4.1 Overview of JavaScript

- Originally developed by Netscape by Brendan Eich, 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)

- This chapter covers *client-side* JavaScript

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

- JavaScript and Java are only related through syntax

  - JavaScript is dynamically typed

  - JavaScript's support for objects is very different

4.1 Overview of JavaScript (continued)

- User interactions through forms are easy

- The Document Object Model makes it possible to support dynamic HTML documents with JavaScript

- Much of what we will do with JavaScript is event-driven computation – covered in Chapter 5

- When JavaScript is interpreted and where

4.2 Object Orientation and JavaScript

- JavaScript is NOT an object-oriented programming language

  - Does not support class-based inheritance
    - Cannot support polymorphism

  - Has prototype-based inheritance, which is much different

- JavaScript objects are collections of *properties*, which are like the members of classes in Java and C++

- JavaScript has primitives for simple types

- The root object in JavaScript is `Object` – all objects are derived from `Object`

- All JavaScript objects are accessed through references

4.3 General Syntactic Characteristics

- For this book, all JavaScript scripts will 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*:

  - *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

  - Comments: both `//` and `/* … */`

4.3 General Syntactic Characteristics

(continued)

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

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

- Also hides it from HTML validators

- Semicolons can be a problem

- They are "somewhat" optional

- *Problem*: When the end of the line can be the end of a statement – JavaScript puts a semicolon there

4.4 Primitives, Operations, & Expressions

- All primitive values have one of the five primitive types: Number, String, Boolean, Undefined, or Null

4.4 Primitives, Operations, & Expressions (continued)

- Number, String, and Boolean have wrapper objects (`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

- Numeric literals – just like Java

- All numeric values are stored in double-precision floating point

- String literals are delimited by either `'` or `"`

  - Can include escape sequences (e.g., `\t`)

  - All String literals are primitive values

4.4 Primitives, Operations, & Expressions (continued)

- Boolean values are `true` and `false`

- The only Null value is `null`

- The only Undefined value is `undefined`

- 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;
```


4.4 Primitives, Operations, & Expressions (continued)

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

  - All operations are in double precision
  - Same precedence and associativity as Java

- The `Math` Object provides `floor`, `round`, `max`, `min`, trig functions, etc.
  e.g., `Math.cos(x)`

- The `Number` Object

  - Some useful properties:
    `MAX_VALUE`, `MIN_VALUE`, `NaN`,
    `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `PI`
    - e.g., `Number.MAX_VALUE`

      - An arithmetic operation that creates overflow returns `NaN`

    - `NaN` is not == to any number, not even itself

    - Test for it with `isNaN(x)`

  - `Number` object has the method, `toString`

4.4 Primitives, Operations, & Expressions (continued)

- *String catenation operator* `+`

- *Coercions*
  - Catenation coerces numbers to strings

  - Numeric operators (other than `+`) coerce strings to numbers (if either operand of `+` is a string, it is assumed to be catenation)

    - Conversions from strings to numbers that do not work return `NaN`

- Explicit conversions
  1. Use the `String` and `Number` constructors
  2. Use `toString` method of numbers
  3. Use `parseInt` and `parseFloat` on strings

- *String properties & methods*:

  - `length` e.g., `var len = str1.length;` (a property)
  - `charAt(`position`)` e.g., `str.charAt(3)`
  - `indexOf(`string`)` e.g., `str.indexOf('B')`
  - `substring(`from, to`)` e.g., `str.substring(1, 3)`
  - `toLowerCase()` e.g., `str.toLowerCase()`

4.4 Primitives, Operations, & Expressions (continued)

- *The* `typeof` *operator*
  - Returns `"number"`, `"string"`, or `"boolean"` for Number, String, or Boolean, `"undefined"` for Undefined, `"function"` for functions, and `"object"` for objects and NULL

- *Assignment statements* – just like C++ and Java

- The `Date` Object

  - Create one with the `Date` constructor (no params)

  - Local time methods of `Date`:

    `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)

4.5 Screen Output & Keyboard Input

- The JavaScript model for the HTML document is the `Document` object

- The model for the browser display window is the `Window` object
  - The `Window` object has two properties, `document` and `window`, which refer to the `Document` and `Window` objects, respectively

- The `Document` object has a method, `write`, which dynamically creates content

  - The parameter is a string, often catenated from parts, some of which are variables

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

  - The parameter is sent to the browser, so it can be anything that can appear in an HTML document (`<br />`, but not `\n`)

- The `Window` object has three methods for creating dialog boxes, `alert`, `confirm`, and `prompt`

4.5 Screen Output (continued)

1. `alert("Hej! \n");`

   - Parameter is plain text, not HTML

   - Opens a dialog box which displays the parameter string and an OK button
     - It waits for the user to press the OK button

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

   - Opens a dialog box and displays the parameter and two buttons, OK and `Cancel`

   - Returns a Boolean value, depending on which button was pressed (it waits for one)

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

   - 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)

→ **SHOW `roots.html` and `roots.js`**

4.6 Control Statements

- Similar to C, Java, and C++

- Compound statements are delimited by braces, but compound statements are not blocks

- *Control expressions* – three kinds

  1. *Primitive values*

     - 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

  2. *Relational Expressions*

     - *The usual six*: `==`, `!=`, `<`, `>`, `<=`, `>=`

       - 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)

     - *The unusual two*: `===` and `!==`
       - Same as `==` and `!=`, except that no coercions are done (operands must be identical)

4.6 Control Statements (continued)

2. *Relational Expressions* (continued)

- Comparisons of references to objects are not useful (addresses are compared, not values)

3. *Compound Expressions*

- The usual operators: `&&`, `||`, and `!`

  - The `Boolean` object has a method, `toString`, to allow `Boolean` values to be printed (`true` or `false`)

- If a Boolean object is used in a conditional expression, it is false only if it is `null` or `undefined`

- *Selection Statements*

- The usual `if-then-else` (clauses can be either single statements or compound statements)

4.6 Control Statements (continued)

- *Switch*

```
switch (expression) {
   case value_1:
      // value_1 statements
   case value_2:
      // value_2 statements
   …
   [default:
      // default statements]
}
```

- 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

→ **SHOW** `borders2.js`

4.6 Control Statements (continued)

- *Loop statements*

`while` (control_expression) statement or cmpnd

`for` (init; control; increment) statement or cmpnd
- init can have declarations, but the scope of such variables is the whole script

→ SHOW `date.js`

`do`
statement or compound
`while` (control_expression)

4.7 Object Creation and Modification

- Objects can be created with `new`

- The most basic object is one that uses the `Object` constructor, as in

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

- The new object has no properties - a blank object

- Properties can be added to an object, any time

4.7 Object Creation and Modification (continued)

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

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

- Properties can be accessed by dot notation or in array notation, as in

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

```
delete myAirplane.model;
```

- *Another Loop Statement (an iterator)*

  - `for (`identifier `in` object`)` statement or compound

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


4.8 Arrays

- Objects with some special functionality

- Array elements can be primitive values or references to other objects

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

- Array objects can be created in two ways, with `new`, or by assigning an array literal

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

- 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, as in

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

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

→ SHOW `insert_names.js`

4.8 Arrays (continued)

- Array methods:
  - `join` – e.g., `var listStr = list.join(", ");`
  - `reverse`
  - `sort` – e.g., `names.sort();`
    - Coerces elements to strings and puts them in alphabetical order
  - `concat` – e.g., `newList = list.concat(47, 26);`
  - `slice`

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

  - `toString`
    - Coerces elements to strings, if necessary, and catenates them together, separated by commas (exactly like `join(", ")`)
  - `push`, `pop`, `unshift`, and `shift`

→ SHOW `nested_arrays.js`

4.9 Functions

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

```
  -- body -
}
```

- Return value is the parameter of `return`
  - 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 are objects, so variables that reference them can be treated as other object references

  - If `fun` is the name of a function,

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

- We place all function definitions in the head of the the XHTML document

- All variables that are either implicitly declared or explicitly declared outside functions are global

- Variables explicitly declared in a function are local

4.9 Functions (continued)

- Parameters are passed by value, but when a reference variable is passed, the semantics are pass-by-reference

- 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

→ SHOW `params.js` and output

- There is no clean way to send a primitive by reference

  - 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];
```


4.9 Functions (continued)

- To sort something other than strings into alphabetical order, write a function that performs the comparison and send it to the `sort` method

  - The comparison function must return a negative number, zero, or a positive number to indicate whether the order is ok, equal, or not ok

```
function num_order(a, b) {return a - b;}
```

  - Now, we can sort an array named `num_list` with:

```
num_list.sort(num_order);
```

4.10 An Example

→ SHOW `medians.js` & output

4.11 Constructors

- Used to initialize objects, but actually create the properties

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

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

- Can also have method properties

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

- Now add the following to the constructor:

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


4.12 Pattern Matching

- JavaScript provides two ways to do pattern matching:

  1. Using `RegExp` objects
  2. Using methods on `String` objects

- *Simple patterns*

- Two categories of characters in patterns:

  a. normal characters (match themselves)

  b. metacharacters (can have special meanings in patterns--do not match themselves)

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

  - A metacharacter is treated as a normal character if it is backslashed

  - period is a special metacharacter - it matches any character except newline

4.12 Pattern Matching (continued)

`search(`**pattern**`)`

- Returns the position in the object string of the pattern (position is relative to zero); returns -1 if it fails

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

- *Character classes*

- Put a sequence of characters in brackets, and it defines 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]
```


4.12 Pattern Matching (continued)

- Character classes (continued)

  - Character class abbreviations

Abbr.	*Equiv. 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

- *Quantifiers*

  - Quantifiers in braces

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

4.12 Pattern Matching (continued)

- *Quantifiers* (continued)

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

  - `*` means zero or more repetitions
    e.g., `\d*` means zero or more digits

  - `+` means one or more repetitions
    e.g., `\d+` means one or more digits

  - `?` Means zero or one
    e.g., `\d?` means zero or one digit

4.12 Pattern Matching (continued)

- *Anchors*

  - The pattern can be forced to match only at the left end with `^`; at the end with `$`

    e.g.,
    `/^Lee/`

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

    `/Lee Ann$/`

    matches `"Mary Lee Ann"`, but not
    `"Mary Lee Ann is nice"`

- The anchor operators (`^` and `$`) do not match characters in the string--they match positions, at the beginning or end

- *Pattern modifiers*

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

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

4.12 Pattern Matching (continued)

- *Pattern modifiers* (continued)

  - The `x` modifier tells the matcher to ignore whitespace in the pattern (allows comments in patterns)

- Other Pattern Matching Methods of `String`

  `replace(`**pattern**`,` **string**`)`

  - Finds a substring that matches the pattern and replaces it with the string (`g` modifier can be used)

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

`str` is now `"Some timbits are timid"`
`$1` and `$2` are both set to `"rab"`

4.12 Pattern Matching (continued)

`match(`**pattern**`)`

- The most general pattern-matching method
- 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`, …

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

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

`split(`**parameter**`)`

`","` and `/,/` both work

→ SHOW `forms_check.js`

4.13 Debugging JavaScript

- *IE9+*

  - Need to turn on syntax error notification and debugging

    - Select *Tools*/*Internet Options* and the *Advanced* tab. Under *Browsing*, remove the check on *Disable script debugging (Internet Explorer)* and set the check on *Display a notification about every script error*

      - Then a script error causes a small window to be opened with an explanation of the error

- *FX3+*

  - Select *Tools*, *Web Developer*, *Error Console*

    - A small window appears to display script errors

    - Remember to `Clear` the console after using an error message – avoids confusion

- *Chrome*

  - Select the wrench icon, *Tools*, *JavaScript console*

    - Produces an error console window

