JavaScript Basics

The Big Picture

JavaScript had to “look like Java” only less so— be Java’s dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JavaScript would have happened. —Brendan Eich, creator of JavaScript

Brendan Eich proposed embedding the Scheme language in the browser (Seibel 2009). Although pressure to create a Java-like syntax prevailed, many Scheme ideas survive in JavaScript.

History

- 1995: Netscape includes LiveScript JavaScript as browser scripting language
- Originally, for simple client-side code such as animations and form input validation
- Document Object Model (DOM) lets JavaScript inspect & modify document elements
- 1997: standardized as ECMAScript
- 1998: Microsoft adds XmlHttpRequest to IE5
- 2005: Google Maps, AJAX takes off

JavaScript Tools

- JavaScript has unnecessary bad rep… maybe
- Presence of JS should enhance app; lack of JS shouldn't kill it
- jQuery is the way to do client-side JS enhancements
- React/Angular better for single-page apps?
- Jasmine/TDD just as important for JS as for server
- Jury still out on server-side JS… not really
- TypeScript looks like a potential enhancement/alternative

TypeScript

TypeScript is an open source syntactic superset of JavaScript that compiles to JavaScript (EcmaScript 3+). TypeScript offers type annotations which provide optional, static type checking at compile time. Since it is a superset of JavaScript, all JavaScript is syntactically valid TypeScript. However, that does not mean all JavaScript can actually be processed by the TypeScript compiler.

Objectives

- To see how scripts can be embedded in HTML
- Be able to use basic JavaScript programming constructs
- Get user input and display output using JavaScript
- Use objects, arrays, and functions in JavaScript
- Use regular expressions to do pattern matching

JavaScript Overview and Basics

Overview of JavaScript

JavaScript is a programming language used for creating dynamic web documents

Allows for:

- Accessing and modifying any elements within the HTML page
- Reacting to events that occur within the web page (e.g. mouse clicks, keyboard presses, etc.)

Main uses:

- Validate form input
- User communication
- Game development (HTML5)
- Updating parts of a page (AJAX)

History of JavaScript

Originally developed by Netscape, as **LiveScript**

Became a joint venture of Netscape and Sun in 1995, renamed JavaScript

Now standardized by the European Computer Manufacturers Association as **ECMA-262** (also ISO 16262)

We'll call collections of JavaScript code ***scripts***, not programs

General Syntactic Characteristics

JavaScript scripts can be embedded in HTML documents

Either directly, as in

```
<script type = "text/javaScript">
        -- JavaScript script –
</script>
```

Or indirectly, as a file specified in the `src` attribute of `<script>`, as in

```
<script type = "text/javaScript"
      src = "myScript.js">
</script>
```

Language Basics

Scripts are a sequence of ***statements*** that consist of identifiers.

Identifiers are words used in the script (e.g. names of variables, functions, etc.)

Each statement is an instruction (set) for the ***interpreter***.

In JavaScript, the browsers usually contain the interpreter (aka ***JavaScript engine***).

Language Basics

Identifier form

- Begin with a letter or underscore, followed by any number of letters, underscores, and digits
- Case sensitive

25 ***reserved words*** plus future reserved words

- These cannot be used by the programmer for variable names, function names, etc.

Comments:

- Text that is ignored by the interpreter.
- Two kinds: // and /* … */

General Syntactic Characteristics

Semicolons indicate an end of a statement.

They are "somewhat" optional

Problem: when the end of the line can be the end of a statement – JavaScript puts a semicolon here

(this may not be what you want)

General Syntactic Characteristics

Scripts are usually hidden from browsers that do not include JavaScript interpreters by putting them in special comments:

```
<!--
-- JavaScript script –
//-->
```

Also required by the HTML validator

Storing and Manipulating Data

Primitives and Variables

A script allows for **storing data** in computer's (client's) main memory.

Data can be stored by declaring a ***variable*** and then initializing it with a value.

Primitives and Variables

Values can be assigned to by the **= operator**.

This is called the ***assignment statement***.

General form: **LHS=RHS**

(causes **RHS** value to be stored in **LHS**)

Example: `var x = 5;`

Data Types

Each variable can store different **data types**

JavaScript has five ***primitive types***:

- Number, String, Boolean, Undefined, Null
- These store a single piece of data

Number type stores double-precision floating point values

String types store sequences of characters

- delimited by either ' or "
- Can include escape sequences (e.g., `\t`)
- All String literals are primitive values

Data Types

Boolean values are logical values of `true` and `false`

The only ***Null*** value is `null`

Variables that did not have any value assigned will have an ***Undefined*** type with a value of `undefined`

Fixed values of different data types can be included in the script as ***literals***

Object Orientation and JavaScript

There are also ***Object types*** that can store multiple properties.

Properties of objects are either primitive types or other objects.

JavaScript objects are collections of properties, which are like the members (attributes) of classes in Java and C++

The root object in JavaScript is `Object`

(All objects are derived from `Object`)

An object variable stores a reference to the data of the object.

- All JavaScript objects are accessed through references

Wrapper Objects

Sometimes it is useful for the primitive data types to be treated like Object types

Number, **String**, and **Boolean** have ***wrapper objects*** that do just that (called `Number`, `String`, and `Boolean`)

In the cases of Number and String, primitive values and objects are **coerced** back and forth so that primitive values can be treated essentially as if they were objects

Primitives and Variables

JavaScript is dynamically typed

- Any variable can be used for anything (primitive value or reference to any object)
- The interpreter determines the type of a particular occurrence of a variable

Variables can be either implicitly or explicitly declared

```
var sum = 0;
today = "Monday";
flag = false;
```

Note: JavaScript and Java are only related through syntax

- Since JavaScript is dynamically typed, there is **no need to declare a type before using variable**
- JavaScript's support for objects is very different from Java

Operations and Expressions

Values (either literals or variables) can be manipulated using expressions

Expressions are combinations of operations and operands

Numeric operators: `++`, `--`, `+`, `-`, `*`, `/`, `%`

All operations are in double precision

Same precedence and associativity as Java:

`++,--` then `*,/,%,` then `+,-`

Operations and Expressions

Expression examples:

```
2 + 4 - 3
x + 7 * 15 / y % 3
x++ + 3
```

More advanced math functions can be performed using the `Math` Object

The `Math` Object provides `floor`, `round`, `max`, `min`, trig functions, etc., e.g.:

```
Math.cos(x)
Math.round(3.43)
```

Operations and Expressions

The `Number` Object provides some useful properties:

`MAX_VALUE`, `MIN_VALUE`, `NaN`,
`POSITIVE_INFINITY`, `NEGATIVE_INFINITY`

e.g., `Number.MAX_VALUE`

An arithmetic operation that creates overflow returns `NaN`

- You can test for it with `isNaN(x)`

Operations and Expressions

Expressions can also involve String types

Strings have a ***concatenation operator*** (+) that appends one String to another, e.g.:

```
"hello " + "world"
```

Note that concatenation **coerces** numbers to strings, e.g.:

```
s = "5" + "5";
```

(stores "55" in variable s)

Operations and Expressions

If either operand of `+` is a string, it is assumed to be concatenation

Numeric operators (other than `+`) coerce strings to numbers, e.g.:

```
s = "5" - "5";
```

(stores 0 in variable s)

Conversions that do not work return `NaN`

You can also make explicit conversions

- Using the `String` and `Number` constructors
- Using `toString` method of numbers
- Using `parseInt` and `parseFloat` on strings

Input and Output

Input and Output

In JavaScript the HTML document is accessed through the ***Document object*** (named `document`)

The Document object has a method, **`write`**, which dynamically creates (HTML) content

The parameter (string) is sent to the browser, so it can be anything that can appear in an HTML document, e.g.:

```
document.write("Answer:"+result+"<br />");
```

Debugging JavaScript

Internet Explorer

- Hit “F12” to open developer tools (console)

MS Edge

- CTRL+SHIFT+I to open the JavaScript console

Google Chrome

- CTRL+SHIFT+J to open the JavaScript console

Firefox

- CTRL+SHIFT+K to open web console

Input and Output

The browser display window can be accessed through the ***`Window` object***

The `Window` object has several properties:

One property is `document` – it refers to the `Document` object inside the window

All variables globally declared are part of the window object

In references, window is the implied global context (so `document.write` is actually `window.document.write`)

Window I/O

The `Window` object has three methods for creating ***dialog boxes***

- `alert`
- `confirm`
- `prompt`

The ***alert method*** opens a dialog box which displays the **parameter** string and an OK button (it waits for the user to press the OK button), e.g.:

```
alert("Hey! \n");
```

Note that the parameter is plain text, not HTML

Window I/O (continued)

The ***confirm method*** opens a dialog box and displays the parameter and two buttons, OK and Cancel

- It returns a Boolean value, depending on which button was pressed (it waits for one), e.g.:

```
confirm("Do you want to continue?");
```

The ***prompt method*** opens a dialog box and displays its string parameter, along with a text box and two buttons, OK and Cancel

- The second parameter is for a default response if the user presses OK without typing a response in the text box (waits for OK), e.g.:

```
prompt("What is your name?", "");
```

Useful Objects and Operators

Accessing Object Methods and Properties

JavaScript provides a couple of objects that have useful methods and properties:

- ***String object*** provides many methods for manipulating and getting information about Strings
- ***Date object*** is used for retrieving a String containing a date in various formats

String properties and methods

Here are some important String properties and methods:

`length` - a property that contains the String length

```
var len = str1.length;
```

`charAt(`position`)` - a method that returns a character at a specific position

```
str.charAt(3)
```

`indexOf(`string`)` - a method that returns the position of a specific character

```
str.indexOf('B')
```

`substring(`from, to`)` - returns a substring from and to a given position

```
str.substring(1, 3)
```

`toLowerCase()` - returns a lower case version of the String

```
str.toLowerCase()
```

The `Date` Object

The Data object is useful for getting the current time

To use it, first create one with the `Date` constructor (no params):

```
var d = new Date();
```

Then you can access various methods:

- `toLocaleString` – returns a string of the date
- `getDate` – returns the day of the month
- `getMonth` – returns the month of the year (0 – 11)
- `getDay` – returns the day of the week (0 – 6)
- `getFullYear` – returns the year
- `getTime` – returns the number of milliseconds since January 1, 1970
- `getHours` – returns the hour (0 – 23)
- `getMinutes` – returns the minutes (0 – 59)
- `getMilliseconds` – returns the millisecond (0 – 999)

The `typeof` operator

Sometimes it may be necessary to know the type of a given variable

This can be done using the **`typeof` operator**, which returns one of the following Strings:

- `"number"` for Number objects
- `"string"` for String objects
- `"boolean"` for Boolean objects
- `"undefined"` for variables that have not been assigned a value
- "object" for objects
- `"function"` for functions

Changing the Flow of Control

Changing Flow of Control

Unless specified, programs are executed in sequence (line by line)

You can modify the sequence (called the ***flow of control***) by inserting ***conditional statements***

Conditional statements determine which instruction (statement) should be executed next based on a ***logical (Boolean) expression***

Logical Expressions

Logical expressions evaluate to a ***Boolean value*** (true or false)

3 kinds of logical expressions:

- Primitive values
- Relational expressions
- Compound expressions

Primitive values are strings or numbers

- If it is a string, it is **true** unless it is empty or "0"
- If it is a number, it is **true** unless it is zero

Logical Expressions

Relational Expressions are composed of operators that compare values and return a true or false Boolean value

JavaScript has the usual six comparison operators:

```
==  (equal)
!=  (not equal)
<   (less than)
>   (greater than)
<=  (less than or equal to)
>=  (greater than or equal to)
```

Operands are coerced if necessary

- If one is a string and one is a number, it attempts to convert the string to a number
- If one is Boolean and the other is not, the Boolean operand is coerced to a number (1 or 0)

Logical Expressions

JavaScript also has two unusual operators: `===` and `!==`

The semantics are the same as `==` and `!=`, except that no coercions are done (**operands must be identical**)

Note that comparisons of references to objects are not useful (addresses are compared, not values)

Logical Expressions

Compound Expressions are composed of ***Boolean operators*** that are a function of one or more truth values

JavaScript has the usual operators:

```
&&  (AND)
||  (OR)
!   (NOT)
```

The function of each operator can be expressed using a ***truth table***:

X	Y	!X	X && Y	X || Y
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

Selection Statements

Boolean expressions can be used as a condition for which statements will execute

`if-then-else` statement selects one of two possible paths for code execution:

Syntax:

```
if (condition)
   …
else
   …
```

Semantics: if condition is true, then execute the following statement(s); otherwise, execute statement(s) in the else clause

Clauses can be either single statements or compound statements, delimited by **braces - { }**

Switch

The **switch** statement evaluates an expression and changes to the flow of control based on a matching case

Syntax:

```
switch (expression) {
    case c1: statement(s);
             break;
    case c2: statement(s);
             break;
    …
    case cn: statement(s);
             break;
    default: statement(s);
 }
```

Switch

The statements can be either statement sequences or compound statements

The control expression can be a number, a string, or a Boolean

Different cases can have values of different types

Loop Statements

Parts of the code may be repeated using ***loop statements***

The statements inside the loop will be executed until some condition is satisfied

Loop Statements

`while` (condition)
    statement(s)

`do`
    statement(s)
    `while` (condition)

`for` (init; condition; increment)
    statement(s)

- init can have declarations, but the scope of such variables is the whole script

Using Objects

Object Creation and Modification

Objects can be created with the **`new`** operator

The most basic object is one that uses the `Object` constructor:

```
var myObject = new Object();
```

The new object has no properties - a blank object

Properties can be added to an object, any time

```
var myAirplane = new Object();
myAirplane.make = "Cessna";
myAirplane.model = "Centurian";
```

Object Creation and Modification

An alternative method for creating objects is to enclose property list in curly brackets

Example:

```
var my_car = {make: "Saturn", model: "Aura"};
```

Object Creation and Modification

Objects can be nested, so a property could be itself another object, created with `new`

Can delete properties using **`delete`** statement

Properties can be accessed by dot notation or in array notation:

```
var property1 = myAirplane["model"];
delete myAirplane.model;
```

Iterator

A special type of a for (**foreach**) loop statement can be used to iterate across properties of an object

Syntax:

`for (`identifier `in` object`)`

statement or compound

Example:

```
for (var prop in myAirplane)
    document.write(myAirplane[prop] + "<br />");
```

Arrays

Arrays

Arrays are objects that can be used to hold a set of elements

Array elements can be primitive values or references to other objects including other arrays

Length is dynamic - the `length` property stores the length

Array objects can be created in two ways: using the new operator or by assigning an array literal, e.g.:

```
var myList = new Array(24, "bread", true);
var myList2 = [24, "bread", true];
var myList3 = new Array(24);
```

Arrays

The length of an array is the highest subscript to which an element has been assigned, plus 1

```
myList[122] = "bitsy"; // length is 123
```

Because the length property is writeable, you can set it to make the array any length you like, e.g.:

```
myList.length = 150;
```

Assigning a value to an element that does not exist creates that element

Array Methods

join

```
var listStr = list.join(", ");
```

reverse

sort

```
names.sort();
```

- Coerces elements to strings and puts them in alphabetical order

concat

```
newList = list.concat(47, 26);
```

Array Methods (cont.)

slice

```
listPart = list.slice(2, 5);
listPart2 = list.slice(2);
```

toString

- Coerce elements to strings, if necessary, and concatenate them together, separated by commas (exactly like `join(", ")`)

push, pop

- add/remove element from end of the list

unshift, shift

- add/remove element from the beginning of the list

Functions

Functions

Sections of code that perform a similar task can be grouped together as a ***function***

Functions take parameters as **input**, execute statements in the function **body**, and return an **output** value

Syntax:

```
function function_name([formal_parameters]) {
    -- body --
}
```

Return value is the parameter of a `return` statement

- If there is no `return`, or if the end of the function is reached, `undefined` is returned
- If `return` has no parameter, `undefined` is returned

Functions

Functions are objects, so variables that reference them can be treated as other object references

e.g.: if `fun` is the name of a function, we can assign it to a different variable and invoke the function using it:

```
ref_fun = fun;
…
ref_fun();  /* A call to fun */
```

We place all **function definitions** in the head of the HTML document (calls are usually in document body)

Functions and Variables

Scope is the set of variables, objects, or functions that are accessible

All variables that are either implicitly declared or explicitly declared outside functions have **global scope** (i.e. can be accessed anywhere)

Variables explicitly declared in a function have **local scope** (i.e. can be accessed only within that function)

Function parameters

Function parameters in JavaScript are passed by value

Pass by value means that actual parameters (ones specified in the function call) are copied into the formal parameters (ones specified in the function definition)

- With pass by value, any changes to the variables inside the function will have no effect outside of the function

However, when a reference variable is passed, the semantics are ***pass-by-reference***

- Pass-by-reference semantics means that changing an object's property will affect the object outside of the function

Function parameters

What if you wanted the function to change a value passed in (have pass-by-reference semantics)?

One (dirty) way is to put the value in an array and send the array's name:

```
function by10(a) {
      a[0] *= 10;
}
...
var listx = new Array(1);
...
listx[0] = x;
by10(listx);
x = listx[0];
```

Function parameters

A few comments about parameter passing:

There is **no type checking of parameters**, nor is the number of parameters checked

- excess actual parameters are ignored
- excess formal parameters are set to `undefined`

All parameters are sent through a property array, `arguments`, which has the length property

Constructors

Constructors

Constructors are special functions used to initialize objects and their properties

They take on the name of the object to be created

Example:

```
function plane(newMake, newModel, newYear){
    this.make = newMake;
    this.model = newModel;
    this.year = newYear
}

myPlane = new plane("Cessna", "Centurian", "1970");
```

Constructors

Constructors can also add function properties:

```
function displayPlane() {
  document.write("Make: ",this.make,"<br />");
  document.write("Model: ",this.model,"<br />");
  document.write("Year: ",this.year,"<br />");
}
```

To include in object add the following to the constructor:

```
this.display = displayPlane;
```

Pattern Matching

Pattern Matching

A common use of JavaScript is to validate whether form input has the right format

Validation can be done by ***pattern matching***, which involves looking for a specific sequence of characters in a String

JavaScript provides two ways to do pattern matching:

- Using `RegExp` objects
- Using methods on `String` objects

Pattern Matching

Patterns can be specified using ***regular expressions*** that are a sequence of special characters which denote a pattern

There are two categories of characters in patterns:

- normal characters (match themselves)
- **metacharacters** (can have special meanings in patterns):

  \ | ( ) [ ] { } ^ $ * + ? .

Patterns are delimited with forward slashes / /

Pattern Matching

Examples:

`/\d\d-\d\d-\d\d/` - matches a date, e.g. 08-20-14

`/Lewis.*/` - matches anything that starts with `"Lewis"`

Period is a special metacharacter – it matches any character except newline, e.g.:

`/…/` - matches any three characters except newlines

A metacharacter is treated as a normal character if it is backslashed, e.g.:

`/\./` - matches a period

Character classes

Brackets are used to define a set of characters, any one of which matches

```
[abcd]
```

Dashes can be used to specify spans of characters in a class

```
[a-z]
```

A **caret** at the left end of a class definition means the opposite

```
[^0-9]
```

Predefined character classes

Name	Equivalent Pattern	Matches
\d	[0-9]	a digit
\D	[^0-9]	not a digit
\w	[A-Za-z_0-9]	a word character
\W	[^A-Za-z_0-9]	not a word character
\s	[\r\t\n\f]	a whitespace character
\S	[^ \r\t\n\f]	not a whitespace character

Character classes – Quantifiers

Quantifier	Meaning
{n}	exactly n repetitions
{m,}	at least m repetitions
{m, n}	at least m but not more than n repetitions

Pattern matching – Quantifiers

Other quantifiers (just abbreviations for the most commonly used quantifiers):

***** means zero or more repetitions

Ex: `\d*` means zero or more digits

+ means one or more repetitions

Ex: `\d+` means one or more digits

? Means zero or one

Ex: `\d?` means zero or one digit

Pattern matching – Anchors

The ***anchor operators*** (`^` and `$`) are used to match positions, at the beginning or end

The pattern can be forced to match only

- at the beginning with `^`
- at the end with `$`

Examples:

`/^Lee/` matches `"Lee Ann"` but not `"Mary Lee Ann"`

`/Lee Ann$/` matches `"Mary Lee Ann"`, but not

`"Mary Lee Ann is nice"`

Pattern matching – Grouping

Place parenthesis around multiple tokens to group them together

You can then apply a quantifier to the group

Example:

`/(cat)+/` - matches one of more of `"cat"`

Pattern Matching – Alternation

Alternation is the regular expression equivalent of "or"

The | symbol is used for alternation

Example:
`/^(cat|dog)$/`
matches `cat` or `dog`

Pattern modifiers

The **`i` modifier** tells the matcher to ignore the case of letters

Example:

`/oak/i` matches `"OAK"` and `"Oak"`

Pattern Matching

The simplest way to search for patterns is to use the **search method** of String

Syntax:

`search(`pattern`)`

It **returns the position** in the object string of the pattern (position is relative to zero) or `-1` if it fails

Example:

```
var str = "Gluckenheimer";
var position = str.search(/n/);
/* position is now 6 */
```

Other Pattern Matching Methods of String

The ***replace method*** finds a substring that matches the pattern and replaces it with the string

Syntax:

`replace(`pattern, string`)`

`g` modifier can be used – done for every match in the string

Example:

```
var str = "Some rabbits are rabid";
str = str.replace(/rab/g, "tim");
```

`str` is now `"Some timbits are timid"`

Other Pattern Matching Methods of String

The ***match method*** is the most general pattern-matching method

Syntax:

`match(`pattern`)`

It **returns an array of results** of the pattern-matching operation

With the `g` modifier, it returns an array of the substrings that matched

Without the `g` modifier, first element of the returned array has the matched substring, the other elements have the values of `$1`, …

Other Pattern Matching Methods of String

Match method example:

```
var str = "My 3 kings beat your 2 aces";
var matches = str.match(/[ab]/g);
```

`matches` is set to `["b", "a", "a"]`

Other Pattern Matching Methods of String

Another useful method is ***split***, which divides a String into an array of substrings

Syntax:

`split(`parameter`)`

The parameter is a character or regular expression that is used as a separator (so `","` and `/,/` both work)

Example:

```
var str = "How are you?";
var res = str.split(" ");
```

`res` will then contain the following values:

```
["How", "are", "you?"]
```

The `RegExp` object

Another way of pattern matching in JavaScript is using the **`RegExp` object**

The `RegExp` object is used to store the regular expression

Declaring a `RegExp` object:

- Use `new`

```
var regex = new RegExp("\\w*");
```

- Or assign a pattern

```
var regex = /\w*/;
```

Methods of the RegExp Object

Once the RegExp object is defined, we can search for patterns using one of 2 methods:

The **`test()` method** searches a string for a specified value and returns true or false, e.g.:

```
if (regex.test("some string to search"))
```

The **`exec()` method** searches a string for a specified value, e.g.:

```
var res = regex.exec("some string to search")
```

Summary

- JavaScript is a scripting language that can be embedded in HTML
- Main uses for JavaScript are input validation, communication, page updates, and implementing functionality of the HTML5 canvas element
- JavaScript can store five different primitive data types as well as objects, arrays, and functions
- HTML can be output using the write method of the document object
- User input can be done via alert, confirm, or prompt dialog boxes
- JavaScript objects can contain multiple properties which may be added dynamically
- JavaScript is dynamically typed and uses call-by-value to pass function parameters
- Regular expressions are a sequence of special characters used for denoting a pattern
- In JavaScript, we can search for patterns using methods of String or using the RegExp object