CSCE 120: Learning To Code

Module 9.0: Organizing Code I
Introduction to Functions

This module is designed to get you started working with functions which allow you to organize code into reusable units promoting proper *procedural abstraction* in your design.

Introduction

In programming languages, a *function* is a sequence of instructions (code) that is packaged into a unit. A function performs a specific task: given a number of inputs, it performs some operation (executes some code) and "returns" (outputs) a result.

A function is usually defined by its *signature*: every function has a name (also called an *identifier*), a list of *input parameters*, and an output. A programming language function is very much like a mathematical function, $f(x)$: x is the input, f is the name of the function, and when evaluated, it produces a value (output).

Defining and using functions in a programming language offers many clear advantages. The most obvious advantage is that it allows you a way to *organize* code. By separating a program it into distinct units code is more organized and it is clearer what each piece of code does. This also facilitates *top-down design*: one way to approach a problem is to split it up into a series of subproblems until each subproblem is either trivial to deal with, or an existing, "off-the-shelf" solution already exists for it.

Another advantage is that by putting code into functions, those functions can be *reused* in multiple places either in your program/project or even in other programs/projects. A prime example of this is the math library which has defined several useful mathematical functions and is used by thousands of programs across multiple different platforms.

Placing code into functions also allows for better and easier *testing*. By isolating pieces of code, we can rigorously test those pieces of code by themselves without worrying about the larger program or contexts.

Finally, functions (sometimes called subroutines or procedures) facilitates *procedural abstraction*. Placing code into functions allows you to abstract the details of how the function computes its answer. As an example: consider the `Math.sqrt()` function: it may use some interpolation method, a Taylor series, or some other method entirely to compute the square root of a given number. However, by putting this functionality into a function, we, as programmers, do not need to concern ourselves about the details. Instead, we simply *use* the function, allowing us to focus on the larger issues at hand in our program.

Functions in JavaScript

JavaScript supports the ability to define and use functions.

Defining Functions

In JavaScript, we define a function using the keyword `function`. We also need to provide a name for the function and a list of its parameters (input values that will be passed to this function as "arguments"). An example:

```
/**
 * This function computes the annual percentage yield given
 * an annual percentage rate (apr).
 */
function annualPercentageYield(apr) {
  var apy = Math.pow((1 + (apr / 12)), 12) - 1;
  return apy;
}
```

The name of this function is `annualPercentageYield` and it takes one parameter. Function names follow the same rules as variables as well as the same conventions: your function names should describe what they do and should be lower camel-case. Function names, like variables, must be unique: you cannot have two different functions with the same name. Finally, declaring a function as above gives the function a *global scope* which means that any other piece of code will be aware of the function and will be able to use it.

Parameters

You can define as many parameters in your function as you like, even none. Multiple parameters are separated by a comma between the parentheses. The following example takes three parameters. Each parameter is like a variable and must follow the same naming rules and conventions. Inside the function you can use the parameters just like

variables in any expression that you write.

```
function getMonthlyPayment(principle, apr, terms) {
  var rate = (apr / 12);
  return (principle * rate) / (1-Math.pow(1+rate, -terms));
}
```

The value passed to the function is the value that parameters take inside the function. This is known as *pass-by-value*. In detail, a *copy* of the variable's value passed to the function is provided to the function. You can change the value of a parameter, but it will have no effect on the variable that was passed to the function. An example:

```
function foo(a) {
  a = 10;
  return 20;
}
...
var x = 5;
var y = foo(x);
/* at this point, x retains its value of 5; inside foo, only the
   parameter was changed x and a are completely different
   variables so changes to one will not affect the other
*/
```

A slight exception to this is when you pass an object to the function. Changes to the object's fields *will* be reflected in the calling function. Another example:

```
function foo(s) {
  s.firstName = "James";
  return;
}
...
var student = {
  "firstName": "John",
  "lastName": "Student"
};
foo(s);
//at this point, student.firstName has a value of James
```

Return Values

The purpose of a function is that you can provide it with inputs and it will give you an output; the result of its computation. This is known as the return value and can be specified using the `return` keyword. Returning a value immediately halts the execution of the function and returns the specified value. The value can be a variable, a constant, or even *nothing at all* (example: `return;`). A function that does not return a value is

referred to as a *void* function.

Calling

Once a function is defined we can use it by calling it or *invoking* it. This is done by providing the function's name and any parameters. When calling a function, we sometimes refer to the parameters as *arguments*. Some examples:

```
var x;
x = getMonthlyPayment(10000, 0.05, 60);
...
var a = 10000, b = 0.05, c = 60;
x = getMonthlyPayment(a, b, c);
```

Optional Parameters

In JavaScript it is possible to call a function and *not* pass some of the parameters. For example, the following function calls are all legal and will execute.

```
y = getMonthlyPayment(a, b, c);
y = getMonthlyPayment(a, b);
y = getMonthlyPayment(a);
y = getMonthlyPayment();
```

If you omit a parameter when invoking a function, the value of the parameter is treated as `undefined` inside the function. Undefined variables result in undefined results when used in expressions. This behavior is actually a feature: you can write functions where some of the parameters are *optional*. You can check whether or not a passed parameter is undefined by using something like the following.

```
if(a === undefined) { ... }
```

And then provide "default" values or alternate behavior. We've previously seen this behavior in jQuery: many of the functions allow different behavior depending on how many arguments you provide it. For example `.val()` with no arguments, returns the value of the form element while providing an argument, say `.val("0.00")` sets the value of the web form element. A full example:

```
/**
 * Computes the monthly payment given a principle amount,
 * apr (default: 5%) and terms (default: 60 months)
 */
function getMonthlyPayment(principle, apr, terms) {
  if(apr === undefined) {
    apr = 0.05;
  }
  if(terms === undefined) {
    terms = 60;
  }
  var rate = (apr / 12);
  return (principle * rate) / (1-Math.pow(1+rate, -terms));
}
```

In this example, we've provided default values for the last two parameters, but the first value is required. Calls to this function will have the following behavior.

```
var x;
x = getMonthlyPayment(10000, 0.05, 60); //188.71
x = getMonthlyPayment(10000, 0.05); //188.71
x = getMonthlyPayment(10000); //188.71
x = getMonthlyPayment(); //undefined
```

Functions as Members of Objects

In all of the previous examples, the functions we defined were *global*. That is, the function could be "seen" and called by any other code block and could be called by any other piece of code. In JavaScript, functions can also be members of an object. An example:

```
var FinancialUtils = {
  getMonthlyPayment: function(principle, apr, terms) {
    var rate = (apr / 12);
    return (principle * rate) / (1-Math.pow(1+rate, -terms));
  },
  annualPercentageYield: function(apr) {
    var apy = Math.pow((1 + (apr / 12)), 12) - 1;
    return apy;
  }
};
```

Note the difference in syntax here. The function identifier is placed first, followed by a colon and the `function` keyword. This should look familiar: this is exactly how functions in the standard Math library are organized. In the example above, the two functions are now members of `FinancialUtils` which is global. Invoking these functions now requires that you "go through" the `FinancialUtils` object.

```
var x = FinancialUtils.getMonthlyPayment(10000, 0.05, 60);

//the following is invalid: getMonthlyPayment() does not exist
x = getMonthlyPayment(10000, 0.05, 60);
```

Callbacks

Functions can call other functions. For example, the `getMonthlyPayment` function calls the math library's `exp` function. By combining functions like this, we can create more complex programs. It also increases abstraction as we reuse pieces of code inside other pieces of code.

As another example, consider the problem of computing the average of an array of numbers. We could write a function that sums up the elements of the array and then divides by its size. However, what if we already had a function to compute the sum of elements in an array?

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

function ave(arr) {
  var sum = 0;
  for(var i=0; i<arr.length; i++) {
    sum += arr[i];
  }
  return (sum / arr.length);
}
```

Having two separate functions with such closely related functionality is a waste of code: the first 4 lines are identical. Instead, we could save code and simplify this by reusing the `sum` function:

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

function ave(arr) {
  var sum = sum(arr);
  return (sum / arr.length);
}
```

Callbacks

Not only can functions call other functions, but functions themselves can be *passed* to other functions as parameters. When passed as a parameter, a function is referred to as a *callback*. When passing a function A to another function B, the expectation is that B will call A (*call* it *back*) for its computation.

As a motivating example, consider again the problem of computing the average of a list of numbers. Instead of a list of numbers, consider a list of objects representing courses. Suppose each course contained a count of the number of students enrolled in the course and we wanted to compute the average number of students per course. To do this we may write a function similar to the following.

```
1 function averageEnrollment(courses) {
2   var sum = 0;
3   for(var i=0; i<arr.length; i++) {
4     sum += courses[i].count;
5   }
6   return (sum / courses.length);
7 }
```

But this is essentially the same code as our original `ave` function with the exception of line 4. What if, instead, we wrote a general `average` function that is given both an array and a callback that can be used to compute the sum of the elements in an array. For example,

```
function average(arr, sumFunc) {
  var sum = sumFunc(arr);
  return (sum / arr.length);
}
```

Here, the second argument is a callback (a function) that we invoke on the array to find the sum. This function is now generic: it can be invoked on an array of *anything* because the logic to sum elements is encapsulated inside the `sumFunc` function. Now that we

have a generalized average function, we can write a specialized function to compute the sum for an array of courses.

```
function sumOfEnrollments(courses) {
  var sum = 0;
  for(var i=0; i<courses.length; i++) {
    sum += courses[i].count;
  }
  return sum;
}
```

We can then invoke the general average function as follows:

```
var courses = [
  {name: "CSCE 120", count: 27},
  {name: "CSCE 220", count: 15},
  {name: "MATH 103", count: 89}];

var x = average(courses, sumOfEnrollments); //43.66
```

We can now reuse `average` for any array of any type and we only need to worry about writing code to sum values. We could further generalize the sum array (or use the array function `reduce()`).

We will revisit these concepts when we explore how to organize data by sorting and searching.

Anonymous Functions

In the previous examples, all of our functions had identifiers. There is another way to define functions where the functions do not have a name. They are then known as *anonymous* functions since they have no name. There is nothing particularly special about doing this: strictly speaking, we don't *need* anonymous functions, but they allow us to use a software programming *pattern* whereby we can define and pass a function to another function all in the same line.

As an example, consider the `sumOfEnrollments` function from before. It had only one use: to compute the sum of enrollments of an array of course objects. As such, the function didn't really have much use beyond creating it and handing it off to the `average` function. There is little reason for giving it a name if it is not going to be reused. Instead, we could have created it in-line when we called the `average` function:

```
var x = average(courses, function(courses) {
  var sum = 0;
  for(var i=0; i<courses.length; i++) {
    sum += courses[i].count;
  }
  return sum;
});
```

In this example, we defined a function, but did not give it a name (we made it anonymous) and immediately passed it to the `average` function to be used as a callback. This is a common pattern that allows us to pass anonymous functions as callbacks.

Asynchronous Computing

Passing callbacks is not just a convenience: its a paradigm extensively used in *asynchronous computing*. The performance of many computing operations are bounded by input/output operations. That is, most of the time, the processor is sitting idle waiting for input/output to complete. By programming with callbacks, we can (potentially) resume normal execution without having to wait for I/O.

For example, we could invoke a function that makes a connection to a remote server for data, then retrieves the data and processes it. This could potentially take a noticeable amount of time to a user especially if the network connection is slow and/or there is a lot of data to transfer. It would negatively impact the user experience (UX) if the rest of the page/application had to wait for the data to be transferred before anything else could be done. The user would essentially see the application as "freezing" while it waited for the data. Instead, we would like to continue normal operation without waiting for this process to complete. To do this, we provide the function a callback to use when it is done executing and we can continue with our normal operation.

As we've already seen, this pattern is used extensively in the jQuery library. In particular,

- The `.each()` function takes a callback that it applies to each element in the set
- The `ready()` function takes a callback that is executed with the page is loaded and ready

Pitfalls

Understanding asynchronous operations can be difficult. In particular, problems can arise when two operations "race" each other. Consider the following jQuery examples that attach an event to a button. The intention is that when the button is clicked, the first paragraph element is slowly hidden then reappears.

```
//example 1:
$("button").click(function(){
  $("#firstPara").hide("slow");
  $("#firstPara").show();
});

//example 2:
$("button").click(function(){
  $("#firstPara").hide("slow",function(){
    $("#firstPara").show();
  });
});
```

The first example, however, will fail. When the `hide()` function is invoked, it takes some time (1.5 seconds) to execute. However, normal execution inside our function continues and the `show` function is called before the `hide` function completes making it have no effect.

The second example correctly achieves our animation. What we really wanted was a *certain sequence* of events to fire, one after the other. To do this, we wrap our animation (callback) within another callback, etc. When the button is clicked, the first paragraph is hidden, but slowly. Only after the animation has been completed does the second callback execute and the paragraph is shown again.