

Computer Science & Engineering 120

Learning to Code

Organizing Code I – Functions

Christopher M. Bourke
cbourke@cse.unl.edu

Part I: Introduction to Functions

Topic Overview

- ▶ Why Functions?
- ▶ Defining Functions
- ▶ Using Functions

Functions

- ▶ Functions are units of code with *inputs* that produce an output
- ▶ Provide code *organization*
- ▶ Functions provide a way to *reuse* code
- ▶ Easier to test, maintain, etc.
- ▶ Also provides for *procedural abstraction*
 - ▶ Logic/process is *encapsulated* inside a function
 - ▶ We don't need to worry about the details of how a computation is executed
 - ▶ We just *use* it
 - ▶ Example: `Math.sqrt()`

Defining Functions

- ▶ Define functions by using the `function` keyword
- ▶ Function needs:
 - ▶ A name (identifier)
 - ▶ A list of parameters (inputs)
 - ▶ A body

```
1 function milesToKm(miles) {  
2   var km = miles * 1.60934;  
3   return km;  
4 }  
5  
6 function format(firstName, lastName) {  
7   return lastName + ", " + firstName;  
8 }
```

Function Parameters

- ▶ Function parameters are essentially variables available to the function
- ▶ Can (but shouldn't) redefine them
- ▶ Can be used in an expression
- ▶ Must follow same identifier rules
- ▶ Multiple parameters separated by a comma

Return Values

- ▶ Functions compute some result and need to communicate it back to the code that called it
- ▶ Use the keyword `return`
- ▶ A function may return any type
- ▶ A function doesn't have to return anything (if it doesn't, it's called a "void" function)
- ▶ Forgetting (or omitting) a return value will end up returning an `undefined` value

Using Functions

- ▶ You call a function as we've been doing: providing values (or variables) as *arguments*, storing the return value

```
1 var x = 2;
2 var y;
3 y = Math.sqrt(x);
4
5 var m = 252.4;
6 var k;
7 k = milesToKm(m);
8 k = milesToKm(100.2);
9
10 var name = format("Chris", "Bourke");
```

Passing By Value I

- ▶ Values stored in variables are *copied* and passed to the function for processing
- ▶ The function knows nothing about the original variable
- ▶ Changes to the function parameters have no effect on the original variable
- ▶ Demonstration

Passing By Value II

```
1 function test(a, b) {
2   a = 10;
3   console.log("a = " + a);
4   var c = a + b;
5   return c;
6 }
7
8 var x = 5;
9 var y = 15;
10 var z = test(x, y);
11 console.log("x = " + x + ", y = " + y + ", z = " + z);
12 //prints 5, 15, 25
```

Optional Parameters I

- ▶ When calling a function, passing argument(s) is *optional*
- ▶ If an argument is not passed to a function, the parameter's value becomes `undefined`
- ▶ Example
- ▶ This can be used as a *feature*: we can define functions with *optional* parameters either:
 - ▶ Provide sensible default values or
 - ▶ Change the behavior/meaning of the function based on the parameters
- ▶ Example: jQuery's `css()` function: one parameter *gets* the value, two parameters *sets* the parameter
- ▶ Check if a parameter is provided by using `x === undefined`

Optional Parameters II

```
1 function min(a, b) {
2   if(a < b) {
3     return a;
4   } else {
5     return b;
6   }
7 }
8
9 var x = 10;
10 var y = 20;
11 var m;
12 m = min(x, y); //10
13 m = min(x); //undefined
14 m = min(); //undefined
```

Functions as Object Members I

- ▶ Declaring a function makes it *globally scoped*
- ▶ Every piece of code can “see” it and use it
- ▶ This has potential to “pollute the namespace”
- ▶ If two libraries both defined a function `showPopup()`, they would be in conflict
- ▶ Solution: organize functions into objects as members
- ▶ Just like `Math` library

Functions as Object Members II

```
1 var MyFunctions = {
2   min: function(a, b) {
3     if(a < b) {
4       return a;
5     } else {
6       return b;
7     }
8   },
9   milesToKm: function(miles) {
10    var km = miles * 1.60934;
11    return km;
12  }
13 };
```

Part II: Callbacks

Topic Overview

- ▶ Functions calling functions
- ▶ Functions as variables & parameters
- ▶ Anonymous Functions
- ▶ Asynchronous Computing

Functions Calling Functions I

- ▶ Functions can call other functions
- ▶ When a function is called, control flow is handed over to the function until it completes
- ▶ After it completes, control is handed back to the calling function
- ▶ Such function calls are *synchronous*

Functions Calling Functions II

```
1 function bar(a) {
2   console.log("bar = " + a);
3 }
4
5 function foo() {
6   bar(10);
7   console.log("foo");
8   bar(20);
9 }
10
11 foo();
```

Functions as Parameters I

- ▶ Variables can hold numbers, strings, objects, arrays, etc.
- ▶ Variables can *also* hold functions!
- ▶ A function's "value" is its name
- ▶ This allows you to *pass* a function to another function!

Functions as Parameters II

```
1 function foo() {
2   ...
3 }
4
5 function bar(x, someFunction) {
6   ...
7 }
8
9 var myFunc = foo;
10 bar(10, myFunc);
11 //or
12 bar(10, foo);
```

Functions as Parameters III

- ▶ The passed function is called a *callback*
- ▶ This allows us to write more generic, general code
- ▶ Example: `forEach()` or jQuery's `$.each()` function
- ▶ Example: Sorting
- ▶ Callbacks are used extensively in jQuery: you can call a function and provide another callback that you want called *after* the completion of the function

Functions as Parameters IV

```
1 function foo(x) {
2   console.log("x = " + x);
3 }
4
5 function bar(x, someFunction) {
6   console.log("bar: " + x);
7   //call the passed function "back"
8   someFunction(x);
9 }
10
11 bar(10, foo);
```

Anonymous Functions I

- ▶ If the only purpose to a function is to pass it off to another function as a callback, there is no need to "pollute the namespace" by declaring the function with a name
- ▶ Alternative: define the function "inline" without a name and immediately pass it to another function
- ▶ Called an *anonymous function*

Anonymous Functions II

```
1 function bar(x, someFunction) {
2   console.log("bar: " + x);
3   //call the passed function "back"
4   someFunction(x);
5 }
6
7 //there is no function foo, just an anonymous
8 //one that does the same thing
9 bar(10, function(x) {
10   console.log("x = " + x);
11 });
```

Asynchronous Computing

- ▶ Some function may execute “long”-running procedures such as making a network connection to get data
- ▶ Don't want these processes to freeze (to “block”) the rest of the application
- ▶ Freezing while waiting would give a bad User Experience (UX)
- ▶ Solution: make some functions *asynchronous*
- ▶ Execution doesn't block the rest of the application
- ▶ We won't go into detail and in fact even with ES6 our ability to do asynchronous computing is limited
- ▶ Pitfall: care needs to be taken to *chain* callbacks appropriately
- ▶ Example demonstration

Part III: Exercises

Exercise

- ▶ Develop a function to round a number to the nearest cent (nearest 100th)
- ▶ Generalize this function so that it supports rounding to any decimal place
- ▶ Rewrite the first function to utilize this function
- ▶ Organize your functions into a utility class