JavaScript: Functions

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

- Importance of functional programming
- Assigning functions to variables, arrays, and other data structures
- Passing functions to other functions
- Returning functions from functions
- Making anonymous functions
- Capturing local variables (making closures)
- Using the apply and call methods

Intro

"JavaScript has more in common with functional languages like Lisp or Scheme than with C or Java."
- JSON and YUI guru Douglas Crockford in article
"JavaScript: The World's Most Misunderstood Programming Language"

Getting Good at JavaScript

- **Experience with other languages can be an impediment here**
  - If you try to program JavaScript like Java 7 and earlier, you will *never* be good at JavaScript. Java 8 and later have functional programming capabilities, however.
  - Beginners or programmers from Ruby, Lisp, Scheme, Python, ML, Haskell, Clojure, Scala sometimes do better
- **Functional programming, not object-oriented programming, is key**
  - Functional programming is much more central to JavaScript programming than OOP is
  - Java programmers (at least prior to Java 8) find functional programming to be the single-hardest part of JavaScript to learn
- **OOP in JavaScript is radically different than in most languages**
  - So different in fact, that some argue that by some definitions of OOP, JavaScript does not have "real" OOP

Overview

- **You can have global functions**
  - Not just methods (functions as part of objects)
- **You don't declare return types or argument types**

```
function square(x) { return(x * x); }
```

  - You never say the input (x) or the output (x * x) are numbers
- **Functions are first-class datatypes**
  - You can pass functions around, store them in arrays, etc.
- **You can create anonymous functions**

```
function foo(...) {...}
var foo = function(...) {...}
```

  - You can also have anonymous functions that capture local variables (closures)
    - This capability is critical to Ajax (discussed in the jQuery section)
- **Functions can have methods**
  - Just as strings and arrays can

Assigning Functions to Variables, Arrays, and Other Data Structures

Functions are First-Class Data Types

- **Can assign functions to variables**
- **Can put functions in arrays (or other data structures)**
- **Can pass functions to other functions**
- **Can return functions from functions**
- **Can create functions without assigning them to variables (anonymous functions)**
- **Can create functions that capture local variables (closures)**
- **Functions can have methods, just as strings and arrays can**

"It is Lisp in C's clothing."
- Douglas Crockford describing the JavaScript language in *JavaScript: The Good Parts*

Assigning to Variables

```
function square(x) {
  return(x*x);
}

square(5); → 25

var f = square; // Not square() or square(5)

f(10); → 100
```


Putting Functions in Data Structures

- **Arrays**

```
function square(x) { return(x*x); }
var f = square;
function double(x) { return(x*2); }
var functs = [square, f, double];
functs[1](5); → 25
functs[2](5); → 10
```

- **Objects (object details covered later)**

```
var person = {
  firstName: "Harry",
  lastName: "Hacker",
  favoriteFunction: square
}
person.favoriteFunction(5); → 25
```

Passing Functions to Other Functions

Functions are First-Class Data Types

- Can assign functions to variables
- Can put functions in arrays (or other data structures)
- Can pass functions to other functions
- Can return functions from functions
- Can create functions without assigning them to variables (anonymous functions)
- Can create functions that capture local variables (closures)
- Functions can have methods, just as strings and arrays can

Simple Examples

```
function applyToSeven(funct) {
  return(funct(7));
}
applyToSeven(square); → 49
applyToSeven(double); → 14
function applyTwice(funct, value) {
  return(funct(funct(value)));
}
applyTwice(square, 5); → 625
function applyRepeatedly(funct, value, numRepeats) {
  for(var i=0; i<numRepeats; i++) {
    value = funct(value);
  }
  return(value);
}
applyRepeatedly(double, 5, 3); → 40
```


Review: Array Methods that Accept Functions

- **sort**

```
function difference(n1,n2) { return(n1-n2); }
function reverseDifference(n1,n2) { return(n2-n1); }
[1,3,5,4,2].sort(difference); → [1,2,3,4,5]
[1,3,5,4,2].sort(reverseDifference); → [5,4,3,2,1]
```

- **map**

```
[1,2,3,4,5].map(square); → [1,4,9,16,25]
```

- **filter**

```
function isEven(n) { return(n%2 == 0); }
[1,2,3,4,5].filter(isEven); → [2,4]
```

- **reduce**

```
function add(n1,n2) { return(n1 + n2); }
[1,2,3,4,5].reduce(add, 0); → 15
```

Applying the Array Methods

- **Goal**
  - Pass in function and array of values. Apply function to each value. Sum the results.
- **Alternative 1 (doing it by hand)**

```
function arraySum1(funct, values) {
  var sum = 0;
  for(var i=0; i<values.length; i++) {
    sum = sum + funct(values[i]);
  }
  return(sum);
}
```

arraySum1(square, [1,2,3,4,5]); → 55

- **Alternative 2 (using array methods)**

```
function arraySum2(funct, values) {
  function add(n1,n2) { return(n1 + n2); }
  return(values.map(funct).reduce(add, 0));
}
```

arraySum2(square, [1,2,3,4,5]); → 55

Returning Functions from Functions

Functions are First-Class Data Types

- Can assign functions to variables
- Can put functions in arrays (or other data structures)
- Can pass functions to other functions
- Can return functions from functions
- Can create functions without assigning them to variables (anonymous functions)
- Can create functions that capture local variables (closures)
- Functions can have methods, just as strings and arrays can

Example

```
function square(x) { return(x*x); }
function double(x) { return(x*2); }
function randomFunct() {
  if(Math.random() > 0.5) {
    return(square);
  } else {
    return(double)
  }
}
var ran = randomFunct();
ran(5) → 25 // Or, could be 10
ran(5) → 25 // Will always match value above
ran(5) → 25 // Will always match value above
ran = randomFunct();
ran(5) → 10 // Or, could be 25
```

Creating Anonymous Functions

Functions are First-Class Data Types

- Can assign functions to variables
- Can put functions in arrays (or other data structures)
- Can pass functions to other functions
- Can return functions from functions
- Can create functions without assigning them to variables (anonymous functions)
- Can create functions that capture local variables (closures)

Basics

- **Simple anonymous functions**

```
var square = function(x) { return(x * x); };
square(5); → 25
(function(x) { return(x * x); })(10) → 100
```

- **Equivalent constructs**

```
function square(x) {
  return(x * x);
}
var square = function(x) {
  return(x * x);
};
```

  - The bottom one is actually widely used in real life so that you can define functions within namespaces as shown in the upcoming section on objects.

```
var namespace = {};
var namespace.functionName = function(...) { ... };
```


Problem with randomFunction Function

- **Example from previous section**

```
function square(x) { return(x*x); }
function double(x) { return(x*2); }

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

- **Drawback: not self contained**
  - Relies on square and double already being defined
  - Changes in functionality if double and square are redefined

Redoing randomFunction

- **With local named functions**

```
function randomFunct2() {
  function square(x) { return(x*x); }
  function double(x) { return(x*2); }
  if(Math.random() > 0.5) {
    return(square);
  } else {
    return(double)
  }
}

var f = randomFunct2();
f(10); → 100 // Or maybe 20
(randomFunct2())(10); → 20 // Or maybe 100
```


Redoing randomFunction

- **With anonymous functions**

```
function randomFunct3() {
  if(Math.random() > 0.5) {
    return(function (x) { return(x*x); });
  } else {
    return(function (x) { return(x*2); })
  }
}

var f = randomFunct3();
f(10); → 100 // Or maybe 20
(randomFunct3())(10); → 20 // Or maybe 100
```

Capturing Local Variables (Making Closures)

Functions are First-Class Data Types

- Can assign functions to variables
- Can put functions in arrays (or other data structures)
- Can pass functions to other functions
- Can return functions from functions
- Can create functions without assigning them to variables (anonymous functions)
- Can create functions that capture local variables (closures)
- Functions can have methods, just as strings and arrays can

Function-Creating Function

- **With fixed value**

```
function makeTimes7Function() {
  return(function(n) { return(n*7); });
}
var f = makeTimes7Function();
f(5); → 35
```

- **With dynamic value**

```
function makeMultiplierFunction(m) {
  return(function(n) { return(n*m); });
}
var test = 8;
var f = makeMultiplierFunction(test);
f(5); → 40
test = 500;
f(5); → 40 // Not 2500. The closure has private copy of m (8)
```


The apply and call Methods

Functions are First-Class Data Types

- **Can assign functions to variables**
- **Can put functions in arrays (or other data structures)**
- **Can pass functions to other functions**
- **Can return functions from functions**
- **Can create functions without assigning them to variables (anonymous functions)**
- **Can create functions that capture local variables (closures)**
- **Functions can have methods, just as strings and arrays can**

The apply Method: Simple Use

- **Idea**
  - Lets you apply function to array of arguments instead of individual arguments.
  - apply is a method *of* functions!
    - someFunction.apply(null, arrayOfArgs);
  - See next slide for usage with an object instead of null for the first argument
- **Examples**

```
function hypotenuse(leg1, leg2) {
  return(Math.sqrt(leg1*leg1 + leg2*leg2));
}
hypotenuse(3, 4); → 5
var legs = [3, 4];
hypotenuse.apply(null, legs); → 5

Math.max.apply(null, [1, 3, 5, 7, 6, 4, 2]);  → 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(); → "undefined undefined"
var person = { firstName: "David", lastName: "Flanagan" };
fullName.call(person); → "David Flanagan"
```


Wrap-up

Summary

- **Can assign functions to variables**
  ```
  var f = square;
  ```
- **Can put functions in arrays (or other data structures)**
  ```
  var functs = [square, f, double];
  ```
- **Can pass functions to other functions**
  ```
  callIt(square, 7);
  ```
- **Can return functions from functions**
  ```
  return(square);
  ```
- **Can create functions without assigning them to variables**
  ```
  (function(x) { return(x*x); })(8); → 64
  ```
- **Can create functions that capture local variables (closures)**
  ```
  var f = makeMultiplierFunction(5);
  ```
- **Can use apply if arguments are already in array**
  ```
  Math.max.apply(null, [1, 3, 5, 7, 6, 4, 2]);  → 7
  ```