

10. More on JavaScript Functions

Nesting Function Definitions

Function definitions can be nested.

Example:

```
<html>
<body>
<script type = "text/javascript">
  function hypotenuse(a, b)
  {
    function square(x)
    {
      return x * x;
    }
    return Math.sqrt(square(a) + square(b));
  }

  document.writeln( hypotenuse(3, 4));
</script>
</body>
</html>
```

5

JavaScript is statically (or lexically) scoped:

A non-local variable in the current block is identified with the variable with that name in the lowest block where the variable is declared that encloses the current block.

Functions as Data

Functions are objects – they're data that can be assigned to variables.

Example:

```
<script type = "text/javascript">
  function square(x)
  {
    return x * x;
  }

  var sqr = square;
  document.writeln( sqr(4) );
</script>
```

16

We can define a function with the `Function()` constructor.

It expects any number of string arguments.

The last is the body of the function.

The rest specify the arguments of the function.

Example:

```
var f = new Function( "x", "y",
                    "return x * y;" );
```

Somewhat similarly, predefined function `eval` takes a string argument and evaluates it.

For example, if `x` is 3 and `y` is 2, then

```
eval( "x + y" )
```

evaluates to 5.

A *function literal* (defining an *anonymous function*) is like a function statement but is used as an expression and no function name is specified.

Example:

```
var f2 = function(x, y)
    {
        return x * y;
    };
```

The advantage of the `Function()` constructor is that we can construct the body of the function at run-time using the usual ways to manipulate strings.

Example:

```
<script type = "text/javascript">
  var f1 = new Function("x", "y",
                        "return x * y;"),
      f2 = function(x, y)
          {
              return x * y;
          };

  document.writeln( f1(3, 4), " ", f2(2, 5) );
</script>
```

12 10

Example:

This shows the things that can be done when functions are used as data.

It shows that functions can be passed as arguments to other functions.

And it shows how functions can be stored in an associative array (explained later).

- An array is an object, which inherits from the `Object` prototype (the highest prototypical object in the inheritance hierarchy of objects).

All objects (not just arrays) are passed by reference (although this isn't illustrated here).

```

<script type = "text/javascript">
  function add(x, y) { return x + y; }
  function subtract(x, y) { return x - y; }
  function multiply(x, y) { return x * y; }
  function divide(x, y) { return x / y; }

  function operate(operator, operand1, operand2)
  {
    return operator(operand1, operand2);
  }

  document.writeln(operate(add,
                           operate(subtract, 10, 5),
                           operate(multiply, 4, 5)),
                   "<BR>");

  operators = new Object();
  operators["add"] = function(x, y) {return x + y; };
  operators["subtract"] =
    function(x, y) {return x - y; };
  operators["multiply"] =
    function(x, y) {return x * y; };
  operators["divide"] = function(x,y) {return x/y; };
  operators["pow"] = Math.pow;

  function operate2(op_name, operand1, operand2)
  {
    if (operators[op_name] == null)
      return "unknown operator";
    else
      return operators[op_name](operand1, operand2);
  }

  document.writeln(operate2("add", "hello",
                           operate2("add", " ", "world")),
                   "<br>",
                   operate2("pow", 10, 2) );

</script>

```

```

25
hello world
100

```

The arguments Object

Although a JavaScript function is defined with a fixed number of named arguments, it can be passed any number of arguments when it is invoked.

The `arguments[]` “array” (actually an object and not an array) gives full access to these argument values, even when some are unnamed.

Given a call `f(v0, v1, ...)`, in the definition of function `f`,

- `arguments[0]` is `v0`,
- `arguments[1]` is `v1`,
- etc.

`arguments.length` is the number of arguments in the call.

JavaScript doesn't check that a function is invoked with the correct number of arguments.

But we can use the `arguments` object to do this.

Example:

```
function f(x, y, z)
{
  if ( arguments.length != 3 ) {
    alert( "function f called with " +
          arguments.length +
          " arguments, but it expects 3." );
    return null;
  }
  ... // The normal code goes here.
}
```

We can also exploit `arguments[]` to write functions that take variable numbers of arguments.

Example:

```
<script type = "text/javascript">
  function max()
  {
    var m = Number.NEGATIVE_INFINITY;

    for ( var i = 0; i < arguments.length; i ++ )
      if ( arguments[i] > m )
        m = arguments[i];

    return m;
  }

  document.writeln( max( 1, 10, 100, 2, 3, 1000,
                        4, 5, 10000, 6 ) );
</script>
```

10000