JavaScript: Objects

For additional materials, please see http://www.coreservlets.com/. The JavaScript tutorial section contains complete source code for all examples in the entire tutorial series, plus exercises and exercise solutions for each topic.

coreservlets.com – custom onsite training

For customized training related to JavaScript or Java, email hall@coreservlets.com

Marty is also available for consulting and development support

Taught by lead author of *Core Servlets & JSP*, co-author of *Core JSF* (4th Ed), and this tutorial.

Available at public venues, or custom versions can be held on-site at your organization.

- **Courses developed and taught by Marty Hall**
  - JavaScript, jQuery, Ext JS, JSF 2.3, PrimeFaces, Java 8 programming, Spring Framework, Spring MVC, Android, GWT, custom mix of topics
  - Courses available in any state or country.
  - Maryland/DC companies can also choose afternoon/evening courses.
- **Courses developed and taught by coreservlets.com experts (edited by Marty)**
  - Hadoop, Hibernate/JPA, HTML5, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- Object basics
- The prototype property
- Namespaces (static methods)
- Anonymous objects
- JSON
- Adding methods to existing classes
- The instanceof and typeof operators
- Functions with variable numbers of arguments

Object Basics

Constructors

- **Constructors**
  - Functions named for class names
    - But not a new type as with most OOP languages
  - Define properties with "this"
    - You must use "this" for properties used in constructors
  - Use "new" to create instances of the "class"
- **Example**

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}
var p = new Person("Polly", "Programmer");
p.firstName; → "Polly"
p.lastName; → "Programmer"
```


Properties and Methods

- **Properties (instance variables)**
  - You don't *have* to define them in constructor
  - Whenever you refer to one, JavaScript just creates it

```
p.fullName = "Polly Programmer";
```

  - Usually better to avoid introducing new properties in outside code and instead do entire definition in constructor. But, it can happen accidentally:

```
p.firstname = "Pollyanna";  // Oops! Real property is firstName
```

- **Methods**
  - Properties whose values are functions, but see prototype property later

```
function Person(...) { ...
  this.fullName = function() {
    return(this.firstName + " " + this.lastName);
  };   ...
}
```

Objects: Example (Circle Class)

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

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

var c = new Circle(10);
c.radius → 10
c.getArea(); → 314.1592...
```

The getArea method works, but it would be better to use the prototype property. The next section explains why.

The prototype Property

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

Objects: Example (Previous Circle Class)

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

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

var c = new Circle(10);
c.radius → 10
c.getArea(); → 314.1592...
```

Objects: Example (Updated Circle Class)

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

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

var c = new Circle(10);
c.radius; → 10
c.getArea(); → 314.1592...
```


Using Namespaces to Avoid Name Conflicts

Static Methods (Namespaces)

- **Idea**
  - You have many related functions
  - 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 files or libraries
  - Similar to static methods in Java and other languages
- **Syntax**
  - Assign functions to properties of an object, but do not define a constructor

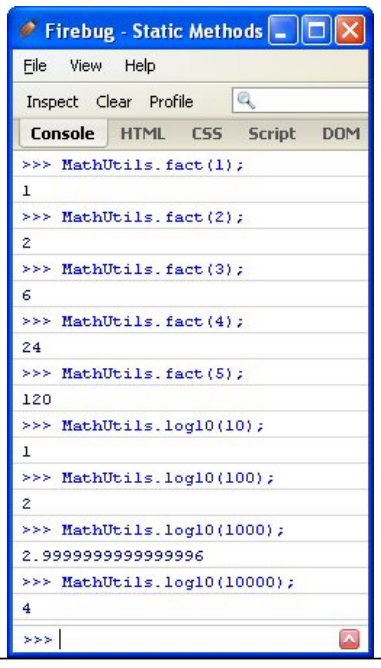
```
var Utils = {}; // Or new Object()
Utils.foo = function(a, b) { … };
Utils.bar = function(c) { … };
var x = Utils.foo(val1, val2);
var y = Utils.bar(val3);
```


Static Methods: Example (Code)

```
var MathUtils = {};

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

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



Namespaces in Real Applications

- **Best practices in large projects**
  - In many (most?) large projects, *all* global variables (including functions!) should be avoided to avoid 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.
- **Minor 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

Anonymous Objects

Anonymous Objects

- **A simple textual representation of JavaScript objects**
  - Called "object literals" or "anonymous objects"

```
var jane = { firstName: "Jane", lastName: "Doe" };
jane.lastName; → "Doe";
```

- **Main applications**
  - One-time-use objects (rather than reusable classes)
  - Arguments supplied to methods that use anonymous objects as a way of having variable argument lists.
    - Short summary coming up, but we will do this all the time with $.ajax later
  - Objects received from the server (as strings) in Ajax
    - JSON, not XML, is now the most popular way to send data from server to browser in Ajax application

Anonymous Objects: Example

This is not strict JSON (see next section)

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

Internet Explorer and Extra Commas

- **Firefox and Chrome tolerate trailing commas**
  - Tolerated in both arrays and anonymous objects

```
var nums = [1, 2, 3, ];
var obj = { firstName: "Joe", lastName: "Hacker", };
```

- **Many IE versions 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 when building JSON data on the server, because it is easier to just put commas after everything when you are using a loop to build the properties

JSON

Idea

- **Turning Strings into objects**
  - Can take anonymous object contained in a String and turn it into a regular JavaScript object
  - Strings that look like JavaScript objects are called JSON (JavaScript Object Notation) and are used by many languages (not just JavaScript) for exchanging data
- **The eval function**
  - Takes any String whose content looks like JavaScript and turns it into a real JavaScript object
    - This is what Firebug uses
- **The JSON.parse function**
  - Takes a string in a much more restricted format and turns it into a real JavaScript object
    - This is what jQuery uses when it gets data from the server

The eval Function

- **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"
```

Informal JSON (JavaScript Object Notation)

- **String whose content looks like JavaScript**
  - I.e., content of the string is the way you would type a data structure into JavaScript
- **Usage**
  - Making JavaScript command line (REPL – Read-Eval-Print-Loop). Firebug reads a String from an HTML page and passes it to eval. You can easily write a mini version of Firebug yourself!
  - Older raw JavaScript applications that used eval (before JSON.parse was added to the language) for the data from the server
  - Older jQuery applications (jQuery version 1.3 and earlier) for data from the server
  - Many JavaScript libraries still use eval for Ajax data, but strict JSON and the use of JSON.parse is strongly preferred for security and portability. Recent jQuery versions require strict JSON.

Object Literals, or Informal JSON (JavaScript Object Notation)

- **Using object literals directly in JavaScript**
  - Just type it into your code

```
var someObject = { property1: value1,
                   property2: value2,
                   ... };
```

- **Using JSON from a string (e.g., for data from server)**
  - If string contains object literals but is not strict JSON
    - Surround object representation in parens
    - Pass to eval
    - Do this only for your *own* data that you trust
  - If string is strict JSON (see upcoming slide)
    - Pass to JSON.parse
  - The approach with strict JSON and JSON.parse is strongly preferred in modern applications, but many older JavaScript libraries used eval

Strict JSON

- **Strict JSON according to json.org**
  - Subset of JavaScript where
    - Object property names must be in double quotes
    - Strings use double quotes only (not single quotes)
  - This is what recent jQuery versions and JSON.parse support.
  - Since this is what is clearly described at json.org, you should follow this format when sending JSON from the server.
- **MIME type for JSON from server**
  - RFC 4627 says JSON responses should have "application/json" MIME type
  - No known libraries enforce this

Strict JSON: Example

- **Not strict (OK for use directly in JavaScript)**

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

- **Strict (used for server data that will be passed to JSON.parse)**

```
var brendan2 = {
  "firstName":  "Brendan",
  "lastName":   "Eich",
  "bestFriend": { "firstName": "Chris", "lastName":  "Wilson" },
  "greeting": function() { return("Hi, I am " + this.firstName +
                                  " " + this.lastName + ".");  }
};
```

eval vs JSON.parse

- **eval**
  - Can parse any legal JavaScript expression that was represented inside a string
  - You must enclose object literals in parens (inside the string) before using eval
  - Should be used only for trusted data
- **JSON.parse**
  - Will only parse strict JSON
  - No extra parens needed
  - More secure and portable
  - Should be used for server data
- **JSON.stringify**
  - Will take object and turn it into strict JSON string

eval vs JSON.parse: Example

```
var janeString = '{ "firstName": "Jane" }';
var jane1 = JSON.parse(janeString);
jane1.firstName;
var jane2 = eval("(" + janeString + ")");
jane2.firstName;
```

Adding Methods to Existing Classes

Overview

- **Idea**
  - You can add methods to existing classes, including builtin classes (!)
    - This capability often causes Java programmers to become apoplectic
  - Use Classname.prototype.yourMethod and use "this" to refer to existing properties of the class
- **Simple but boring example**

```
String.prototype.describeLength = function() {
  return("My length is " + this.length);
};

"Any String".describeLength(); → "My length is 10";
```

More Useful Examples

- **Adding "sum" to arrays**

```
Array.prototype.sum = function() {
  function add(n1,n2) { return(n1 + n2); }
  return(this.reduce(add, 0));
}
[1,2,3].sum(); → 6
```

- **Adding "reverse" to strings**

```
String.prototype.reverse = function() {
  return(this.split("").reverse().join(""));
}
"abc".reverse(); → "cba"
```


The instanceof and typeof Operators

The instanceof operator

- **Idea**
  - Informally, determines if left side is a member of class on right side
  - Formally, tests if left side was created by constructor on right side
    - So very confusing for string literals and numbers, which always say false
- **Examples**

```
var myCircle = new Circle(10);
myCircle instanceof Circle; → true
[1,2,3] instanceof Array; → true
var obj = { firstName: "Joe" };
obj instanceof Object; → true
"foo" instanceof String; → false // Didn't use String constructor
(new String("foo")) instanceof String; → true
```

- **Typical usage**

```
if (blah instanceof Array) {
  doSomethingWith(blah.length);
}
```


The typeof operator

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

```
var testNumber = 3;
typeof testNumber; → "number"
var testString = "Hello";
typeof testString; → "string"
typeof bogusVariable; → "undefined"
```

Functions with a Variable Number of Arguments

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
    - 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

```
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);
  return(num);
}
```

```
› convertString("1010");
  1010 base 10 equals 1010 base 10.
  1010
› convertString("1010", 2);
  1010 base 2 equals 10 base 10.
  10
› convertString("11");
  11 base 10 equals 11 base 10.
  11
› convertString("11", 16);
  11 base 16 equals 17 base 10.
  17
```

Varargs

```
function longestString(/* varargs */) {
  var longest = "";
  for(var i=0; i<arguments.length; i++) {
    var candidateString = arguments[i];
    if (candidateString.length > longest.length) {
      longest = candidateString;
    }
  }
  return(longest);
}

longestString("a", "bb", "ccc", "dddd");  → "dddd"
```


Using Anonymous Objects 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 jQuery 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);
```

Wrap-up

For additional materials, please see http://www.coreservlets.com/. The JavaScript tutorial section contains complete source code for all examples in the entire tutorial series, plus exercises and exercise solutions for each topic.

Summary

- **Objects**
  - Constructor defines class. Use "this". Use prototype to define methods.

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

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

- **Other tricks**
  - var someValue = JSON.parse(someString);
    - String usually comes from server. Shown later.
  - Use objects as namespaces to avoid name conflicts

```
var Utils = {};
Utils.myFunction = function(…) { … };
```

coreservlets.com – custom onsite training 

Questions?

More info:
http://www.coreservlets.com/javascript-jquery-tutorial/ -- Tutorial on JavaScript, jQuery, and jQuery UI
http://courses.coreservlets.com/Course-Materials/java.html – General Java programming tutorial
http://www.coreservlets.com/java-8-tutorial/ – Java 8 tutorial
http://courses.coreservlets.com/java-training.html – Customized Java training courses, at public venues or onsite at *your* organization
http://coreservlets.com/ – JSF 2, PrimeFaces, Java 8, JavaScript, jQuery, Ext JS, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training
Many additional free tutorials at coreservlets.com (JSF, Android, Ajax, Hadoop, and lots more)

For additional materials, please see http://www.coreservlets.com/. The JavaScript tutorial section contains complete source code for all examples in the entire tutorial series, plus exercises and exercise solutions for each topic.