305CDE Lab 2

JavaScript Objects and Functions: Part I

October 2014

Overview

- Objects
- Functions

Objects

What Are Objects?

Example:

```
var employee = {
    firstName: "Colin",
    lastName: "Stephen",
    department: "Computing",
    hireDate: new Date()
};
```

JavaScript objects can be thought of as simple collections of *name-value pairs*. As such, they are similar to:

- Dictionaries in Python
- Hash tables in C and C++
- Associative arrays in PHP

Name-Value Pairs

The "name" part is a JavaScript string. Using "" signs is optional *if the string would be a valid JS variable name*:

- `"age"` OK, `age` OK
- `"first_name"` OK, `first_name` OK
- `"second-name"` OK, `second-name` NOT OK
- `"date of birth"` OK, `date of birth` NOT OK

The last two examples are strings that are not valid as JS variable names.

The "value" can be *any* JavaScript value:

- `112233`
- `"hello world"`
- `function() { \\ do something }`
- `false`

Object Creation

The preferred way to create objects in JS is using an "object literal":

```
var empty_object = {};

var physicist = {
  "first-name": "Albert",
  "second-name": "Einstein"
  "age": 135
};
```

Nested Objects

Remember that the value can be *any* JS value. That includes other objects. In other words: objects can be *nested*.

```
var flight = {
  airline: "BA",
  number: 882,
  departure: {
    IATA: "SYD",
    time: "2014-09-22 14:45",
    city: "Sydney"
  },
  arrival: {
    IATA: "LAX",
    time: "2014-09-23 10:32",
    city: "Los Angeles"
  }
};
```

Getting Object Values

Object values can be retrieved in two ways:

- Use `[ ]` around a string with the name to retrieve as a suffix to the object name:

```
physicist["first-name"]  // returns "Albert"
flight["number"]  // returns 882
```

- If the name is a legal JS name (and not a reserved word) then the `.` notation can also be used:

```
flight.airline  // returns 882
flight.departure.city  // returns "Sydney"
```

Undefined Values

If you try to retrieve a nonexistent name from an object, JS returns `undefined`:

```
physicist["middle-name"]  // returns undefined
flight.capacity  // returns undefined
```

TIP: the OR operator || can be used to fill in "default" values:

```
var middle = physicist["middle-name"] || "(none)"
var capacity = flight.capacity || "unknown capacity"
```

Undefined Objects

If you try to retrieve a value from an object that is `undefined`, JS throws a `TypeError` exception:

```
fakeObject["any-string"]  // throw "TypeError"

flight.capacity  // returns undefined
flight.capacity.minimum  // throw "TypeError"
```

TIP: the AND operator && can be used to guard against this problem:

```
flight.capacity  // undefined
flight.capacity.minimum  // throw "TypeError"
flight.capacity && flight.capacity.minimum  // undefined
```

Setting Object Values

Object values are set in two ways:

- During object creation, unless your object is empty {}:

```
var employee = {name: "Colin"};
employee.name  // returns "Colin"
```

- By assignment. This sets a new value if the name does not already exist. Otherwise, it updates the existing value:

```
physicist["middle-name"]  // returns undefined
physicist["middle-name"] = "Bob";
physicist["middle-name"]  // returns "Bob"

flight.arrival.city  // returns "Los Angeles"
flight.arrival.city = {full: "Los Angeles", short: "LA"}
flight.arrival.city.short  // returns "LA"
```

Call By Reference

Objects are passed around in JS programs "by reference". They are never copied.

```
var a = {};
var b = {};
a.test = "hello";
b.test  // returns undefined

var a = {};
var b = a;
a.test = "hello";
b.test  // returns "hello"

var stooge = {first: "Jerome", second: "Howard"}
var x = stooge;
x.nickname = "Curly";
var nick = stooge.nickname;
nick  // returns "Curly"
```

Functions

Reminder: Function Creation

Saw the basics last week. For example:

```
function add(x, y) {
  var total = x + y;
  return total;
};
```

- This creates a function object using a *function literal*

Function Scope

JavaScript uses *function scope* rather than block scope.

- Parameters and variables defined in a function are *not visible* outside the function.
- Parameters and variables defined anywhere within a function are *visible everywhere* within the function.

TIPS:

- Declare all the variables used in a function at the top of the function body.
- Look for (references to) function literals when trying to determine scope.

Anonymous vs Named Functions

Note that we named the function `add`. Why name functions?

- The function can use its own name to call itself recursively.
  - Useful for manipulating tree structures, such as browser DOM
- The name can be used by debuggers and other tools.

However, the function name is optional. For example:

```
var add = function(x, y) {
  var total = x + y;
  return total;
};
```

- This defines an *anonymous function* and assigns it to the variable name `add`
- `add` could be reassigned later in the program.
- Both anonymous and named functions are common in JS.

Invoking and `this`

Invoking or *calling* a function

- Call with brackets `( )` containing 0 or more expressions:
  - `add(1,2) // returns 3`
- Suspends the current JS execution
- Passes control and parameters to the function being invoked
- Also passes two additional hidden parameters:
  - `this`
  - `arguments`

`this` parameter:

- The value of `this` inside the scope of the function *depends on how the function was invoked*:
  - method
  - function
  - constructor (next week)
  - `apply` (next week)

Methods: Functions as Object Properties

- Methods are functions stored as properties of objects.
- When a method is invoked, `this` is bound to that object.

```
var myValueObject = {
  value: 0,
  increment: function(inc) {
    this.value += typeof inc === 'number' ? inc : 1;
  }
};

myValueObject.increment();
myValueObject.value  // returns 1

myValueObject.increment(2);
myValueObject.value  // returns 3
```

‘Regular’ Function Invocation

You have seen it, just use brackets after a direct reference to the function object.

```
add(3, 4);  // returns 7
```

Contrast with method invocation which looks like:

```
myObject.methodName(3, 4);
myObject["methodName"](3, 4);
```

`this` and `that`

PROBLEM: function invocation binds `this` to the global object!!

- So methods cannot invoke inner functions to help do their work.

Why?

- The `this` within the invoked function definition is not bound to the `this` of the method calling it!!

WORKAROUND: in the method, define a variable `that` and assign it the value of `this`.

this and that Example

```
// PROBLEM
myValueObject.double = function () {
  var helper = function() {
    this.value = add(this.value, this.value);
  };
  helper();  // function invocation BAD
};

// WORKAROUND
myValueObject.double = function() {
  var that = this;
  var helper = function() {
    that.value = add(that.value, that.value);
  };
  helper();  // function invocation GOOD
};
```