JavaScript III

Functions: Part 1

Creating Functions: Syntax

- Functions bodies always require enclosing braces, even if it has only one statement.
- The first version, called Declaration Notation, does not require that the function be declared before it is used in your code, but the second one does.
- Don’t use declaration notation inside of a loop or if block.

```
//create a function
function sendAlert(){
  alert("Alert!");
}
//create a function and assign it to a var
var a = function(){
  alert("Alert a!");
}
//calling both kinds of functions
sendAlert();
a();
```

Functions with Arguments

```
//function with an argument 'x'
function squareIt(x){
  return x * x;
}
//using a function call as an argument
alert(squareIt(5));
```

Functions With Multiple Arguments

Some functions return values, other just produce side effects

```
var power = function(base, exponent) {
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
};

console.log(power(2, 10));
// → 1024
```

Parameters and Scope

- Parameters (arguments) to a function are given to it from the caller
- Variables declared inside of functions are always *local* to the function if the `var` keyword is used
    - If var is not used, the interpreter will search for the nearest variable with the same name and used that. If none are found, it declares it in the global scope
- Variables declared outside of a function are *global*

Scope Examples

```
var x = "outside";

var f1 = function() {
  var x = "inside f1";
};
f1();
console.log(x);
// → outside
```

```
var f2 = function() {
  x = "inside f2";
};
f2();
console.log(x);
// → inside f2
```

Nested Scope

In JavaScript, functions can be created inside of functions!

```
var landscape = function() {
  var result = "";
  var flat = function(size) {
    for (var count = 0; count < size; count++)
      result += "_";
  };
  var mountain = function(size) {
    result += "/";
    for (var count = 0; count < size; count++)
      result += "'";
    result += "\\";
  };
  flat(3);
  mountain(4);
  flat(6);
  mountain(1);
  flat(1);
  return result;
};

console.log(landscape());
// → ___/''''\______/'\_
```

- Inner scopes can access outer scopes, but not vice-versa. The inner functions cannot see each other's scopes.
  - `result` is used inside the functions, not declared.

Functions as Values

- Function variables as names for specific pieces of the program
- But they can do all the things other values can do
- You can use them in arbitrary expressions, pass them into a function, store them, assign them a new value, as well as call them
- This is all part of the functional programming features of JavaScript
- More about this later

Functions as Variables Example

- Notice that the `launchMissles` function below can be assigned a new value given certain conditions
- If you want to call the function, just put `()` after the variable name

```
var launchMissiles = function(value) {
  missileSystem.launch("now");
};
if (safeMode)
  launchMissiles = function(value) {/* do nothing */};
```

Optional Arguments

- You can put more or less arguments into a function call than have been defined for it
  - Extra arguments are just ignored.
    Arguments that aren't passed in are assigned the value undefined

```
function power(base, exponent) {
  if (exponent == undefined)
    exponent = 2;
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
}

console.log(power(4));
// → 16
console.log(power(4, 3));
// → 64
```

Closures

- What happens to `localVariable`?
- The ability to reference a specific instance of local variables in an enclosing function: *closure*

```
function wrapValue(n) {
  var localVariable = n;
  return function() { return localVariable; };
}

var wrap1 = wrapValue(1);
var wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

Closure, Another Example

References to local variables are “frozen” inside the returned function for later use by *that* function --- even though they not available to anything else.

```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

var twice = multiplier(2);
console.log(twice(5));
// → 10
```

Closures are used as *callbacks* --- a function that needs to be called when the original outer function has already completed and needs access to the latest values of outer function’s local variables --- more on this later

Functions and Side Effects

- Functions can be called for their side effects, or for their return value.
    - E.g., to print a line, or to return a number; the second is useful in more situations than the first.
- A pure function is one that only returns a value and has no side effects.
- Pure functional programming has no side-effects.