JavaScript Fundamentals

Pat Morin

COMP 2405

Outline

- Data types

- Arrays

- Functions

- Objects

- Regular Expressions

Carleton
UNIVERSITY
Canada's Capital University

Data Types

- There is only one kind of variable and it is declared with the `var` keyword

- Basic values are
  - Numbers, like 42 and 3.14159
  - Booleans, `true` and `false`
  - Strings, like "thanks for all the fish..." and 'have a good day'
  - `null`
  - `undefined`, the value of a variable declared by never assigned to

```
var answer = 42;

answer = "thanks for all the fish...";
```

Type Conversion

- Expressions involving a string, the + operator and a number will convert the numbers to strings

```
x = "The answer is " + 42
  // returns "The answer is 42"

y = 42 + " is the answer"
  // returns "42 is the answer"
```

- With other operators, strings are converted to numbers

```
"37" – 7 // returns 30
"37" + 7 // returns "377"
```

Type Conversion (2)

- Boolean values
  - `null` and `undefined` treated as `false`
  - 0 treated as `false`
  - "" (the empty string) treated as false
  - Everything else treated as true

Carleton
UNIVERSITY
Canada's Capital University

Variables

- A variable can be declared using the var keyword or simply assigned to

- The scope of a variable is limited to the containing block

```
var x = 42;
var q;          // q = undefined
y = 75;
if (condition) {
  var z = 411;
}
// z no longer in scope
```

Global Variables

- Global variables are properties of the global object

- In web pages, this object is called `window`

- We can use the `window.variable` syntax to do this

- We can access a global variable in another window if we know the name of the other window

- This can be a security risk

Constants

- Read-only named constants can be created with the const keyword

- Scoping rules for constants are exactly the same as for variables

- Constants live in the same name-space as variables and function names

```
const pi = 3.14156;
```

Array Literals

- Array literals can be specified with the [] syntax
- Adding extra commas creates undefined array entries
- This actually creates an `Array` object

```
var fish = ["lion", "angel", "grouper"];

var places = ["home", , "school", "work"];
places = ["home", undefined, "school", "work"]
```

More Literals

- Strings
  - single- or double-quoted

- Boolean
  - true and false

- Integers
  - specified in decimal, octal, or hexadecimal

- Floating-point Literals
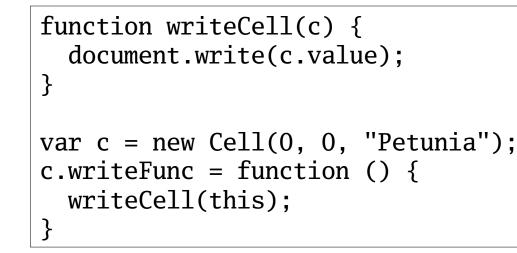 - in the usual ways

Functions

- JavaScript functions can be created using the `function` keyword

- Functions can return a value using the return keyword (or return `undefined` by default)

```
function factorial(n) {
   if ((n == 0) || (n == 1))
      return 1;
   else {
      var result = (n * factorial(n-1) );
      return result;
   }
}
```

Carleton
UNIVERSITY
Canada's Capital University

11

Functions - Weirdness

- Scope rules for functions are the same as for variables

- Functions don't have to have names

- Functions can be assigned to variables or object properties

```
function writeCell(c) {
  document.write(c.value);
}

var c = new Cell(0, 0, "Petunia");
c.writeFunc = function () {
  writeCell(this);
}
```

12

Functions – As Arguments

- A function can be an argument to a function

```
function generalSum (f, a) {
  var sum = a[0];
  for (var i = 1; i < a.length; i++) {
    sum = f(sum, a[i]);
  }
  return sum;
}

var myArray = [1, 9, 4, 2, 3];
var sum = generalSum(
    function (x,y) { return x + y; },
    myArray);
```

Function Arguments

- JavaScript is very flexible with function arguments

- A function can be called with more or less arguments than the number of declared parameters

- Too few arguments: leaves parameters undefined

```
function showThese(x, y) {
  document.write(x + "\n");
  document.write(y + "\n");
}
showThese("hello");   // prints hello and undefined
```

Functions - arguments

- All the arguments to a function can be accessed through the (implicit) arguments pseudo-array

```
function printThese () {
  for (var i = 0; i < arguments.length; i++) {
    document.writeln(arguments[i]);
  }
}
printThese("a", "b", "c", "d", 42);
```

eval

- The eval function evaluates a string as if it were JavaScript code

- The evalation environment is the same as that in which eval is called

```
var myCode = "document.writeln(x);"

var x = 56;
eval(myCode);     // prints 56
```

Object Literals

- JavaScript has objects but these are not what you're used to

- They are closer to C structs or Perl hashes than Java objects

```
var employee {
   firstName: "Patrick",
   lastName: "Morin",
   eId: 244333433
};

document.write(employee.lastName + ", "
               + employee.firstName + ", "
               + employee.eId);
```

Objects

- The values in an object are usually called *properties*

- Property names can also be numbers

```
var employee = {
   1: "Patrick",
   2: "Morin",
   eId: 244333433
};

document.write(employee[1] + ", "
             + employee[2] + ", "
             + employee.eId);
```

Accessing Object Properties

- Object properties can be accessed using the . or [] operators

```
var car = { color: "red",
            weight: 2000,
            mfr: "Hyundai",
            cost: 21000,
            1: "dohc",
            2: "abs" };

document.write(car.color); // "red"
document.write(car["color"]); // "red"
document.write(car[color]); // ERR: color undefined
var prop = "color";
document.write(car[prop]); // "red"
document.write(car.1); // "dohc"
document.write(car[1]); // "abs"
```

Operators

- We have already seen familiar operators, but these are new:
  - `delete`
  - `in`
  - `instanceof`
  - `new`
  - `this`
  - `typeof`
  - `void`

delete

- delete removes something from a name space

- Can remove an object, a property, or an element at an index

- Future accesses to that will evaluate to undefined

```
delete objectName
delete objectName.property
delete objectName[index]
```

- Can remove an implicitly declared variable, but not one declared using the `var` keyword

in

- The in operator determines whether an object has a certain property or an array has a certain index

```
if ("cost" in car) {
  document.write("Cost: " + car.cost + "\n");
}

if (23 in a) {
  document.write("twenty-third: " + a[23]);
}
```

typeof

- The `typeof` operator returns a string representing the type of the argument

- Can be one of
  - `"function"`
  - `"string"`
  - `"number"`
  - `"object"`
  - `"undefined"`

```
document.write(typeof(car));  // "object"
```

void

- The void operator specifies an expression to evaluate without returning a value

- Useful within an `href` attribute:

```
<a href="javascript:void(0)">Do nothing</a>

<a href="javascript:void(document.form.submit())">
Click here to submit
</a>
```

Objects and Classes

- JavaScript doesn't really have classes

- Instead, you define a *constructor function* that sets the properties of the implicit variable "this"

```
function Cell(i, j, val) {
  this.row = i;
  this.col = j;
  this.value = val;
}
var c = new Cell(5, 4, "Priscilla");
document.write("c.row = " + c.row + "\n");
document.write("c.col = " + c.col + "\n");
document.write("c.value = " + c.value + "\n");
```

instanceof

- The `instanceof` keyword tests if an object is of a specific class (created by a constructor with a specific name)

- This really checks if the object was created using the named constructor function

```
var c = new Cell(5, 4, "Priscilla");

if (c instanceof Cell) {
  document.write("c is a Cell");
}
```

Prototypes – Adding Properties

- Constructor functions have a property named `prototype` that allows for the creation of properties *after the fact*

```
Cell.prototype.width = 20;

var c1 = new Cell(0, 0, "treasure");
var c2 = new Cell(4, 0, "hunt");

c1.width = "10";
document.writeln("c2.width = " + c2.width);
document.writeln("c2.width = " + c2.width);
```

Object Methods

- Any function can be turned into an object method that has access to `this`

```
function pC () {
  document.writeln(this.value);
}

var c = new Cell(0, 3, "hello");
c.print = pC;

c.print();
```

Objects and Default Parameters

- Here's a common idiom for making default parameter values

```
function Cell(i, j, val) {
  this.row = i || -1;
  this.col = j || -1;
  this.value = val || "";
}
```

- This works because a || b evaluates to a unless a is false

- If a is false then a || b evaluates to b

Object Methods (Cont'd)

- But it's easier to use anonymous functions within the constructor function

```
function Cell(i, j, val) {
  this.row = i;
  this.col = j;
  this.value = val;
  this.printOn = function (doc) {
    doc.writeln("[" + this.value + "]");
  }
}

...

c.printOn(document);
```

Getters and Setters

- Recall that, for a text input `t`, setting `t.value` causes the displayed text to change?

- This is the result of a *setter* for t.value

- Getters and setters are pieces of code that are executed when you ask for the value of a variable or when you set the value of a property
    - Can have side effects (e.g., change displayed text)
    - The property may not really exist (e.g., computed from other properties)

Getter and Setter Example

```
var temp = {
  c: 0,
  get f() { return (this.c*9/5 + 32) },
  set f(x) { this.c = (x-32)*5/9 }
};

temp.c = 23;
document.writeln(temp.c + "C");
document.writeln(temp.f + "F");
temp.f = 85;
document.writeln(temp.c + "C");
document.writeln(temp.f + "F");
```

Getters and Setters in Constructors

```
function Temperature () {
  this.kelvin = 0;
  this.celsius getter = function() {
    return this.kelvin - 273;
  }
  this.celsius setter = function(x) {
    this.kelvin = x + 273;
  }
  this.fahrenheit getter = function() {
    return (this.kelvin - 273) * 9 / 5 + 32;
  }
  this.fahrenheit setter = function(x) {
    this.kelvin = (x - 32) * 5 / 9 + 273;
  }
}
```

Getters and Setters (Cont'd)

- The syntax is awkward, but getters and setters can be added to existing classes

```
var d = Date.prototype;
d.__defineGetter__(
  "year",
  function() { return this.getFullYear(); }
);

d.__defineSetter__(
  "year",
  function(y) { this.setFullYear(y); }
);
```

Using JavaScript Objects

- In Java, you have classes

- In JavaScript a class is defined by it's constructor function

```
function MyClass (idata) {
}

var c = new MyClass("here is some data");
```

Using JavaScript Objects (Cont'd)

- In Java you have instance methods

- In JavaScript you have functions defined within a class

```
function MyClass (idata) {
  this.toString = function() {
    return "a MyClass";
  }
}

var c = new MyClass("here is some data");
document.write(c);
```

Using JavaScript Objects (Cont'd)

- In Java you have instance variables

- In JavaScript you have properties

```
function MyClass (idata) {
  this.data = idata;

  this.toString = function() {
    return "MyClass(" + this.data + ")";
  }
}


var c = new MyClass("here is some data");
document.writeln(c);
```

Using JavaScript Objects (Cont'd)

- In Java you can declare instance methods and variables to be private

- In JavaScript you can use variables inside of constructors

```
function MyClass (idata) {
  var data = idata;

  this.toString = function() {
    return "MyClass(" + data + ")";
  }
}
var c = new MyClass("here is some data");
document.writeln(c);
document.writeln(c.data);    // undefined
```

Private Data (Cont'd)

- This works because Java does static lexical scoping

- When any function is called (or block of code executed) a new *stack frame* is created to hold the local variables

- These local variables are only accessable by blocks of code defined within the scope of those local variables

Inheritance

- In Java we have inheritance

- A subclass inherits the instance variables and methods of its superclass

- In JavaScript we use the `prototype` property of functions

- Recall that setting `Class.prototype.xxxx` specifies a property (`xxxx`) that all objects created by the Class constructor function have

Subclassing – First Way

```
function Employee (name, dept) {
  this.name = name || "";
  this.dept = dept || "general";
}


function Manager () {
  this.reports = [];
}
Manager.prototype = new Employee();


function WorkerBee () {
    this.projects = [];
}
WorkerBee.prototype = new Employee();
```

How this Works

- Remember the prototype property of a constructor functions f contains properties (and initial values) that all objects constructed by f have

- This *is not* the same as in Java

- The constructor for the parent class is only called once, when we set the prototype

- Creating a new instance of the subclass does not call the parent class constructor function again

- If we want to do that we should explicitly call the parent constructor

Subclassing – Another Way

- We can also do subclassing by simply calling the parent class' constructor

```
function Employee () {
  this.name = "";
  this.dept = "general";
}

function Manager () {
  this.reports = [];
  this.base = Employee;
  this.base();
}
```

- How do these interact with `instanceof`?

JavaScript Regular Expressions

- JavaScript supports the creation of regular expressions using the // operator or the RegExp class

```
var re1 = /ab+c/;    // match a, one or more b's the

var re2 = RegExp("ab+c");
```

- The first form is evaluated at parse (compile) time
- The second form is evaluated each time it is executed

Using Regular Expressions

- Once we have a RE we can use these operations
  - exec/match – execute a search and return an array of information
  - test/search – test for a match of the RE in a string and return a boolean value of an index, respectively
  - replace – replace a match with something else
  - split – split a string into an array of substrings

- See documentation for more details
  - http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide:Regular_Expressions

Summary

- JavaScript is similar in syntax to C/C++ and Java but
  - Variables have no type
  - Functions are more "first-class"
  - Objects are more like hashes
  - Classes are defined by creating a constructor function
  - Getters and setters offer some nice syntactic sugar
  - Private variables, subclassing, multiple inheritance, are all possible
  - Language support for regular expressions

Regular expressions