Department of Computing
COVENTRY UNIVERSITY

JavaScript

Objects, Functions & Closures

Coventry University

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 (quotes if not a valid variable name)

```
age
"first name"
"age"

first name
```

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",
  departure: {
    IATA: "SYD",
    time: "2014-09-22 14:45"
  },
  arrival: {
    IATA: "LAX",
    time: "2014-09-23 10:32"
  }
};
```

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 the . notation can 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"
```

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"
```

Avoiding TypeErrors

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 at Creation

Object values are set in two ways:
During object creation, unless your object is empty {}:

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

Setting Values by Assignment

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

```
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"
```

Prototypes

- Every JavaScript object has a prototype. The prototype is also an object.
- All JavaScript objects inherit their properties and methods from their prototype.
- Objects created with new Object(), inherit from a prototype called Object.prototype, Objects created with new Date() inherit the Date.prototype
- With a constructor function, you can use the **new** keyword to create new objects from the same prototype

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}

var somone = new Person("John", "Doe", 50, "blue");
```

● Sometimes you want to add new properties (or methods) to an existing object.

```
someone.name = function () {
    return this.firstName + " " + this.lastName;
};
```

● Sometimes you want to add new properties (or methods) to all existing objects of a given type.

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
    this.nationality = "English";
    this.name = function() {return this.firstName + " " + this.lastName;};
}
```

```
Person.prototype.nationality = "English";

Person.prototype.name = function() {
    return this.firstName + " " + this.lastName;
};
```

Function Creation

This creates a function object using a function literal

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

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.

Named Functions

The function can use its own name to call itself
Useful for manipulating tree structures

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

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 Functions

There are several ways to call a function:

- function (stand-alone)
- method (part of an object)
- Immediately Invoked Function Expression (IIFE)
- Constructor Function

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.

Example

```
// PROBLEM
myObject.double = function() {
  var helper = function() {
    // stuff goes here...
  };
  helper(); // invocation BAD
};
```

```
// WORKAROUND
myObject.double = function() {
  var that = this;
  var helper = function() {
    // stuff goes here...
  };
  helper(); // invocation GOOD
};
```

Closures

Inner functions get access to parameters and variables of functions they are inside (except this and arguments)

The inner function can “live longer” than its container

Can be used to maintain state or to protect “private” data:

Closure Example

```
var myObject = (function() {
    var value = 0; // private data!
    return {
        increment: function(inc) {
            value += inc || 1;
        },
        getValue: function() {
            return value;
        }
    };
})();
```

Closure Explained

The previous example creates myObject by invoking a function that returns an object literal

The function defines value

That variable is available to the increment() and getValue() methods

- Even when the outer function has completed its execution!

The value is not available to the rest of the program

IIFE Functions

The previous example uses an immediately-invoked function expression (IIFE, pronounced ‘IFFY’). Here is a simpler use:

```
var a = 1;
var b = 2;
(function() {
    var b = 3;
    a += b;
})();
a // returns 4
b // returns 2
```