Introduction

We've talked about a few different variable types, and one of the things you may not be aware of is that in JavaScript a function can also be treated as a variable type. This is one of the reasons JavaScript is also considered a functional language.

Functions in the Global Scope

Functions that we declare in programs outside of any other scope live in the global scope. When you declare a function using the `function` keyword, you're adding it to the global scope, the same way that you'd be adding a variable to the global scope if you declared it outside of any other containing scope.

```
function greeter(str, arr) {
  var counter;
  for (counter = 0; counter < arr.length; counter++) {
    console.log(str + " " + arr[counter]);
  }
}

console.log(window.greeter);
// the code of the function greeter

window.greeter("Hello", ["Fred", "Judy"]);
// Hello Fred
// Hello Judy
```

Protecting the Global Scope

Just as with variables, it's important to protect the global scope from the functions that you declare that are specific to your program. Conveniently, all of the tricks that we have used for protecting the global scope from variables also apply to functions. I'll give you an example, using the immediately invoked function expression.

```
(function() {
  function greeter(str, arr) {
    var counter;
    for (counter = 0; counter < arr.length; counter++) {
      console.log(str + " " + arr[counter]);
    }
  }

  greeter("Hello", ["Fred", "Judy"]);
  // Hello Fred
  // Hello Judy
}());

console.log(window.greeter);
// undefined

window.greeter("Hello", ["Fred", "Judy"]);
// error TypeError: window.greeter is not a function
```

This is one of the main reasons why many programmers today write their programs inside of immediately invoked function expressions, so that they can protect the global scope.

Functions Declared in Functions

Functions are also protected from being in the global scope if they're declared in the context of another function. It's interesting to note that a function can declare local functions within their own local scope, just the same way that an immediately invoked function expression can declare a function inside of its local scope.

Another good reason to do this is if you're creating a function that's really only needed within the scope of a particular function.

```
function greeter(str, arr) {
  var counter;
  var phrase;
  for (counter = 0; counter < arr.length; counter++) {
     phrase = str + " " + arr[counter];
     console.log(shout(phrase));
  }

  function shout(words, emphasis) {
    var punctuation = emphasis;
    if (!punctuation) {
      punctuation = "!";
    }
    return(words.toUpperCase() + punctuation);
  }
}

greeter("Hello", ["Fred", "Judy"]);
// "HELLO FRED!"
// "HELLO JUDY!"
```

Passing Functions to Functions

I mentioned that functions can be treated as values, and that means that you can actually pass a function to a function. A function can take an anonymous function, which can be declared right in the arguments that you're passing to that function, and you don't even need to give it a name. It'll still work.

```
function greeter(str, arr, display) {
  var counter;
  var output;
  for (counter = 0; counter < arr.length; counter++) {
    output = display(str + " " + arr[counter]);
    console.log(output);
  }
}

greeter("Hello", ["Fred", "Judy"], function(str) {
  return(str);
});
// Hello Fred
// Hello Judy

greeter("Hello", ["Fred", "Judy"], function(str) {
  return(str.toUpperCase() + "!!!");
});
// "HELLO FRED!!!"
// "HELLO JUDY!!!"
```

Using inline anonymous functions this way is very common in event driven JavaScript. One common example is known as a callback, in which you pass a function to a function, and the function that you pass in is something that you want to have invoked after everything else inside of the function has been executed.

Using this approach you can decide which function you want invoked at the time that you make that call and pass it in as a parameter, along with all the other parameters.

Assigning Functions to Variables

Since I've demonstrated that functions can be treated as values, I want to show you one more example of a variable type because we can actually assign an anonymous function to a variable, just like any other value. This is one of my preferred ways of defining a function.

```
var greeter = function(str, arr) {
  var counter;
  for (counter = 0; counter < arr.length; counter++) {
    console.log(str + " " + arr[counter]);
  }
};

greeter("Hello", ["Fred", "Judy"]);
// Hello Fred
// Hello Judy
```

So what we've done here is we've taken the name of the function, and made it the name of a variable. And we've assigned the value of that variable to be a anonymous inline function that operates on a string in an array and does all of the same things the greeter does. This whole block now becomes a single statement in JavaScript.

Since both functions and variables affect the global scope, defining your functions this way can help remind you to keep the global scope free of both functions and variables.

Treating Functions as Other Variables

Once you start treating your functions just like other variables in JavaScript, you'll start recognizing that you can pass a function to another function as a variable, instead of declaring it inline, and that makes for cleaner, more modular code. Let's go back to our greeter example and show how formatting our code using functions that have been assigned to variables, to format our strings, can make our code more clean.

```
var greeter = function(str, arr, display) {
  var counter;
  var output;
  for (counter = 0; counter < arr.length; counter++) {
    output = display(str + " " + arr[counter]);
    console.log(output);
  }
};

var say = function(str) {
  return(str);
};

var shout = function(str) {
  return(str.toUpperCase() + "!!!");
};

greeter("Hello", ["Fred", "Judy"], say);
// Hello Fred
// Hello Judy

greeter("Hello", ["Fred", "Judy"], shout);
// "HELLO FRED!!!"
// "HELLO JUDY!!!"
```

So now, we've got a function assigned to the variable `say`, which we can pass in to `greeter`, and that's much cleaner than we had it before, with the full function defined inline inside of the greeter call.

You can see how convenient it is to be able to define functions like this which you might be able to use in different contexts. By creating small named functions assigned to variables,you can create code that's very reusable.