Topics in This Section

- **Functions**
  - Basics
  - As first-class data types
  - Anonymous functions (closures)
- **Objects**
  - Object basics
  - Namespaces (static methods)
  - JSON
  - eval
- **Functions with variable numbers of arguments**

---

**Intro**

“JavaScript has more in common with functional languages like Lisp or Scheme than with C or Java.”
- Douglas Crockford in article “JavaScript: The World’s Most Misunderstood Programming Language”.

Customized Java EE Training: [http://courses.coreservlets.com/](http://courses.coreservlets.com/)  
Servlets, JSP, JSF 2.0, Struts, Ajax, GWT 2.0, Spring, Hibernate, SOAP & RESTful Web Services, Java 6. 
Developed and taught by well-known author and developer. At public venues or onsite at **your** location.
Getting Good at JavaScript

• JavaScript is not Java
  – If you try to program JavaScript like Java, you will never be good at JavaScript.

• Functional programming is key approach
  – Functional programming is much more central to JavaScript programming than OOP is.
  – Java programmers find functional programming to be the single-hardest part of JavaScript to learn.
    • Because Java does not support functional programming
    • But programmers who use Ruby, Lisp, Scheme, Python, ML, Haskell, Clojure, Scala, etc. are accustomed to it

• OOP is radically different than in Java
  – So different in fact, that some argue that by Java’s definition of OOP, JavaScript does not have “real” OOP.

Functions

“It is Lisp in C’s clothing.”
- JSON and YUI guru Douglas Crockford, describing the JavaScript language in *JavaScript: The Good Parts*.

Customized Java EE Training: [http://courses.coreservlets.com/](http://courses.coreservlets.com/)
Servlets, JSP, JSF 2.0, Struts, Ajax, GWT 2.0, Spring, Hibernate, SOAP & RESTful Web Services, Java 6.
Developed and taught by well-known author and developer. At public venues or onsite at your location.
Overview

• Not similar to Java
  – JavaScript functions *very* different from Java methods

• Main differences from Java
  – You can have global functions
    • Not just methods (functions as part of objects)
  – You don’t declare return types or argument types
  – Caller can supply any number of arguments
    • Regardless of how many arguments you defined
  – Functions are first-class datatypes
    • You can pass functions around, store them in arrays, etc.
  – You can create anonymous functions (closures)
    • Critical for Ajax
    • These are equivalent
      – function foo(...) {...}
      – var foo = function(...) {...}

Functions are First-Class Data Types

– Can assign functions to variables
  • function square(x) { return(x*x); }
  • var f = square;
  • f(5); \rightarrow 25

– Can put functions in arrays
  • function double(x) { return(x*2); }
  • var functs = [square, f, double];
  • functs[0](10); \rightarrow 100

– Can pass functions into other functions
  • someFunction(square);

– Can return functions from functions
  • function blah() { … return(square); }

– Can create a function without assigning it to a variable
  • (function(x) {return(x+7);})(10); \rightarrow 17
Assigning Functions to Variables

• **Examples**

  function square(x) { return(x*x); }
  var f = square;
  square(5); \(\rightarrow\) 25
  f(5); \(\rightarrow\) 25

• **Equivalent forms**

  function square(x) { return(x*x); }
  var square = function(x) { return(x*x); };
Passing Functions into Other Functions

function third(x) {
    return(x / 3);
}

function triple(x) {
    return(x * 3);
}

function nineTimes(x) {
    return(x * 9);
}

function operate(f) {
    var nums = [1, 2, 3];
    for(var i=0; i<nums.length; i++) {
        var num = nums[i];
        console.log("Operation on \%o is \%o.",
                    num, f(num));
    }
}

Returning Functions from Functions

• Examples

function randomFunct() {
    if(Math.random() > 0.5) {
        return(square);
    } else {
        return(double)
    }
}

var f3 = randomFunct();
f3(5); // Returns either 25 or 10

• Dynamically created functions
  – Instead of a predefined function like square, you can return a new function with return(function(…) { …});
Can Create a Function without Assigning it to a Variable

- **Examples**
  
  \[
  \text{(function(x) \{ return(x+7); \})(10);} \rightarrow 17
  \]

  ```javascript
  function randomFunct2() {
    if(Math.random() > 0.5) {
      return(function(x) { return(x*x); });
    } else {
      return(function(x) { return(x*2); });
    }
  }
  ```
  
  – Same behavior as previously shown randomFunct

- **More on anonymous functions**
  
  – Called “closures” if the functions refer to local variables from the outside. Can’t do Ajax without them!
Anonymous Functions with Static Data

- Examples
  ```javascript
  function makeTimes7Function() {
    return(function(n) { return(n*7); });
  }
  var f = makeTimes7Function();
  f(7); → 49
  ```

- Equivalent form of function above
  ```javascript
  function makeTimes7Function() {
    var m = 7;
    return(function(n) { return(n*m); });
  }
  var m = 700; // Value of global m is irrelevant
  var f = makeTimes7Function();
  f(7); → 49
  ```

Anonymous Function with Captured Data (Closures)

```javascript
function makeMultiplierFunction(m) {
  return(function(n) { return(n*m); });
}

var test = 10;
var f = makeMultiplierFunction(test);
f(7); → 70

var test = 100;
f(7); → 70 // Still returns 70
```

Point: when you call makeMultiplierFunction, it creates a function that has its own private copy of m. This idea of an anonymous function that captures a local variable is the only way to do Ajax without having the global variable problems that we showed in first section.
The apply Method: Simple Use

• Idea
  – Lets you apply function to array of arguments instead of individual arguments. It is a method of functions!
    • someFunction.apply(null, arrayOfArgs);
  – Later, we cover advanced usage with obj instead of null

• Examples
  function hypotenuse(leg1, leg2) {
    return(Math.sqrt(leg1*leg1 + leg2*leg2));
  }
  hypotenuse(3, 4); \rightarrow 5
  var legs = [3, 4];
  hypotenuse.apply(null, legs); \rightarrow 5
  Math.max.apply(null, [1, 3, 5, 7, 6, 4, 2]); \rightarrow 7

The call and apply Methods: Use with Objects

• Idea
  – call
    • Lets you call function on args, but sets “this” first.
      – Will make more sense once we cover objects, but the main idea is that “this” lets you access object properties. So, “call” treats a regular function like a method of the object.
  – apply
    • Same idea, but you supply arguments as array

• Examples
  function fullName() {
    return(this.firstName + " " + this.lastName);
  }
  fullName(); \rightarrow "undefined undefined"
  var person = { firstName: "David", lastName: "Flanagan" };
  fullName.call(person); \rightarrow "David Flanagan"
Object Basics

Basics

- **Constructors**
  - Functions named for class names. Then use “new”.
  - No separate class definition! No “real” OOP in JavaScript!
  - Can define properties with “this”
    - You must use “this” for properties used in constructors
      ```javascript
      function MyClass(n1) { this.foo = n1; }
      var m = new MyClass(10);
      m.bar = 20; // Now m.foo is 10 and m.bar is 20
      ```
- **Properties (instance variables)**
  - You don’t define them separately
    - Whenever you refer to one, JavaScript just creates it
      ```javascript
      m.bar = 20; // Now m.foo is 10 and m.bar is 20
      ```
    - Usually better to avoid introducing new properties in outside code and instead do entire definition in constructor
- **Methods**
  - Properties whose values are functions
Objects: Example (Circle Class)

```javascript
function Circle(radius) {
    this.radius = radius;

    this.getArea =
        function() {
            return(Math.PI * this.radius * this.radius);
        }
}

var c = new Circle(10);
c.getArea(); // Returns 314.1592...
```

The prototype Property

- **In previous example**
  - Every new Circle got its own copy of radius
    - Fine, since radius has per-Circle data
  - Every new Circle got its own copy of getArea function
    - Wasteful, since function definition never changes
- **Class-level properties**
  - Classname.prototype.propertyName = value;
- **Methods**
  - Classname.prototype.methodName = function() {...};
    - Just a special case of class-level properties
  - This is legal anywhere, but it is best to do it in constructor
- **Pseudo-Inheritance**
  - The prototype property can be used for inheritance
    - Complex. See later section on Prototype library
Objects: Example (Updated Circle Class)

```javascript
function Circle(radius) {
    this.radius = radius;

    Circle.prototype.getArea =
    function() {
        return(Math.PI * this.radius * this.radius);
    };
}

var c = new Circle(10);
c.getArea(); // Returns 314.1592...
```
**Static Methods (Namespaces)**

- **Idea**
  - Have related functions that do not use object properties
  - You want to group them together and call them with `Utils.func1`, `Utils.func2`, etc.
    - Grouping is a syntactic convenience. Not real methods.
    - Helps to avoid name conflicts when mixing JS libraries
  - Similar to static methods in Java

- **Syntax**
  - Assign functions to properties of an object, but do not define a constructor. E.g.,
    - `var-utils = { }; // Or new Object(), or make function `Utils` 
      `Utils.foo = function(a, b) { ... };`
      `Utils.bar = function(c) { ... };`
      `var x = Utils.foo(val1, val2);`
      `var y = Utils.bar(val3);`

**Static Methods: Example (Code)**

```javascript
var MathUtils = {};

MathUtils.fact = function(n) {
    if (n <= 1) {
        return(1);
    } else {
        return(n * MathUtils.fact(n-1));
    }
};

MathUtils.log10 = function(x) {
    return(Math.log(x)/Math.log(10));
};
```

![Firebug - Static Methods](image)
Namespaces in Real Applications

- **Best practices in large projects**
  - In many (most?) large projects, *all* global variables (including functions!) are forbidden due to the possibility of name collisions from pieces made by different authors.
  - So, these primitive namespaces play the role of Java’s packages. Much weaker, but still very valuable.

- **Fancy variation: repeat the name**
  - var MyApp = { }
  - MyApp.foo = function foo(...) { … };
  - MyApp.bar = function bar(...) { … };
  - The name on the right does not become a global name. The only advantage is for debugging
  - Firebug and other environments will show the name when you print the function object.

---

JSON:
Anonymous Objects

Customized Java EE Training: [http://courses.coreservlets.com/](http://courses.coreservlets.com/)
Servlets, JSP, JSF 2.0, Struts, Ajax, GWT 2.0, Spring, Hibernate, SOAP & RESTful Web Services, Java 6.
Developed and taught by well-known author and developer. At public venues or onsite at your location.
JSON (JavaScript Object Notation)

- **Idea**
  - A simple textual representation of JavaScript objects
    - Called “object literals” or “anonymous objects”
  - Main applications
    - One-time-use objects (rather than reusable classes)
    - Objects received via strings

- **Directly in JavaScript**
  - `var someObject =`
    ```javascript
    { property1: value1, 
    property2: value2, 
    ... };
    ```

- **In a string (e.g., when coming in on network)**
  - Surround object representation in parens
  - Pass to the builtin “eval” function

---

### JSON: Example

```javascript
var person =
{ firstName: 'Brendan',
  lastName: 'Eich',
  bestFriend: { firstName: 'Chris',
                lastName: 'Wilson' },
  greeting: function() {
    return("Hi, I am " + this.firstName + 
          " " + this.lastName + ").
  }
};
```

```
>>> person.firstName;
"Brendan"
>>> person.lastName;
"Eich"
>>> person.bestFriend.firstName;
"Chris"
>>> person.bestFriend.lastName;
"Wilson"
>>> person.greeting();
"Hi, I am Brendan Eich."
...```

Internet Explorer and Extra Commas

- **Firefox & Chrome tolerate trailing commas**
  - Tolerated in both arrays and anonymous objects
    - var nums = [1, 2, 3, ];
    - var obj = { firstName: "Joe", lastName: "Hacker", };  
- **IE will crash in both cases**
  - For portability, you should write it *without* commas after the final element:
    - var nums = [1, 2, 3];
    - var obj = { firstName: "Joe", lastName: "Hacker"};
  - This issue comes up moderately often, especially when building JSON data on the server, as we will do in upcoming lectures.

Other Object Tricks

- **The instanceof operator**
  - Determines if lhs is a member of class on rhs
    - if (blah instanceof Array) {
      doSomethingWith(blah.length);
    }

- **The typeof operator**
  - Returns direct type of operand, as a String
    - "number", "string", "boolean", "object", "function", or "undefined".
    - Arrays and null both return "object"

- **Adding methods to builtin classes**
  - `String.prototype.describeLength = function() { return("My length is " + this.length); };`
  - "Any Random String".describeLength();

- **eval**
  - Takes a String representing *any* JavaScript and runs it
    - `eval("3 * 4 + Math.PI");  // Returns 15.141592`
More on eval

- **Simple strings**
  - Just pass to eval
    - var test = "[1, 2, 3, 2, 1].sort()";
    - eval(test); \[1, 1, 2, 2, 3\]

- **Strings that are delimited with { ... }**
  - You have to add extra parens so that JavaScript will know that the braces are for object literals, not for delimiting statements.
    - It never hurts to do this, so add parens routinely
    - var test2 = "\{ firstName: 'Jay', lastName: 'Sahn' \}";
    - var person = eval("(" + test2 + ")");
    - person.firstName; \"Jay\"
    - person.lastName; \"Sahn\"
Variable Args: Summary

- **Fixed number of optional args**
  - Functions can *always* be called with any number of args
  - Compare typeof args to "undefined"
  - See upcoming convertString function

- **Arbitrary args**
  - Discover number of args with arguments.length
  - Get arguments via arguments[i]
  - See upcoming longestString function

- **Optional args via anonymous object**
  - Caller always supplies same number of arguments, but
    one of the arguments is an anonymous (JSON) object
    - This object has optional fields
    - This is the most widely used approach for user libraries
  - See upcoming sumNumbers function

Optional Args: Details

- **You can call any function with any number of arguments**
  - If called with fewer args, extra args are undefined
    - You can use typeof arg == "undefined" for this
      - You can also use boolean comparison if you are sure that no real
        value could match (e.g., 0 and undefined both return true for !arg)
    - Use comments to indicate optional args to developers
      - function foo(arg1, arg2, /* Optional */ arg3) {...}
  - If called with extra args, you can use "arguments” array
    - Regardless of defined variables, arguments.length tells
      you how many arguments were supplied, and arguments[i]
      returns the designated argument.
    - Use comments to indicate varargs
      - function bar(arg1, arg2 /* varargs */) { ... }
Optional Arguments

```javascript
function convertString(numString, /* Optional */ base) {
    if (typeof base == "undefined") {
        base = 10;
    }
    var num = parseInt(numString, base);
    console.log("%s base %o equals %o base 10.", numString, base, num);
}
```

```
>>> convertString("1010");
1010 base 10 equals 1010 base 10.
>>> convertString("1010", 2);
1010 base 2 equals 10 base 10.
>>> convertString("2");
2 base 10 equals 2 base 10.
>>> convertString("2", 16);
2 base 16 equals 2 base 10.
```
Using JSON for Optional Arguments

• **Idea**
  - Caller always supplies same number of arguments, but one of the arguments is an anonymous (JSON) object
    • This object has optional fields
  - This approach is widely used in Prototype, Scriptaculous, and other JavaScript libraries

• **Example (a/b: required, c/d/e/f: optional)**
  - someFunction(1.2, 3.4, {c: 4.5, f: 6.7});
  - someFunction(1.2, 3.4, {c: 4.5, d: 6.7, e: 7.8});
  - someFunction(1.2, 3.4, {c: 9.9, d: 4.5, e: 6.7, f: 7.8});
  - someFunction(1.2, 3.4);

---

Using JSON for Optional Arguments: Example Code

```javascript
function sumNumbers(x, y, extraParams) {
    var result = x + y;
    if (isDefined(extraParams)) {
        if (isTrue(extraParams.logInput)) {
            console.log("Input: x=%s, y=%s", x, y);
        }
        if (isDefined(extraParams.extraOperation)) {
            result = extraParams.extraOperation(result);
        }
    }
    return(result)
}

function isDefined(value) {
    return(typeof value != "undefined");
}

function isTrue(value) {
    return(isDefined(value) && (value == true));
}
```
Using JSON for Optional Arguments: Example Results

```python
>>> sumNumbers(2, 3);
5
>>> sumNumbers(2, 3, {logInput: true});
Input: x=2, y=3
5
>>> function square(x) { return(x * x); }
>>> sumNumbers(2, 3, {logInput: true, extraOperation: square});
Input: x=2, y=3
25
```

Wrap-up
Summary

• General
  – Don’t try to universally use Java style when programming in JavaScript. If you do, you will see the bad features of JavaScript, but never the good features.

• Functions
  – Totally different from Java. Passing functions around and making anonymous functions very important.
    • Don’t think of this as rare or unusual, but as normal practice.

• Objects
  – Constructor defines class. Use “this”. Use prototype for methods.
    • Totally different from Java. Not like classical OOP at all.

• Other tricks
  – someFunction.apply(null, arrayOfArgs);
  – var someValue = eval("" + someString + ");
  – Various ways to do optional args. Object literals often best.

Questions?

Customized Java EE Training: http://courses.coreservlets.com/
Servlets, JSP, JSF 2.0, Struts, Ajax, GWT 2.0, Spring, Hibernate, SOAP & RESTful Web Services, Java 6.
Developed and taught by well-known author and developer. At public venues or onsite at your location.