.gif, pic_bulboff.gif
- See js3.html
- See js4.html, bmi.js
- See bmi.html
- See forms.html
- See js5.html, js5.js, hike.html, surf.html
- See js6.html, js6.js
JavaScript, the language

- Types
  - String
  - Number
  - Boolean
  - Null
  - Undefined
  - Object
Strings

> “Hello world”
> s = ‘Hello world’
> typeof s  -> “string”
> s.charAt(1)
> s.substr(6)
> s.toUpperCase()
> s.length
> t = s
> s == t
> s += ‘test’
> . . .
Numbers

All numbers are 64-bit floating-point numbers in JavaScript

> n = 6.23
> typeof n
> 1/3  -> 0.333333
> a = 3/undefined  -> NaN
> typeof a  -> “number”
> b = 6/0   -> Infinity
> b == Number.POSITIVE_INFINITY  -> true
> c = -6/0   -> -Infinity
> Math.pow(3,2)  -> 9
> Math.round(3.22)  -> 3
> . . .
Booleans

- JavaScript supports a boolean type with the literal values true and false

> t = true
> typeof t -> "boolean"
> f = false
> f != t -> true (other relational ops: >, <, ==, !=, >=, <=)
> . . .
Null

- Null is a data type that has a single value: \texttt{null}

```javascript
> n = null
> typeof n -> "object" (null is considered to be of type object)
> ```
Undefined

- The undefined data type is returned when you access a property on an object that does not exist, or use a variable before it is declared, or before it is assigned a value.

> typeof g  -> “undefined”
Objects

- Other than the types outlined so far, all data types in JavaScript are objects
- This includes arrays, functions, dates, and user defined objects
Truthy and falsey values

- The following are all considered false in JavaScript:
  - false
  - 0 (zero)
  - "" (empty string)
  - null
  - undefined
  - NaN

- == vs. ===
  - == means “equal to”
  - === means “equal value and equal type”
  - Examples:
    - x = 5
    - x == 5   -> true
    - x === 5     -> true
  - x == "5" -> true
  - x === "5"  -> false

- Similarly for != vs. !==
Dynamic typing

- JavaScript is a dynamically typed language
- Type of a value is checked at run-time
- cf. statically typed language
  - Type of a piece of code is checked at compile-time
- Example:
  ```javascript
  function add(v1, v2) {
    return v1 + v2;
  }
  > add(1, 1) -> 2
  > add(1, "1") -> "11"
  ```
Object-oriented

- Both classical class-based as well as prototype-based

- **Class-based**
  - As in Java and C++
  - Rigid
    - Define a class first
    - Create instances based on the class
    - Structure of the instances remain the same

- **Prototype-based**
  - More on next slide
Class?-based objects

- Create a constructor and instances based on the constructor

```javascript
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
var myFather = new Person("John", "Doe", 50, "blue");
var myMother = new Person("Sally", "Rally", 48, "green");
```
Built-in JavaScript constructors

- JavaScript has built-in constructors for native objects:

```javascript
var x1 = new Object();    // A new Object object
var x2 = new String();    // A new String object
var x3 = new Number();    // A new Number object
var x4 = new Boolean();   // A new Boolean object
var x5 = new Array();     // A new Array object
var x6 = new RegExp();    // A new RegExp object
var x7 = new Function();  // A new Function object
var x8 = new Date();      // A new Date object
```
Some advice

- As you can see, JavaScript has object versions of the primitive data types String, Number, and Boolean.
- There is no reason to create complex objects. Primitive values execute much faster.
- And there is no reason to use new Array(). Use array literals instead: []
- And there is no reason to use new RegExp(). Use pattern literals instead: /()/
- And there is no reason to use new Function(). Use function expressions instead: function () {}
- And there is no reason to use new Object(). Use object literals instead: { }
Some advice (cont.)

- Examples

```javascript
var x1 = {};            // new object
var x2 = "";            // new primitive string
var x3 = 0;             // new primitive number
var x4 = false;         // new primitive boolean
var x5 = [];            // new array object
var x6 = /()/           // new regexp object
var x7 = function(){};  // new function object
```
Prototype-based objects

- Flexible
- Start with a possibly empty object without a class definition
- Add properties as needed to the object at run-time

Example:
  > o1 = {}
  > typeof o1  -> “object”

Hmm… o1 is not of any particular type - it is just an “object”!
- How do we write code without knowing the type?
- If a function is passed an object without its type, how does it know what to do with it?
Determining an object’s type

- **Duck typing**: a style of typing in which an object’s methods and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of an explicit interface.

- The term comes from the **duct test**: “When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck” [James W. Reiley]

- Example: “If it has name, major, and gpa properties, it is probably a student.”
Creating an object (1)

- An empty object is not very useful – let’s add some properties to it
  
  ```javascript
  > o1.fn = ‘John’;
  > o1.ln = “Doe’;
  > o1.age = 22;
  > o1.incAge = function() {
  // this.age++;
  }
  > o1  -> Object {fn: “John”, ln: “Doe”, age: 22, incAge: function}
  > o1.fn       -> “John”
  > o1[‘fn’]    -> “John”
  > o1.incAge() 
  > o1.age      -> 23
  ```

- Object in JavaScript is really just an associative array (i.e., a hash map)
Creating an object (2)

- All at once

```javascript
> o2 = {fn: 'Amy', ln: 'Jones', age: 21, incAge: function() { this.age++; } };  
```

- This would be a major inconvenience, especially if you have many methods on your object.

- Well, see next slide . . .
Creating an object (3)

- As a **prototype** of another object by cloning

```
> function clone(obj) {
    if (obj == null || typeof obj != 'object') {
        return obj;
    }
    var newo = { }
    for (var key in obj) {
        newo[key] = clone(obj[key]);
    }
    return newo;
}
> o3 = clone(o2);
```

- We have retained the methods already defined on its prototype!
- We will see prototypes in detail later and **redo this**!
With web apps, we need to send data between clients and servers

JSON is a data format useful for serialization and de-serialization

- Simpler than XML format, far less verbose!

Example:

```javascript
> s = JSON.stringify(p1)
> s
"{"fn":"John", "ln":"Doe","age":22,"addr":{"city":"LA","zip":90900}}"
> p2 = JSON.parse(s)
```

See [http://www.json.org/](http://www.json.org/) for more info
JSON values

- Three types
  - **Objects**, denoted with curly brackets `{ }`
  - **Arrays**, which are denoted by square brackets `[ ]` and contain comma separated values just like JavaScript arrays
  - **Literal values** (strings, numbers, booleans)
- An object or an array can in turn contain any of these 3 types

- JavaScript makes it possible to convert from a *stringified* version to an *object* representation purely because the language does not rely on classes!

- JSON has become a widely used data format even for apps that do not use JavaScript
Prototypes

- JavaScript provides an elegant mechanism for extending objects called *prototypes*, thus is considered a *prototype-based* language.

- Prototype languages use existing objects as the basis for new objects, which are then modified to meet their specific needs.

- Hmm… where did the `toString` method below come from?

  ```javascript
  > o1= {
  > o1.toString()    ->   "[object Object]"
  ```

- All JavaScript objects have a `prototype` object!
  - This prototype object is an object and can encapsulate properties and methods.
  - All JavaScript objects inherit their properties and methods from their prototype.
Chain of objects/prototypes.

- There can be a whole chain of objects because the object that is our prototype may itself have a prototype.

- If no objects in the property chain have a property matching the name specified, then JavaScript returns a special type called `undefined`.

- The prototype of our empty object was provided by `Object.prototype`.
  - This is the *only* object that does not have a prototype of its own, and therefore is the end of the chain. (cf. `Object` in Java)

- We can overwrite the `toString` in our own object, e.g.,
  ```javascript
  > o1.toString = function() { return “I use my own toString”; };
  > o1.toString()    ->    “I use my own toString”
  ```

- Other objects, e.g., an empty object `o2`, are not affected by this new `toString` though!
Creating a prototype

- Create an object prototype using an object constructor function

  function Person(first, last, age, eyeColor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyeColor;
  }

- With a constructor function, you can use the `new` keyword to create new objects from the same prototype:

  var myFather = new Person("John", "Doe", 50, "blue");
  var myMother = new Person("Sally", "Rally", 48, "green");
Adding properties and methods to objects

- Sometimes you want to add new properties (or methods) to an existing object.

- Sometimes you want to add new properties (or methods) to all existing objects of a given type.

- Sometimes you want to add new properties (or methods) to an object prototype.
Adding a property to an object

- Adding a new property to an existing object:

  ```javascript
  myFather.nationality = "English";
  ```

- The property will be added to `myFather`. Not to `myMother`. Not to any other person objects.
Adding a method to an object

- Adding a new method to an existing object

```javascript
myFather.name = function () {
    return this.firstName + " " + this.lastName;
};
```

- The method will be added myFather. Not to myMother.
Adding properties/methods to a prototype

- You cannot add a new property to a prototype the same way as you add a new property to an existing object, because the prototype is not an existing object.
- To add a new property or a method to a constructor, you must add it to the constructor function:

```javascript
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
    this.nationality = "English"
    this.name = function() {
        return this.firstName + " " + this.lastName;
    };
}
```
Using the prototype property

- The JavaScript prototype property allows you to add new properties or methods to an existing prototype:

```javascript
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
Person.prototype.nationality = "English";
Person.prototype.name = function() {
    return this.firstName + " " + this.lastName;
};
```
Array.prototype

- **Object.prototype** is the prototype of **Array.prototype**

- **Array.prototype** has its own properties and methods, some (e.g., **toString**) overwriting the ones in **Object.prototype**

```javascript
> a = [1, 2, 3, 4, 5];
> typeof a         ->  "object"
> a.toString()    ->  "1,2,3,4,5"     // note the output format
> a.reverse()     ->  [5, 4, 3, 2, 1]
> a.pop()         ->  1
> a.push(6)       ->  5
> a                ->  [5, 4, 3, 2, 6]
```
Adding things to `Array.prototype`

- All arrays are based on the same prototype, i.e., `Array.prototype`.
- If we add methods to this prototype, they immediately become available to all arrays.
- Prototypes are just objects themselves and can be modified just like any other object.
- Arrays don’t have a ‘`contains`’ method and we can add it to the prototype
  ```javascript
  Array.prototype.contains = function(val) {
    for (var i = 0; i < this.length; i++) {
      if (this[i] === val) { return true; }
    }
    return false;
  }
  > [1, 2, 3, 4, 5].contains(3)    ->   true
  ```
clone redone as extends

- The new implementation of clone (renamed extends) still accepts an object, but now returns a new empty object with that object set as its prototype!

```javascript
> function extends(obj) {
    function E() { }; // constructor
    E.prototype = obj;
    return new E();
};

> p2 = extends(p1);
> p2.fn = 'Ken';
> p2.ln = 'Jones';
> p2.age = 23;
```

- p2 now has whatever properties and methods that p1 has
Functional programming in JavaScript

- JavaScript supports first-class functions, i.e., functions and methods are actually objects in JavaScript
  
  - Defined on `Function.prototype`
  
  - Higher order functions
    - `filter`
    - `map`
    - `reduce`
    - . . .
    - See `hof.html`
Function arguments

- JavaScript does not check that the arguments passed to a function match the signature of the function
- You may pass more arguments to a function than it expects
  - The extra arguments will be ignored
- You may pass fewer arguments to a function than it expects
  - The parameters are assigned the value of `undefined`
- So, it is not possible to overload functions or methods in JavaScript
- Any time a function is called, a variable named arguments is available as an array containing all the actual arguments
Examples

> function add() {
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}

> add(6, 3, 2, 4, 12)
Closures

- JavaScript supports closures
- We may wish to access function scoped variables even after the function has completed!
- Example

  ```javascript
  > function f() {
  >     var i = 0;
  >     return function() { // anonymous function
  >         return ++i;
  >     };
  > }
  > incrementer = f()                        > incrementer2 = f()
  > incrementer()  -> 1                     > incrementer2()  -> 1
  > incrementer()  -> 2                     > incrementer2()  -> 2
  > incrementer()  -> 3                     > incrementer2()  -> 3
  ```
What is going on?

- The function does the following:
  1. Declares a function scoped variable i initialized to 0.
  2. Returns an anonymous function that will increment this variable when invoked.

- When the anonymous function was defined inside f, it “closed” over its environment as it existed at that point in time, and kept a copy of that environment.

- Since the variable i was accessible when the function was declared, it is still available when the function is invoked afterwards.
Alternatives to closure?

- Use a regular function? No!

  > function f() {
  var i = 0;
  return ++i;
  }

- Use a global variable inside the previous example? No!
Alternatives to closure? (cont.)

- Use an object? Not quite!

```javascript
> obj1 = { i: 0,
    increment: function() {
        return ++this.i;
    }
}

> obj1.increment()    -> 1
> obj1.increment()    -> 2
> obj1.i = 22;
> obj1.increment()    -> 23
> obj1.i = "hello world"
> obj1.increment()    -> NaN
```

- JavaScript objects can’t hide data – there is no visibility control mechanism such as `private` in Java

- Data hiding (or encapsulation) is enormously important when writing large apps!
The **module** design pattern

- The closure-based incremener is one such example.
- The **module** design pattern is also commonly implemented by returning an object rather than a function.
- This allows the object to use private variable for its private properties and functions, while still exposing its public properties and methods.

- Let’s rewrite the incremener using an object this time:
  ```javascript
  function createIncrementer() {
    var i = 0;
    return { increment: function() { return ++i; } }; // an object
  }
  
  obj = createIncrementer();
  obj.increment() -> 1
  obj.increment() -> 2  // this would merely create a new property on the object.
  obj.i = 10;
  obj.increment() -> 3
  ```
Threading

- Within a browser environment, all JavaScript code executes on a single thread!

- This is also the thread that the browser utilizes to update the window.

- Browsers allow documents in different tabs to utilize different threads.
  - This is possible because the tabs do not interact with one another.

- The browser manages the single thread with an event queue for all different types of events: browser initiated events (e.g., resizing a window), user initiated ones (e.g., button clicking), JavaScript initiated ones (e.g., a timer executing a function), etc.
Arrays

- Arrays are objects
- See arrays.html
• jQuery

Next . . .