•  [Functions are first-class objects](http://www.dofactory.com/tutorial/javascript-function-objects#section1)

•  [Functions are copied by reference](http://www.dofactory.com/tutorial/javascript-function-objects#section2)

•  [Passing a function as a callback](http://www.dofactory.com/tutorial/javascript-function-objects#section3)

•  [What is currying?](http://www.dofactory.com/tutorial/javascript-function-objects#section4)

•  [What is memoization?](http://www.dofactory.com/tutorial/javascript-function-objects#section5)

**Functions are first-class objects**

In JavaScript, functions are objects. You can work with functions as if they were objects. For example, you can assign functions to variables, to array elements, and to other objects. They can also be passed around as arguments to other functions or be returned from those functions.

function message() {

alert("Greetings Linda!");

}

alert(typeof message); // => function

alert(message instanceof Object); // => true

We see that a function is indeed an object. JavaScript functions are a special type of objects, called function objects. A function object includes a string which holds the actual code -- the function body -- of the function. The code is literally just a string. Although not recommended, you can create a new function object by passing the built-in Function constructor a string of code, like so:

var body = "return Math.PI \* radius \* radius";

var circle = new Function("radius", body);

alert(circle(5)); // => 78.5398..

You can also create a new function with a custom constructor function (remember that by convention a constructor function always starts with an upper-case letter). In the code below we have a Book constructor function that is used to create book instances. In the constructor function we are assigning a function object to the getDetails property.

function Book (type, author) {

this.type = type;

this.author = author;

this.getDetails = function () {

return this.type + " written by " + this.author;

}

}

var book = new Book("Fiction", "Peter King");

alert(book.getDetails()); // => Fiction written by Peter King

The Book constructor accepts an argument which is assigned to a property named type. Once an object is created, you can assign property values and invoke methods just like any other object.

--%>

Function objects can also be created as part of an object literal. Below we create an object named circle with a property named area which is a function object.

var circle = {

radius : 10,

area : function() {

return Math.PI \* this.radius \* this.radius;

}

};

alert(circle.area()); // => 314.15..

alert(typeof circle.area); // => function

Next, let's look at an example where a function object is passed around like a regular object. The negate function takes a function as its argument. A new function is returned from the negate call that invokes the passed in function and returns the logical negation of its return value.

Following the function declaration we pass the built-in isNaN function to negate and assign the function returned to the isNumber variable. The variable isNumber is a function object that can be passed around like any object. To invoke the function you call it with different parameters.

function negate (f) {

return function (i) {

return !f(i);

};

}

var isNumber = negate(isNaN); // function object

alert(isNumber(5)); // => true

alert(isNumber(NaN)); // => false

alert(isNumber("A")); // => false

Here is another example using the same negate function. We added a custom function to test whether a number is prime or not (non-prime numbers are called composite). This function is passed to the negate function as an argument.

var isPrime = function (number) { // determines if number is prime

var divisor = parseInt(number/2, 10);

var prime = true;

while(divisor > 1) {

if (number % divisor === 0) {

prime = false;

divisor = 0;

} else {

divisor -= 1;

}

}

return prime === true;

};

function negate(f) {

return function (i) {

return !f(i);

};

}

var isComposite = negate(isPrime); // function object

alert([2, 4].every(isComposite)); // => false (2 is prime, 4 is not)

alert([4, 6].every(isComposite)); // => true (4 or 6 are composite)

Note: the built-in Array.every() method tests whether all elements in the array pass the test implemented by the function that is passed as an argument; in our case the isComposite function object.

**Functions are copied by reference**

When you assign a function object to another variable JavaScript does not create a new copy of the function. Instead it makes the new variable reference the same function object as original. It is just that two variables having different names are accessing the same underlying function object.

function original() {

// ...

}

original.message = "Hello";

var copy = original;

alert(original.message); // => Hello

alert(copy.message); // => Hello

copy.message = "Greetings";

alert(original.message); // => Greetings

alert(copy.message); // => Greetings

This example shows that if we add a property to the function object, then both variables, original and copy, will be changed because they are referencing the same function object. This confirms that function objects are indeed copied by reference.

Changing the function object's body

We just learned that function objects are copied by reference. However, when modifying the actual function body, things are a bit different because this will cause a new function object to be created. In the next example the original function body is changed and JavaScript will create a new function object.

function original() {

alert("I am Original");

}

var copy = original;

original(); // => I am Original

copy(); // => I am Original

original = function() { // Modify the original code

alert("I am Altered");

};

original(); // => I am Altered

copy(); // => I am Original

Assigning a new function body to original will create a new function object. Note that the copy variable still references the old function object.

**Passing a function as a callback**

Just like a regular object, you can pass a function object to another function (actually you've already seen this in action with the negate function example).

In the next example, two different functions, add and multiply, are passed as a parameter to the function action. Of course, only a reference to the function is passed. The functions add and multiply are called callbacks or more appropriately callback functions. The function action will call it back (i.e. invoke it) with the two argument values provided:

function action (callback, x, y) {

var result = callback(x, y);

alert(result);

}

function add(x, y) {

return x + y;

}

function multiply(x, y) {

return x \* y;

}

action(add, 2, 3); // => 5

action(multiply, 2, 3); // => 6

Callback functions play an important role in many frameworks, including JQuery. Consider the code below. The alert message executes well before the hide animation is complete, which is probably not what you want.

$("#blue").hide(2000);

alert("Animation complete.."); // executes before animation is complete

[Run](javascript:void(0);)  [Reset](javascript:void(0);)

This can be solved by passing in a callback function which will execute only when the animation is complete.

$("#red").hide(2000, function() {

alert("Animation complete.."); // executes after animation

});

[Run](javascript:void(0);)  [Reset](javascript:void(0);)

So, instead of waiting for a function to complete you can utilize callbacks to execute it asynchronously. This is beneficial for tasks that take some time to complete, such as the above hide animation. Another example is when executing an AJAX operation and you don't want the user to wait for the call to come back. This allows the browser to continue to respond to user requests while waiting for the callback function to be called.

Asynchronous programming is an important skill to have when working with JavaScript. To learn more about callbacks and the important event loop we suggest you check our unique [JavaScript + jQuery Design Pattern Framework](http://www.dofactory.com/products/javascript-jquery-design-pattern-framework) where we explore these and other topics in much greater detail. [Click here](http://www.dofactory.com/products/javascript-jquery-design-pattern-framework) for more details.

**What is currying?**

In situations where a function is repeatedly called with mostly the same arguments, you may have a candidate for currying. To curry a function is essentially caching and reusing argument values.

A curried function uses a closure to cache the state of recurring arguments so that you don't need to pass them every time. The new function then uses them to pre-populate the entire list of arguments that the original function requires.

The input to the currying process is a function that accepts two or more arguments. It then transforms the function to produce a new function that offers the similar functionality but with partial (fewer than the original) arguments. It binds the rest of the arguments to fixed values.

Let's look at an example. The function name accepts two arguments for first name and last name: first and last. It concatenates them to generate the person's name.

function name (first, last) {

return first + " " + last;

}

alert(name("Joni", "Mitchell"));

Next, we create a curried version of the same function. If we pass two arguments it executes normally, just like the example above. If, however, we only pass the first argument, then an another function is returned with its closure which holds the first value. The returned function accepts a partial list of arguments, in our example, just one argument which is last because it already knows what the first value is. Again, it performs the same job as name, but the value of first is remembered in the closure associated with the anonymous helper function that is returned.

function name (first, last) {

if (typeof last === 'undefined'){

return function(last) { // curry function, creates a closure

return first + " " + last;

};

}

return first + " " + last;

}

alert(name("Joni", "Mitchell")); // => Joni Mitchell

var partialName = name("Joni");

alert(partialName("Mitchell")); // => Joni Mitchell

alert(partialName("McNamara")); // => Joni McNamara

[Run](javascript:void(0);)

In JavaScript, a function needs a helper function to achieve currying. This helper function is commonly referred to as the curry function.

There is a generic implementation of currying that can be applied to any function and the code below demonstrates this. It creates a closure that stores both the original function and the arguments to curry. Then, when called again, it concatenates the incoming arguments with the prior array of arguments and executes the original function. One array holds the arguments passed to the current invocation and the other array holds the arguments passed to the curry function. The concatenated array is then passed to the original function.

Function.prototype.curry = function () {

var f = this;

var curryArgs = Array.prototype.slice.call(arguments);

// Return a function that returns the result

// of invoking the original function

return function () {

return f.apply(this, curryArgs.concat(

Array.prototype.slice.call(arguments)));

};

};

function show(message) {

alert(message);

}

var joke = show.curry("Two friends walk into a bar...");

joke(); // Two friends walk into a bar..

joke(); // Two friends walk into a bar..

joke(); // Two friends walk into a bar..

[Run](javascript:void(0);)

Function bind() and currying

<%--

All JavaScript functions have a method called bind that binds to an object and returns a new function. The first argument to bind sets the this context of the function.

function area (height) {

return this.width \* height;

}

var obj = { width : 5 };

var bound = area.bind(obj);

alert(bound(4)); // => 20

[Run](javascript:void(0);)

Calling bound(4); invokes the original function area as a method of obj, like obj.area(4);. The argument you pass to bound is passed as the height argument to the function area.

In addition to binding a function to an object,--%>

EcmaScript 5 supports a bind method that brings native currying to JavaScript. You no longer need to use a curry helper function. The arbitrary number of arguments that you pass to bind are also bound.

function add(x, y, z) {

return x + y + z;

}

var partial = add.bind(null, 1, 2);

var result = partial(3); // pass 3 for the z argument

alert(result); // => 6

This creates a new function called partial. The this value is bound to null, i.e. the global object, and the x and y arguments are bound to 1 and 2 respectively. Calling partial with the argument value 3 binds this value to z and then executes the add function without the need to write a curry function.

Next, let's look at a practical application of currying in the area of unit conversions:

function unitConversion(toUnit, factor, offset, input) {

offset = offset || 0;

return [((offset + input) \* factor).toFixed(2), toUnit].join(" ");

}

var meterToKm = unitConversion.bind(null, 'km', 0.001, 0);

var kgToGram = unitConversion.bind(null, 'grams', 1000, 0);

alert(kgToGram(3.7)); // => 3700 grams

alert(meterToKm(2000)); // => 2 km

The unitConversion function performs the actual unit conversion computations. Look at bind and, again, the null argument refers to the global object to be used as this. The next three arguments are bound to toUnit, factor, and offset respectively. These bindings are stored and maintained in the closure that is associated with the meterToKm and kgToGram function objects. In the last two lines we call each function with a value which is then bound to the input variable and the original unitConversion function is called returning the desired results.

<%--

Caching previously computed results as properties of the function itself

Suppose you are building a car racing game in JavaScript and you need to keep track of the total number of car objects that have been instantiated. In Java, you would use the static keyword for this, but JavaScript does not offer such functionality out-of-the-box.

Of course, you could simply store this piece of data in a global variable but this would add unnecessary variables to the global namespace. A better solution is to store this information in a property of a function object. Let's call the function carCount and use it to memorize the next value to be returned. Then the function would be able to return a different value each time it is invoked.

function carCount() {

var count = 0;

var increment = function() {

return ++count;

}

return increment;

}

alert(carCount()); // => 1

alert(carCount()); // => 2

alert(carCount()); // => 3

alert(carCount()); // => 4

In functional programming, caching the results of a function call is referred to as memoization. Let's explore this in a little more detail.

**What is memoization?**

Memoization is an optimization technique that is used to improve the performance of a function by caching its return values so that it does not need to redo the potentially heavy computations next time it is called.

Not all functions can be memoized; only referential transparent functions. A referential transparent function is one that always produces the same output on a given input. For instance, if you invoke a function with a value x passed to it, it will perform calculations on x and always returns the same associated y value.

Good examples of where memoization can be beneficial are HTML5 Canvas animations and recursive mathematical calculations, such as, factorial computation, generating Fibonacci sequences, and matrix chain multiplications.

Let's look at how memoization can be used to improve computing the Fibonacci numbers. The recursive approach to generating these numbers does not scale very well. In the example below the getFibonacci function is recursively called 177 times to generate the sum of the first 10 Fibonacci numbers:

function getFibonacci (num) {

if (num < 2) {

return num;

}

return getFibonacci(num - 1) + getFibonacci(num - 2);

}

alert(getFibonacci(10)); // => 55 (with 177 iterations)

[Run](javascript:void(0);)

The program does a lot of extra work by not keeping track of previously calculated values. This is where memoization comes in. First, declare a cache array where you can store the already-calculated function values that were returned in previous calls. Then, instead of invoking the function, return these values in subsequent calls to the function, like so:

function getFibonacci (num) {

var cache = [];

var fib = function(value) {

if(value < 2 ) return value;

if( cache[value] ) return cache[value];

cache[value] = (fib(value -1)) + (fib(value-2));

return cache[value];

};

return fib(num);

}

alert(getFibonacci(10)); // => 55 (with 20 iterations)