JavaScript Programming

August 2006

Sehwan Noh
sehnoh@gmail.com

Contents

- What Is JavaScript?
- JavaScript Basics (ECMAScript)
- Object Oriented Programming in JavaScript
- Regular Expressions
- Error Handling
- References

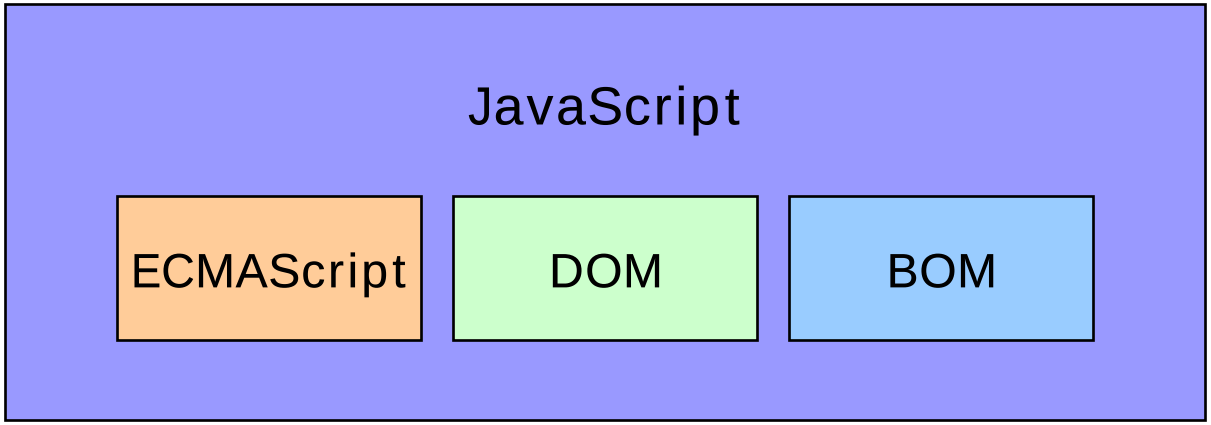
What Is JavaScript?

A Brief History of JavaScript

- Developed by Brendan Eich of Netscape, under the name of *Mocha*, then *LiveScript*, and finally *JavaScript*.
- 1995 - JavaScript 1.0 in Netscape Navigator 2.0 (Dec)
- 1996 - JavaScript 1.1 in Netscape Navigator 3.0 (Aug), JScript 1.0 in Internet Explorer 3.0 (Aug). *JavaScript had no standards governing its syntax or features*.
- 1997 - ECMAScript 1.0 (ECMA-262, based on JavaScript 1.1) (Jun), JavaScript 1.2 in Netscape Navigator 4.0 (Jun), JScript 3.0 in Internet Explorer 4.0 (Sep)
- 1998 - JavaScript 1.3 in Netscape 4.5 (ECMAScript 1.0) (Oct)
- 1999 - JScript 5.0 in Internet Explorer 5.0 (ECMAScript 1.0) (Mar), ECMAScript 3.0 (Regular expressions, error handling, etc.) (Dec)
- 2000 - JScript 5.5 in Internet Explorer 5.5 (ECMAScript 3.0) (Jul), JavaScript 1.5 in Netscape 6.0 (ECMAScript 3.0) (Nov)
- 2001 - JScript 5.6 in Internet Explorer 6.0 (Aug)
- 2005 - JavaScript 1.6 in Firefox 1.5 (Nov)

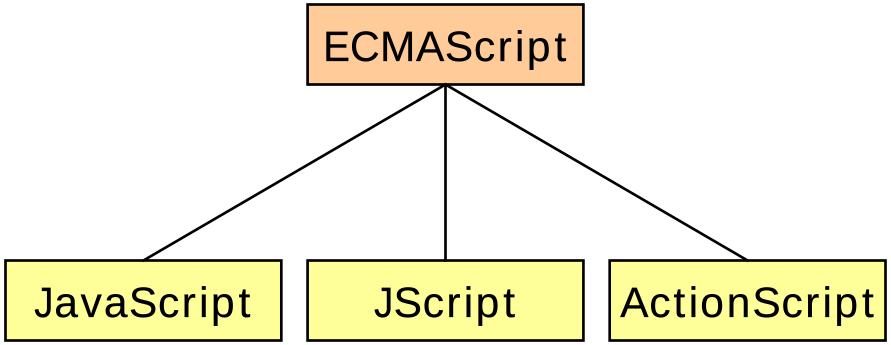
JavaScript Implementations

- A complete JavaScript implementations is made up of three distinct parts:
  - The Core (ECMAScript)
  - The Document Object Model (DOM)
  - The Browser Object Model (BOM)



ECMAScript

- ECMAScript is simply a description, defining all the properties, methods, and objects of a scripting language.
  - Syntax
  - Types
  - Statements
  - Keywords
  - Reserved Words
  - Operators
  - Objects



- Each browser has it own implementation of the ECMAScript interface, which is then extended to contain the DOM and BOM.
- Today, all popular Web browsers comply with the 3rd edition of ECMA-262.

Document Object Model (DOM)

- The Document Object Model (DOM) describes methods and interfaces for working with the content of a Web page.
- The DOM is an tree-based, language-independent API for HTML as well as XML. (cf. The SAX provides an event-based API to parse XML.)
- The W3C DOM specifications: Level1, Level2, Level3
- The *document* object is the only object that belongs to both the DOM and the BOM.
  - getElementsByTagName(), getElementsByName(), getElementById()
- All attributes are included in HTML DOM elements as properties.
  - oImg.src = "mypicture.jpg";
  - oDiv.className = "footer"; // cf. class → className

Browser Object Model (BOM)

- The Browser Object Model (BOM) describes methods and interfaces for interacting with the browser.
- Because no standards exist for the BOM, each browser has it own implementations.
- The *window* object represents an entire browser window.
  - objects
    - document - anchors, forms, images, links, location
    - frames, history, location, navigator, screen
  - methods
    - moveBy(), moveTo(), resizeBy(), resizeTo(),
    - open(), close(), alert(), confirm(), input()
    - setTimeOut(), clearTimeOut(), setInterval(), clearInterval()
  - properties
    - screenX, screenY, status, defaultStatus, etc.

JavaScript is...

- JavaScript is one of the world’s most popular programming languages.
- JavaScript is not interpreted Java.
- JavaScript has more in common with functional language like Lisp or Scheme that with C or Java.
- JavaScript is well suited to a large class of non-Web-related applications.
- Design errors? No programming language is perfect.
- Lousy implementations were embedded in horribly buggy web browsers.
- Nearly all of the books about JavaScript are quite awful.
- Many of people writing in JavaScript are not programmers.
- JavaScript is now a complete object-oriented language.
- JavaScript does not have class-oriented inheritance, but it does have prototype-oriented inheritance.

JavaScript Basics (ECMAScript)

Syntax

- JavaScript borrows most of its syntax from Java,
  but also inherits from Awk and Perl, with some inherits
  influence from Self in its object prototype system.
- The basic concepts of JavaScript:
  - Everything is case-sensitive.
  - Variables are loosely typed.
    - Use the *var* keyword.
    - Variables don’t have to be declared before being used.
  - End-of-line semicolons are optional.
    - var test1 = “red”
      var test2 = “blue”; //do this to avoid confusion.
  - Comments are the same as in Java, C, and Perl.
  - Braces indicate code blocks.

Keywords & Reserved Words

- The keywords and reserved words cannot be used as variables or function names.
- Keywords

  break, case, catch, continue, default, delete, do, else, finally, for, function, if, in, instanceof, new, return, switch, this, throw, try, typeof, var, void, while, with
- Reserved Words

  abstract, boolean, byte, char, class, const, debugger, double, enum, export, extends, final, float, goto, implements, import, int, interface, long, native, package, private, protected, public, short, static, super, synchronized, throws, transient, volatile

Primitive and Reference Values

- Primitive Values
  - Primitive values are simple pieces of data that are stored on the *stack*,
  - which is to say that their value is stored directly in the location that the variable accesses.
  - The value is one of the JavaScript primitive types:
    - *Undefined, Null, Boolean, Number,* or *String*.
  - Many languages consider strings as a reference type and not a primitive type, but JavaScript breaks from this tradition.
- Reference Values
  - Reference values are objects that are stored in the *heap*,
  - meaning that the value stored in the variable location is a pointer to a location in memory where the object is stored.

Primitive Types

- JavaScript has five primitive types:
  - Undefined
    - The Undefined type has only one value, *undefined*.
  - Null
    - The Null type has only one value, *null*.
  - Boolean
    - The Boolean type has two values, *true* and *false*.
  - Number
    - 32-bit integer and 64-bit floating-point values.
    - Infinity → isFinite()
    - NaN (Not a Number) → isNaN()
  - String
    - Using either double quotes(”) or single quote(’).
    - JavaScript has no character type.

The typeof Operator

- To determine if a value is in the range of values for a particular type, JavaScript provides the *typeof* operator.
- Why the typeof operator returns “object” for a value that is null.
  - An error in the original JavaScript implementation.
  - Today, it is rationalized that null is considered a placeholder for an object.

Value	typeof
Boolean	boolean
Number	number
String	string
Undefined	undefined
Null	object
Object	object
Array	object
Function	function

Conversions

- Converting to a String
  - Primitive values for booleans, numbers, strings are *pseudo-objects*, which means that they actually have properties and methods.

  -
    ```
    alert(“blue”.length); //outputs “4”
    alert((false).toString()); //outputs “false”
    alert((10).toString(2)); //outputs “1010”
    alert((10).toString(16)); //outputs “A”
    ```
- Converting to a Number
  - JavaScript provide two methods for converting non-number primitives into numbers: parseInt() and parseFloat().

  -
    ```
    var num1 = parseInt(“0xA”); //returns 10
    var num2 = parseFloat(“4.5.6”); //returns 4.5
    var num3 = parseInt(“blue”); // returns NaN
    ```

Type Casting

- Boolean(value)
  - Boolean(“”) → false; Boolean(“hi”) → true
  - Boolean(0) → flase; Boolean(100) → true
  - Boolean(null) → false; Boolean(undefined) → false
  - Boolean(new Object()) → true
- Number(value)
  - Number(false) → 0; Number(true) → 1
  - Number(null) → 0; Number(undefined) → NaN
  - Number(“4.5.6”) → NaN (cf. parseFloat())
  - Number(new Object()) → NaN
- String(value)
  - String(null) → “null”
  - String(undefined) → “undefined”

Reference Types

- Reference types are commonly referred to as *classes*, which is to say that when you have a reference value, you are dealing with an object.
- ECMAScript defines “object definitions” that are logically equivalent to “classes” in other programming languages.
- The *new* operator
  - □ var obj = new Object;
    var obj = new Object(); //do this to avoid confusion.
- The *instanceof* operator
  - □ The instanceof operator identifies the type of object you are working with.
  - □ var aStringObject = new String(“Hello”);
    var result = (aStringObject instanceof String); //returns true.

The Object Class

- The Object class in JavaScript is similar to java.lang.Object in Java.
- Each of properties and methods are designed to be overridden by other classes.
- Properties of the Object class:
  - constructor – A reference value (pointer) to the function that created the object.
  - prototype – A reference value to the object prototype for this object.
- Methods of the Object class:
  - hasOwnProperty(property)
  - isPrototypeOf(object)
  - propertyIsEnumerable(property)
  - toString()
  - valueOf()

Primitive Type-related Classes

- The Boolean Class
  - It’s best to use Boolean primitives instead of Boolean objects.
  - var result = (new Boolean(false)) && true; //returns true;
    - cf. All objects are automatically converted to true in Boolean expressions.
- The Number Class
  - Methods:
    - toFixed(), toExponential(), toPrecision(), etc.
  - Whenever possible, you should use numeric primitives instead.
- The String Class
  - Property: length
  - Methods:
    - charAt(), charCodeAt(), indexOf(), lastIndexOf()
    - localeCompare(), concat(), slice(), substring()
    - replace(), split(), match(), search()
    - toLowerCase() toLocaleLowerCase(), toUpperCase(), toLocaleUpperCase(), etc.

Operators

- Unary
  - delete, void, Prefix ++/--, Postfix ++/--, Unary +/-
- Bitwise
  - ~, &, |, ^, <<, >>, >>>
- Boolean
  - !, &&, ||
- Arithmetic
  - +, -, *, /, %
- Assignment
  - =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=
- Comparison
  - ==, !=, >, >=, <, <=, ===, !==
- Conditional
  - variable = boolean_expression ? true_value : false_value;
- Comma
  - var iNum = 1, iNum=2;

Statements

- if…else
- switch
- while
- do…while
- for
- for…in
  - It is used to enumerate the properties of an object. (cf. All object has a method propertyIsEnumerable().)
  - for (*property* in *expression*) *statement*
- with
  - A very slow segment. It is best to avoid using it.
- Label, break and continue
- try…catch…finally
- throw

Functions

- A function is a collection of statements that can be run anywhere at anytime.
- The *function* keyword
  - □ function functionName(arg0, arg1,..., argN) { functionBody }
  - □ var functionName = function(arg1, arg2,..., argN) { functionBody }
- Functions that have no return value actually return *undefined.*
- Functions cannot be overloaded.
  - □ The last function becomes the one that is used.
- The Function Class
  - □ Functions are actually full-fledged objects.
  - □ var functionName = new Function(arg1, arg2,..., argN, functionBody);

Functions

- The arguments object
  - Within a function’s code, a special object called *arguments* give the developer access to the function’s parameters without specifically naming them.
  - Any developer-defined function accepts any number of arguments (up to 255).
  - Any missing arguments are passed in as *undefined*.
  - Any excess arguments are ignored.

    -
      ```
      function doAdd() {
          if (arguments.length == 1) {
              alert(arguments[0] + 10);
          } else if (arguments.length == 2) {
              alert(arguments[0] + arguments[1]);
          }
      }
      doAdd(10);      //outputs “20”
      doAdd(30, 20); //outputs “50”
      ```

Functions

- Closures
  - A *closure* is an expression (typically a function) that can have free variables together with an environment that binds those variables (that *closes* the expression).
  - Functions can be defined inside of other functions. The inner function has access to the variables and parameters of the outer function.
  - The inner function is a closure.

    -
      ```
      var iBaseNum = 10;
      function addNumbers(iNum1, iNum2) {
         function doAddition() {
            return iBaseNum + iNum1 + iNum2;
         }
         return doAddition():
      }
      ```

Object Oriented Programming in JavaScript

Object Oriented Terminology

- A *class* is a kind of recipe for an object.
- An *object* is a particular instance of a class.
- ECMAScript has no formal classes; ECMA-262 describes *object definitions* as the recipes for an object.
- ECMA-262 defines an *object* as an "unordered collection of properties each of which contains a primitive value, object, or function".
- If a member of an object is a function, it is considered to be a *method*; otherwise, the member is considered a *property*.
- ECMAScript supports the requirements of object-oriented languages.
  - Encapsulation
  - Inheritance
  - Polymorphism

Class-based vs. Prototype-based

- Class-based Programming
  - A style of object-oriented programming in which inheritance is achieved by defining classes of objects, as apposed to the objects themselves.
  - The most popular and developed model of OOP.
  - Smalltalk, Java, C++, etc.
- Prototype-based Programming
  - A style of object-oriented programming in which classes are not present, and behavior reuse (aka. inheritance) is accomplished through a process of cloning existing objects which serve as prototypes.
  - Class-less, prototype-oriented, or instance-based programming.
  - Self, Cecil, ECMAScript(JavaScript), etc.

Early Binding vs. Late Binding

- *Early binding* means that properties and methods are defined for an object (via its class) before it is instantiated so the compiler/interpreter can properly assemble the machine code ahead of time.
  - Java, Visual Basic, etc. (cf. IntelliSense)
- *Late binding* means that the compiler/interpreter doesn't know what type of object is being held in a particular variable until runtime.
  - ECMAScript(JavaScript), etc.
- Due to the late binding, JavaScript allows a large amount of object manipulation to occur without penalty.

Types of Objects in JavaScript

- Native Objects
  - □ Any object supplied by an ECMAScript implementation independent of the host environment.
  - □ *Object, Boolean, Number, String, Function, Array, Date, RegExp, Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError, URInternet Explorerrror*
- Built-in Objects
  - □ Any object supplied by an ECMAScript implementation, independent of the host environment, which is present at the start of the execution of an ECMAScript program.
  - □ Every build-in object is a native object.
  - □ *Global, Math*
- Host Objects
  - □ Any object that is not native, provided by the host environment of an ECMAScript implementation.
  - □ All BOM and DOM objects are considered to be host objects.

The Array Class

- How to create an Array object:
  - var aValues = new Array(10);
  - var aColors = new Array(“red”, “green”, “blue”);
  - var aColors = [“red”, “green”, “blue”];
- The array dynamically grows in size with each additional item.
  - aColor[3] = “yellow”; aColor[4] = “white”; …
- Property of the Array class:
  - length
- Methods of the Array class:
  - join(), split()
  - concat(), slice()
  - push(), pop()
  - shift(), unshift()
  - reverse(), sort()

The Date Class

- Based on earlier versions of java.util.Date from Java.
- How to create a new Date class:
  - var today = new Date();
- Methods of the Date class:
  - parse(), UTC()
  - Overides toString() and valueOf() differently.
  - toDateString(), toTimeString(), toLocaleString(), toLocaleDateString(), toLocaleTimeString(), toUTCString()
  - getTimezoneOffset()
  - getTime(), getFullYear(), getUTCFullYear(), getMonth(), getUTCMonth(), getDate(), getUTCDate(), getDay(), getUTCDay(), getHours(), getUTCHours(), getMinutes(), getUTCMinutes(), getSeconds(), getUTCSecounds(), getMilliseconds(), getUTCMillisenconds()
  - Also has the equivalent set methods to above get methods.

The Global Object

- The Global object is the keeper of all the functions and variables which were not defined inside of other objects.
- The Global object does not have an explicit name.
  - var pointer = Global; // error
- Sometimes the *this* variable points at it, but often not.
- In the web browsers, *window* and *self* are members of the Global object which point to the Global object.
- Properties of the Global object:
  - undefined, NaN, Infinity, and all native object constructors.
- Methods of the Global object:
  - isNaN(), isFinite(), parseInt(), parseFloat()
  - encodeURI(), encodeURIComponent(), decodeURI(), decodeURIComponent() → Unicode encoding support
    - cf. escape(), unescape() → ASCII encoding only; BOM
  - eval()

The Math Object

- Properties of the Math object:
  - E, LN10, LN2, LOG2E, LOG10E, PI, SQRT1_2, SQRT2
- Methods of the Math object:
  - min(), max(), abs()
  - ceil(), floor(), round()
  - exp(), log(), pow(), sqrt()
  - acos(), asin(), atan(), atan2(), cos(), sin(), tan()
  - random()
- A practical example:

  -
    ```
    function selectFrom(iFirstValue, iLastValue) {
        var iChoices = iLastValue - iFirstValue + 1;
        return Math.floor(Math.random() * iChoices
            + iFirstValue);
    }
    //select from between 2 and 10
    var iNum = selectFrom(2, 10);
    ```

Scope

- All properties and methods of all objects in JavaScript are *public*.
- Due to the lack of *private* scope, a convention was developed to indicate properties and methods should be considered private.
  - someObject.__color__ = “red”; or someOjbect._color = “red”;
- Strictly speaking, JavaScript doesn’t have a *static* scope.
- The *this* keyword always points to the object that is calling a particular method.
- When used inside of a function, *var* defines variables with *function-scope*. The variables are not accessible from outside of the function.
- Any variables used in a function which are not explicitly defined as *var* are assumed to belong to an outer scope, possibly to the Global Object.

Objects in JavaScript

- In JavaScript, objects are implemented as a collection of named properties.
- The most basic objects in JavaScript act as hashtables or dictionaries.
- Objects can be created directly through object literal notation:

  -
    ```
    var myDog = {
        age: 3,
        color: “black”,
        bark: function() { alert(“Woof!”); }
    }
    ```
  - The object’s properties and methods are defined as a set of comma-separated name/value pairs inside curly braces.
  - Each of the members is introduced by name, followed by a colon and then the definition.
  - The methods are created by assigning an anonymous function.

Objects in JavaScript

- Being an interpreted language, JavaScript allows for the creation of any number of properties in an object at any time.
  - myDog.name = “Snuppy”; //using *dot notation*
    myDog[“breed”] = “Afghan Hound”; //using *subscript notation*
  - var name = myDog[“name”]; //returns “Snuppy”
    var breed = myDog.breed; //returns “Afghan Hound”
  - The reserved words cannot be used in the dot notation, but they can be used in the subscript notation.
- JavaScript Object Notation (JSON)
  - JSON is a simple notation that uses JavaScript-like syntax for data exchange.
  - JSON is used pretty much everywhere in JavaScript these days, as arguments to functions, as return values, as server responses (in strings), etc.

Objects in JavaScript

- Objects can also be created by using the *new* operator and providing the name of the class to instantiate.
  - var myDog = new Object();

- A simple object:
  - var obj = new Object();
    obj.x = 1;
    obj.y = 2;

  - In addition to the x and y properties, the object has an additional property called *constructor*.

  - The object also contains a hidden link property which points to the *prototype* member of the object's constructor.

obj	
x	1
y	2
Object.prototype	
constructor	Object

Constructor

- In JavaScript, a new class is defined by creating a simple function.
- When a function is called with *new* operator, the function servers as the *constructor* for the class.
- Internally, JavaScript creates an Object, and then calls the constructor function. Inside the constructor, the variables *this* is initialized to point the just created Object.

  -
    ```
    function Foo() {
        this.x = 1;
        this.y = 2;
    }

    var obj = new Foo();
    ```
- The constructor will return the new object, unless explicitly overridden with the return statement.

obj	
x	1
y	2
Foo.prototype	
constructor	Foo
Object.prototype	
(Constructor)	Object

Prototype

- The constructed object will contain a hidden link property, which contains a reference to the constructor's *prototype* member.
- The *prototype* object is a kind of template upon which an object is based when instantiated.
- Any properties or methods on the prototype object will be passed on all instances of that class.
- Prototype Chaining
  - When evaluating an expressions to retrieve a property, JavaScript first looks to see if the property is defined directly in the object.
  - If it is not, it then looks at the object's prototype to see if the property is defined there.
  - This continues up the *prototype chain* until reaching the root prototype.
  - If the prototype chain is exhausted, the *undefined* is returned.

Defining Classes and Objects

- Factory Paradigm

```
function createCar(sColor, iDoors) {
    var oTempCar = new Object;
    oTempCar.color = sColor;
    oTempCar.doors = iDoors;
    oTempCar.showColor = function () {  alert(this.color); };
    return oTempCar;
}

var oCar1 = createCar("red", 4);
var oCar2 = createCar("blue", 3);
oCar1.showColor();    //outputs "red"
oCar2.showColor();    //outputs "blue"
```

  - □ No new operator → semantically out of favor.
  - □ Every object has its own version of showColor()
    → Each object should share the same function.

Defining Classes and Objects

- Constructor Paradigm

```
function Car(sColor, iDoors) {
    this.color = sColor;
    this.doors = iDoors;
    this.showColor = function () { alert(this.color); };
}

var oCar1 = new Car("red", 4);
var oCar2 = new Car("blue", 3);
oCar1.showColor();    //outputs "red"
oCar2.showColor();    //outputs "blue"
```

  - The new operator.
  - Just like factory paradigm, constructors duplicate functions.
  - Constructors can be rewritten with external functions, but semantically it doesn’t make sense.

Defining Classes and Objects

- Prototype Paradigm

```
function Car() {}

Car.prototype.color = "red";
Car.prototype.doors = 4;
Car.prototype.drivers = new Array("Mike", "Sue");
Car.prototype.showColor = function () { alert(this.color); };

var oCar1 = new Car();
var oCar2 = new Car();
oCar1.drivers.push("Matt");
alert(oCar1.drivers);    //outputs "Mike,Sue,Matt"
alert(oCar2.drivers);    //outputs "Mike,Sue,Matt"
```

  - The constructor has no arguments.
  - Functions can be shared without any consequences, but objects rarely meant to shared across all instances.

Defining Classes and Objects

- Hybrid Constructor/Prototype Paradigm

```
function Car(sColor, iDoors, iMpg) {
    this.color = sColor;
    this.doors = iDoors;
    this.drivers = new Array("Mike", "Sue");
}

Car.prototype.showColor = function () { alert(this.color); };

var oCar1 = new Car("red", 4);
var oCar2 = new Car("blue", 3);
oCar1.drivers.push("Matt");
alert(oCar1.drivers);    //outputs "Mike,Sue,Matt"
alert(oCar2.drivers);    //outputs "Mike,Sue"
```

  - Use the constructor paradigm to define all nonfunction properties of the object, and use the prototype paradigm to define the function properties (methods) of the object.

Defining Classes and Objects

- Dynamic Prototype Method

```
function Car(sColor, iDoors) {
    this.color = sColor;
    this.doors = iDoors;
    this.drivers = new Array("Mike", "Sue");

    if (typeof Car._initialized == "undefined") {
        Car.prototype.showColor = function () { alert(this.color); };
        Car._initialized = true;
    }
}
```

  - Use a flag(_initialized) to determine if the prototype has bean assigned any methods yet.

A Practical Example – StringBuffer

- Defining the StringBuffer class:

  -
    ```
    function StringBuffer() {
        this.__string__ = new Array();
    }
    ```

  -
    ```
    StringBuffer.prototype.append = function (str) {
        this.__strings__.push(str);
    };
    ```

  -
    ```
    StringBuffer.prototype.toString = function () {
        return this.__strings__.join("");
    };
    ```
- Testing the code:

  -
    ```
    var buffer = new StringBuffer();
    buffer.append(“hello ”);
    buffer.append(“world”);
    var result = buffer.toString(); //outputs “hello world”
    ```

Modifying Objects

- Creating a New Method
  - □
    ```
    Number.prototype.toHexString = function() {
        return.this.toString(16);
    }
    ```
  - □
    ```
    var iNum = 15;
    alert(iNum.toHexString()); // outputs “F”
    ```

- Redefining an Existing Method
  - □ The Function’s toString() method normally outputs the source code of the function.
  - □
    ```
    Function.prototype.toString = function() {
        return “code hidden”;
    }
    ```
  - □
    ```
    function sayHi() { alert(“Hi”); }
    alert(sayHi.toString()); //outputs “code hidden”
    ```

Implementing Inheritance

- Using Object Masquerading

```
function ClassA(sColor) {
    this.color = sColor;
    this.sayColor = function () { alert(this.color); };
}

function ClassB(sColor, sName) {
    this.newMethod = ClassA;
    this.newMethod(sColor);
    delete this.newMethod;
    this.name = sName;
    this.sayName = function () { alert(this.name); };
}

var objA = new ClassA("red");
var objB = new ClassB("blue", "Nicholas");
objA.sayColor();  // outputs “red”
objB.sayColor();  // outputs “blue”
objB.sayName(); // outputs “Nicholas”
```

- □ Object masquerading not intended for use in the original ECMAScript.
- □ Object masquerading supports *multiple inheritance*.

Implementing Inheritance

- Using Object Masquerading – The call() Method

```
function ClassA(sColor) {
    this.color = sColor;
    this.sayColor = function () { alert(this.color); };
}

function ClassB(sColor, sName) {
    ClassA.call(this, sColor);
    this.name = sName;
    this.sayName = function () {  alert(this.name); };
}
```

  - The 3rd edition of ECMAScript includes two new methods of the Function object: *call()* and *apply()*.
  - The first argument is the object to be used for this, and all other arguments are passed directly to the function itself.

Implementing Inheritance

- Using Object Masquerading – The apply() Method

```
function ClassA(sColor) {
    this.color = sColor;
    this.sayColor = function () { alert(this.color); };
}

function ClassB(sColor, sName) {
    ClassA.apply(this, new Array(sColor));
    // or ClassA.apply(this, arguments);
    this.name = sName;
    this.sayName = function () {  alert(this.name); };
}
```

  - □ The apply() method takes two arguments: the object to be used for this and an array of arguments to be passed to the function.
  - □ You may use the *arguments* object.

Implementing Inheritance

- Using Prototype Chaining

```
function ClassA() {}
ClassA.prototype.color = "red";
ClassA.prototype.sayColor = function () { alert(this.color); };

function ClassB() {}
ClassB.prototype = new ClassA();
ClassB.prototype.name = "";
ClassB.prototype.sayName = function () { alert(this.name); };

var objA = new ClassA();
var objB = new ClassB();
objA.color = "red";
objB.color = "blue";
objB.name = "Nicholas";
objA.sayColor();  //outputs “red”
objB.sayColor();  //outputs “blue”
objB.sayName(); //outputs “Nicholas”
```

Implementing Inheritance

- Using Prototype Chaining (continued)
  - Prototype chaining is the form of inheritance actually intended for use in ECMAScript.
  - Any properties or methods on the prototype object will be passed on all instances of that class.
  - No parameters are passed into the constructor call in prototype chaining.
  - The *instanceof* operator works in a rather unique way in prototype chaining.
  - Prototype chaining has no support for multiple inheritance.
  - Because of the unique nature of the prototype object, inheritance doesn’t work with dynamic prototyping.

Implementing Inheritance

- Hybrid Method: Object Masquerading/Prototype Chaining

```
function ClassA(sColor) {
    this.color = sColor;
}

ClassA.prototype.sayColor = function () { alert(this.color); };

function ClassB(sColor, sName) {
    ClassA.call(this, sColor);
    this.name = sName;
}

ClassB.prototype = new ClassA();
ClassB.prototype.sayName = function () { alert(this.name); };

var objA = new ClassA("red");
var objB = new ClassB("blue", "Nicholas");
objA.sayColor();  //outputs”red”
objB.sayColor();  //outputs “blue”
objB.sayName(); //outputs “Nicholas”
```

A Practical Example - Polygon

- Creating the base class: Polygon()

  -
    ```
    function Polygon(iSides) {
        this.sides  = iSides;
    }
    ```

  -
    ```
    Polygon.prototype.getArea = function () {
        return 0;
    };
    ```
- Creating the subclass: Triangle()

  -
    ```
    function Triangle(iBase, iHeight) {
        Polygon.call(this, 3);
        this.base = iBase;
        this.height = iHeight;
    }
    ```

  -
    ```
    Triangle.prototype.getArea = function () {
        return 0.5 * this.base * this.height;
    };
    ```

A Practical Example - Polygon

- Creating the subclass: Rectangle()

  -
    ```
    function Rectangle(iLength, iWidth) {
        Polygon.call(this, 4);
        this.length = iLength;
        this.width = iWidth;
    }
    ```

  -
    ```
    Rectangle.prototype.getArea = function () {
        return this.length * this.width;
    };
    ```

- Testing the code:

  -
    ```
    var triangle = new Triangle(12, 4);
    var rectangle = new Rectangle(22, 10);
    alert(triangle.sides);       //outputs "3"
    alert(triangle.getArea()); //outputs "24"
    alert(rectangle.sides);     //outputs "4"
    alert(rectangle.getArea()); //outputs "220"
    ```

Regular Expressions

Regular Expression Support

- Regular expressions are strings with a special syntax indicating the occurrence of specific characters or substrings within another string.
- Regular expressions was introduced into the 3rd edition of ECMAScript.
- JavaScript supports regular expressions through the ECMAScript RegExp class.
  - □ var reCat = new RegExp(“Cat”);
    var reCat = new RegExp(“cat”, “gi”); //g=global, i=case-insensitive (cf. m=multiline).
- Some regular expression literals use Perl-style syntax:
  - □ */string_pattern/*[*processing_instruction_flags*]
  - □ var reCat = /cat/gi;

Using a RegExp Object

- Using the methods of the RegExp object
  - test() – Determine if a string matches the specified pattern.
    - alert(reCat.test(“The cat meows.”)); //outputs “true”
  - exec() – Returns an Array. The first item in array is the first match; the others are back references.
    - var result = reCat.exec(“A Cat catch cAt Bat”); //returns an array containing “Cat”.
- Using the methods of the String object.
  - match() – Returns an array of all matches of the string.
    - var result = “A Cat catch cAt Bat”.match(reCat); //returns an array containing “Cat”, “cat” and “cAt”.
  - search() – Acts the same way as indexOf(), but uses a RegExp object instead of a substring.
    - alert(“A Cat catch cAt Bat”.search(reCat)); //outputs “2”

Characters in Regular Expressions

- Metacharacters
  - A *metacharacter* is a character that is part of regular expression syntax.
  - . ^ $ * + ? { [ ] \ | ( )
  - Metacharacters are not used as literals, and don't match themselves in regular expressions.
  - Any time you want to use one of these characters inside of a regular expression, they must be escaped.
    - var reQMark = /\?/; or
      var reQMark = new RegExp("\\?"); *// double escaping.*
- Using special characters
  - To represent a character using ASCII
    - Two-digit hexadecimal code: \x62 → "b" (cf. octal: \142)
  - To represent a character using Unicode
    - Four-digit hexadecimal code: \u0062 → "b"
  - Predefined special characters
    - \t, \n, \r, \f, \a, \e, \cX, \b, \v, \0

Character Classes

- Character classes are groups of characters to test for, which are enclosed inside of square brakets([]).
- Simple classes
  - □ A *simple class* specifies the exact characters to look for.
  - □ var result = “bat cat eat fat”.match(/**[**bcf**]**at/gi); //returns an array containing “bat”, “cat” and “fat”.
- Negation classes
  - □ A *negation class* matches all characters except for a selected few.
  - □ Use the caret (^).
  - □ var result = “bat cat eat fat”.match(/[**^**bc]at/gi); //returns an array containing “eat” and “fat”.
- Range classes
  - □ A *range class* specifies a range of characters.
  - □ Use the dash (-), which should be read as *through*.
  - □ Var result = “no1, no2 no3 no4”.match(/no[1**-**3]/gi); //returns an array containing “no1”, “no2” and “no3”.

Character Classes

- Combination classes
  - □ A *combination class* is a character class that is made up of several other character classes.
  - □ /[a-m1-4]/ → "a1", "b3", "h2", "m4", etc.
  - □ JavaScript doesn't support *union* and *intersection* classes, such as [a-m[p-z]] or [a-m[^b-e]].
- Predefined classes
  - □ . = [^\n\r] → Any character except '\n' and '\r'.
  - □ \d = [0-9] → A digit
  - □ \D = [^0-9] → A non-digit
  - □ \s = [\t\n\x0B\f\r] → A white-space character
  - □ \S = [^ \t\n\x0B\f\r] → A non-white-space character
  - □ \w = [a-zA-Z_0-9] → A word character
  - □ \W = [^a-zA-Z_0-9] → A non-word character

Quantifiers

- Quantifiers enable you to specify how many times a particular pattern should occur.
- Simple quantifiers
  - ? → Either zero or one occurrence
  - * → Zero or more occurrences
  - + → One or more occurrences
  - {n} → Exactly n occurrences
  - {n,m} → At least n but no more than m occurrences
  - {n,} → At least n occurrences

    - /ba?d/ → “bd”, “bad”
    - /ba*d/ → “bd”, “bad”, “baad”, “baaad”,..
    - /ba+d/ → “bad”, “baad”, “baaad”,..
    - /b?rea?d/ = /b{0,1}rea{0,1}d/ → “bread”, “read”, “red”,..
    - /b[ea]{1,2}d/ → “baed”, “bead”, “baad”, “bed”,..

Quantifiers

- The three kinds of regular expression quantifiers are *greedy*, *reluctant*, and *possessive*.
- The use of the *, ?, and + symbols
  - Greedy: ?, *, +, {n}, {n,m}, {n,}
  - Reluctant: ??, *?, +?, {n}?, {n,m}?, {n,}?
  - Possessive: ?+, *+, ++, {n}+, {n,m}+, {n,}+

    -
      ```
      var strToMatch = “abbbaabbbaaabbb123”;
      var re1 = /.*bbb/g; //greedy
      var re2 = /.*?bbb/g; //reluctant
      var re3 = /.*+bbb/g; //possessive

      strToMatch.match(re1); // “abbbaabbbaaabbb”
      strToMatch.match(re2); // “abbb”, “aabbb” and “aaabbb”
      strToMatch.match(re3); // null
      ```

- Some browsers don’t support for the possessive quantifier.

Grouping

- To handle character sequences instead of individual characters, regular expressions support grouping.
- Grouping is used by enclosing a set of characters, character classes, and/or quantifiers inside of a set of parentheses.
  - □ /dogdog/g = /**(**dog**)**{2}/g → “dogdog”
  - □ /(mom(and dad)?)/ → “mom” or “mom and dad”
- A practical example:
  - □ Let’s add your own trim() method to the String object.
  - □
    ```
    String.prototype.trim = function () {
        var reExtraSpace = /^\s+(.*?)\s+$/;
        return this.replace(reExtraSpace, "$1");
    };
    ```
    - \s → A white-space character
    - ^ → Beginning of the line
    - $ → End of the line

Backreferences

- Each group is stored in a special location for later use. These special values, stored from your groups, are called *backreferences*.
- To use the backreferences:
  - The values of the backreferences can be obtained from the RegExp constructor itself. (RegExp.$1-$9).
    - var reNumbers = /#(\d+)/;
reNumbers.test("#123456789");
alert(**RegExp.$1**); //ouputs "123456789"
  - Backreferences can also be included in the expression that defines the groups. Use the special escape sequences \1, \2, and so on.
    - /(dog)**\1**/ = /dogdog/
  - Backreferences can be used with the String's replace() method by using the special character sequences $1, $2, and so on.
    - var reMatch = /(\d{4}) (\d{4})/;
var result = "1234 5678".replace(reMatch, "**$2 $1**");
//returns "5678 1234"

Non-capturing Groups

- Groups that create backreferences are called *capturing groups*. There are also *non-capturing groups*, which don’t create backreferences.
- Use non-capturing groups to avoid the overhead of storing the results in long regular expressions.
- To create a non-capturing group, just add a question mark followed by a colon immediately after the opening parenthesis.

  -
    ```
    var reNumbers = /#(?:\d+)/;
    reNumbers.test(“#123456789”);
    alert(RegExp.$1); //ouputs “”
    ```
- A practical example:
  - Let’s create your own stripHTML() method for a String.

  -
    ```
    String.prototype.stripHTML = function () {
        var reTag = /<(?:.|\s)*?>/g;
        return this.replace(reTag, "");
    };
    ```

Alternation

- Alternation can be used to match a single regular expression out of several possible regular expressions.
- The alternation operator is the same as the ECMAScript bitwise OR, a pipe(|), and it is placed between two independent patterns.
  - □
    ```
    var reCatDog = /\b(cat|dog)\b/;
    var result = “cat dog mouse hotdog”.match(reCatDog);
    //returns an array containing “cat” and “dog”.
    ```
- A pratical example:
  - □ Let’s remove inappropriate words from user input.
  - □
    ```
    function filterText(sText) {
        var reBadWords = /badword|anotherbadword/gi;
        return sText.replace(reBadWords, function (sMatch) {
            return sMatch.replace(/./g, "*");
        });
    }
    ```

Lookaheads

- *Lookaheads* are used to capture a particular group of characters only if they appear before another set of characters.
    - JavaScript does not support *lookbehinds*.
- Positive lookaheads
    - Created by enclosing a patter between (?= and).
    - var reBed = /(bed(**?=**room))/;
      alert(reBed.test("bedroom")); //outputs “true”
      alert(reBed.test("bedding")); //outputs “false”
      alert(RegExp.$1); //outputs “bed”
        - A lookahead is not returned as part of group.
- Negative lookaheads
    - Created by enclosing a patter between (?! and).
    - var reBed = /(bed(**?!**room))/;
      alert(reBed.test("bedroom")); //outputs “false”
      alert(reBed.test("bedding")); //outputs “true”

Boundaries

- *Boundaries* are used in regular expressions to indicate the location of a pattern.
  - ^ → Beginning of the line
  - $ → End of the line
  - \b → Word boundary
  - \B → Non-word boundary
- Examples:

  -
    ```
    var reLastWord = /(\w+)\.$/;
    reLastWord.test(“Important word is the last one.”);
    alter(RegExp.$1); //outputs “one”
    ```

  -
    ```
    var strMatch = “first second third fourth fifth sixth”;
    var reWords = /\b(\S+?)\b/g;
    var result = strMatch.match(reWords); //returns an array
    “first”, “second”, “third”, “fourth”, “fifth” and “sixth”.
    ```
    - It is easier to use the word character class(\w):
      ```
      var reWords = /(w+)/g;
      ```

Multiline Mode

- If there are multiple lines contained in a string, you can specify *multiline mode* adding an *m* to the options of the regular expression.

- Examples:
  - □ var strMatch = “first second\nthird fourth\nfifth sixth”;
    var reWords = /(\w+)$/g;
    var result = strMatch.match(reWords); // returns an array contains only “sixth”.

  - □ var strMatch = “first second\nthird fourth\nfifth sixth”;
    var reWords = /(\w+)$/g**m**;
    var result = strMatch.match(reWords); //returns an array containing “second”, “fourth” and “sixth”.

The RegExp Object

- Instance properties:
  - global, ignore, lastIndex, multiline, source, etc.
- Static properties:
  - input ($_), lastMatch($&), lastParen($+), leftContext($`), rightContext($’)
  - Backrefercences: $1, $2,…, $9
  - An example:

    -
      ```
      var reShort = /(s)ho(rt)/g;
      reShort.test("bbq is short for barbecue“);
      alert(RegExp.$_); //outputs “bbq is short for barbecue”
      alert(RegExp[“$&”]); //outputs “short”
      alert(RegExp[“$+”]); //outputs “rt”
      alert(RegExp.leftContext); //outputs “bbq is ”
      alert(RegExp.rightContext); //outputs “ for barbecue”
      alert(RegExp.$1); //outputs “s”
      alert(RegExp.$2); //outputs “rt”
      ```

A Practical Example

- Validating e-mail addresses
  - An valid e-mail satisfies that:
    - at least one character must precede the at(@) symbol,
    - and at least three must come after it,
    - the second of which must be a period.
  - The regular expression is the following:
    - var reEmail = /^(?:\w+\.?)*\w+@(?:\w+\.)+\w+$/;
    - (?:\w+\.?) → One or more word characters followed by zero or one period.
    - \w+@ → A word character is always before the @.
    - (?:\w+\.) → One or more word characters followed by one period.
    - \w+$ → A word character must be the last character.

  -
    ```
    function isValidEmail(sText) {
        var reEmail = /^(?:\w+\.?)*\w+@(?:\w+\.)+\w+$/;
        return reEmail.test(sText);
    }
    ```

Error Handling

Handling Errors

- JavaScript offers two specific ways to handle errors:
  - The BOM includes the *onerror* event handler on both the window object and on images;
  - The 3rd edition of ECMAScript implements the *try…catch…finally* construct as well as *throw* statement to deal with exceptions.

- Errors vs. Exceptions
  - *Syntax errors*, also called *parsing errors*, occur at compile time for traditional programming languages and at interpret time for JavaScript.
  - *Runtime errors*, also called *exceptions*, occur during execution after compilation or interpretation.

The onerror Event Handler

- The onerror event is fired on the window object whenever an exception occurs on the page.
  - window.onerror = function() { alert(“An error occurred.”); }
  - To hide the browser’s error message, return a value of true.

- Error information is passed as three parameters into the onerror event handler:
  - Error message, URL and line number.

    -
      ```
      Window.onerror = function(sMsg, sUrl, sLine) {
          alert(“An error occurred:\n” + sMsg + “\nURL:” +
              sURL + “\nLine Number:” + sLine);
          return true;
      }
      ```
  - The image’s onerror event handler doesn’t pass any arguments for error information.

The try...catch Statement

- The basic syntax:

  -
    ```
    try {
        //code to run
    } catch ([exception]) {
        //code to run if an exception occurs.
    } [finally {
        //code that is always executed.
    }]
    ```
  - Unlike Java, the ECMAScript standard specifies only one catch clause per try...catch statement.
- The Error Object
  - Properties:
    - name – A string indicating the type of error
    - message – The actual error message
  - Subclasses:
    - EvalError, RangeError, ReferenceError, SyntaxError, TypeErro, URIError

Determining the Type of Error

- Using the *name* property of the Error object:

```
try {
    eval("a ++ b"); //causes SyntaxError
} catch (oException) {
    if (oException.name == "SyntaxError") {
        alert("Syntax Error: " + oException.message);
    } else {
        alert("An exception occurred: " + oException.message);
    }
}
```

- Using the *instanceof* operator and use the class name of different errors:

```
if (oException instanceof SyntaxError) {
    alert("Syntax Error: " + oException.message);
} else {
    alert("An exception occurred: " + oException.message);
}
```

Raising Exceptions

- The *throw* statement is used to raise exceptions purposely.
- The syntax:
  - throw *error_object*;
  - The error_object can be a string, a number, a Boolean value, or an actual object.
    - throw “error1”;
    - throw 5001;
    - throw new SyntaxError(“I don’t like your syntax.”);
- A practical example:

  -
    ```
    function addTwoNumbers(a, b) {
        if (arguments.length < 2) {
            throw new Error(“Two numbers are required.”);
        } else {
            return a + b;
        }
    }
    ```

References

Professional JavaScript for Web Developers – wrox.com

http://javascript.crockford.com

http://www.ecma-international.org

www.wikipedia.com

To be continued...

BOM, DOM, Events, Forms, Drag & Drop,
XML, Client-Server Communication(AJAX),
Plugins, Deployment Issues, Etc.

Questions?