JavaScript and the Document Object Model (DOM)
JavaScript is...

- A client-side scripting language
- Interpreted
- Weakly typed
- Prototype-based
- Used to add interactivity to websites
Client-side Scripting Language

- Client-side – Processed by the browser
  - Compare: “server-side”
- Scripting language – Controls the behavior of another application
  - Like, say, a browser
Interpreted Language

- Lexed and parsed by the browser
- Translated into intermediate bytecode
- Executed by the browser
- Advantage: No need to precompile, portability
- Disadvantage: Overhead of preprocessing
Strong Typing

- Variable types explicitly defined
- Language constructs must behave according to the rules of each type
- Type *coercion* (making one type behave as another) *must* be done manually
- Ex: Java, C++
Weak/Loose Typing

- Variable types not explicitly defined
- Type coercion performed depending on context automatically (may usually be done manually)
- Ex: PHP, JavaScript
Prototype-based Language

• Object-oriented – in fact, every non-primitive is an object

• No notion of classes!

• Behavior is reused by cloning pre-existing objects, or prototypes
JavaScript is useful...

- Dynamically change the page
  - Enhance forms
  - Improved ways of displaying content
  - Special effects (in moderation!)
- Asynchronous JavaScript and XML (AJAX) – Pattern used to make additional requests without leaving the page
...But we can't rely on it.

- Not all browsers implement the same functionality
  - Surprise!
- Users can disable JavaScript
  - Security concerns
  - Poorly-designed experiences
When not to use JavaScript

- When it can be done in CSS
- Form validation (when used exclusively)
- Any functionality that is *required* for the functioning of your site
Progressive Enhancement

- Start with content
- Annotate with semantic HTML
- Apply cross-browser CSS
  - Embellish CSS with less supported features after
- Add JavaScript to enhance the final experience for those users supporting it
Using JavaScript
Adding JavaScript: <script>

- Within the <script> tag
- Empty <script> element, with the src attribute pointing to a .js file (preferred)
- type="text/javascript"
  - Required in HTML 4.01/XHTML 1.0
  - Optional in HTML5
- Strongly recommended to use as *late* in the page as possible
<aside> JavaScript and XHTML

- Data inside a script tag will be treated as parsed character data (PCDATA) as XML by default!
- `<![CDATA[ content ]]>`
  - `content` is only character data (CDATA), not parsed as XML
- Another case for just using src, but if you really want to know...
<aside> JavaScript and XHTML

- <script type="text/javascript">//<![CDATA[
  alert('Statler & Waldorf');
//]]></script>
<aside> HTML Comments

- Some browsers didn't know how to interpret JavaScript
- Wrapping the script in an HTML comment hid them from these browsers
- Don't. We're not in 1990 anymore.
- Using -- operator may be parsed as the end of a comment
<aside> HTML Comments

• Some browsers didn't know how to interpret JavaScript

• Wrapping the script in an HTML comment hid them from these browsers

• Don't. We're not in 1990 anymore.

• Using -- operator may be parsed as the end of a comment
Writing JavaScript
Language Elements

- Comments
- Variables
- Values
  - Primitives
  - Objects
- Operators
- Control Structures
- Functions
JavaScript Comments

• // C++ Style Comments
• /* C-style Comments */
JavaScript Variables

- *Variables* are not the same as *values*
- Variables are containers for either values or pointers to values
- Declaring variables:
  - `var variable = value`
JavaScript Variable Scope

- Variables scoped at the function level, *not* the block level
- This may not work as you expect:

```javascript
function foo() {
    var x = 1;
    for (i = 0; i < 5; i++) {
        x++;
    }
    return x;
}
```
Global Scope

- Declaring variables without var...
  - Look for the variable name within scope
  - If it doesn't exist, variable defined on the global scope
- Always use the var keyword to avoid unexpected behavior!
JavaScript Values

- Primitives are stored completely within a variable
  - String - “foo”
  - Number (float) – 3.14159, 42
  - Boolean – true, false
  - Explicit null value - null
  - Declared but not assigned a value - undefined
JavaScript Values

- **Objects** are referred to by variables
  - When assigning `var obj2 = obj1`, only the reference is copied
  - `obj1` and `obj2` now point to the same object

- **Types of objects**
  - Primitive wrappers
  - Arrays
  - Other Predefined Objects
  - Functions
  - Object literals
JavaScript Values

- *Primitive wrappers* for strings, numbers, boolean *wrap* the primitive value while providing new methods of manipulating it

- Primitives will be coerced to primitive wrapper objects when necessary
Type Coercion

- When attempting to use a variable in a context where another type is expected, it will be automatically cast to the appropriate type.

- Example:

```javascript
var s = "foo";
/* s is a primitive string. When calling string manipulation functions on it, it is coerced to an object with the appropriate wrapper */
alert(s.length);
```
JavaScript Control Structures

- if/else if/else
- Ternary operator ( (bool_test) ? truevalue : falsevalue; )
- for
- while/do-while
- for..in
  - for (variable in object) { /* Iterate through object's properties */ }
JavaScript Arrays

- Arrays are objects that may be declared as an object or in array literal notation
  - var a = new Array();
    a[0] = "Foo";
    a[1] = 42;
  - var a = new Array("Foo", 42, true);
  - var a = ["Foo", 42, true]; // literal
- Also has array manipulation methods built in
  - push(), pop(), shift(), unshift(), etc.
- Can grow dynamically
Predefined Objects

- Many helpful objects are predefined by the language itself
- document represents the current page
- window represents the actual browser viewport
- Math contains helpful methods for advanced calculations
- etc.
Example: JavaScript Dates

• Predefined object meant to manipulate dates/times

• Multiple constructors...
  • var d = new Date();
  • var d = new Date(ms);
  • var d = new Date(“Jan 20, 2012”);
  • var d = new Date(yr, mo, d, hr, min, sec, ms);

• Get, set, locale, toString() methods
JavaScript Objects

- All other objects derive themselves from the base `Object()`
- Can be instantiated using the new keyword
  - `var o = new Object();`
  - **Note:** Instantiation via the new keyword is really cloning an existing object!
- Can be declared using object literal notation
  - `var o = {}; // literal`
JavaScript Objects

• Fundamentally, *objects* in JavaScript are just *hashes* (associative arrays)
  • Data structure with name => value pairs

• Can be nested

• `var myObj = {
    val1: "foo",
    val2: true,
    val3: 42,
    val4: ["bar", "baz", "bang"]
};`
JavaScript Objects

- Can refer to keys using `myObj.foo` or `myObj[“foo”]`
  - Latter is required if there's a dash in the name
- Can add keys to existing objects once declared
  - `myObj[“val5”] = {};
  - `myObj.val6 = null;`
  - Only applies to this instance!
JavaScript Objects

- Objects keys are either *properties* or *methods*
- *Properties* simply refer to a value
- *Methods* have a special Function() object as their value, which is evaluated when called
JavaScript Functions

• Functions in JavaScript are actually Function() objects
  • Includes object methods
  • Note: JavaScript is case-sensitive!
• return statement required at end
• Do not have to be declared in the global scope; can be nested (advanced)
• Declared a number of ways...
JavaScript Functions

- **Function statement**
  - `function myFunction(arg1, arg2) { return arg1*arg2; }
  - Parsed when loaded
  - Creates a variable with the same name as the function name in the scope it was declared

- **Dynamic Functions**
  - Declared at runtime
  - Assigned to a variable
Dynamic Functions

• Function constructor
  • var fn = new Function(“arg1”, “arg2”, “return arg1*arg2;”);
  • Not recommended...

• Function expression
  • var fn = function foo(arg1, arg2) {
      return arg1*arg2;
  };
  • alert(fn(2, 3));
Function Expressions

• Expression name can be used within function scope to refer to itself

```javascript
var fn = function foo(arg1, arg2) {
    if (arg1 < arg2) {
        alert("SWAP");
        return foo(arg2, arg1);
    }
    else {
        return arg1*arg2;
    }
};
alert(fn(2,3)); // SWAP, 6
```
Anonymous Functions

- Function expressions don't require a name
  - var fn = function(arg1, arg2) {
    return arg1*arg2;
  };
- Many potential uses for this...
Parameter Passing

- Function params are *passed by value*
- Remember, object references are also values!
  - This means the object referred to *can* be manipulated within a function
Recursion

- JavaScript supports recursion

```javascript
function factorial(n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```
Creating Objects - Properties

- Any named function can be used as a constructor when combined with the new keyword

  ```javascript
  function Student(fname, lname, rin) {
    this.firstName = fname;
    this.lastName = lname;
    this.rin = rin;
  }
  ```

  ```javascript
  var foo = new Student("Jane", "Smith", "6609999999");
  ```
<aside> this Keyword

- The special variable *this* always refers to the current object instance
- Sometimes this is referred to as the *activation object*
Creating Objects - Methods

• function Student(fname, lname, rin) {
  /* Initialize properties */
  this.toString = function() {
    return this.firstName + " " + this.lastName();
  };
}

var foo = new Student("Jane", "Smith", "6609999999");
alert(foo.toString());
<aside> Concatenation

• var a = 1 + 1; // 2
• var a = “1” + “1”; // 11
• var a = “a1” + 1; // a11
What can we do with it?

- Can be executed server-side
  - V8 JavaScript engine + node.js
- Usually executed client-side
  - Retrieve information about the document
  - Manipulate the document once this information is retrieved
Document Object Model (DOM)
Document Object Model (DOM)

- Standard way of referring to an XML and HTML documents
- Returns to the notion of XML/HTML as a tree of nodes
  - Sound familiar?
- Languages like JavaScript, implement the DOM to refer to the document root and its children as objects
- Once accessed, possible to manipulate the document structure
<aside> DOM Support

- Proprietary at first
- W3C standardized
- Modern browsers support at least DOM Level 2
document Object

• Type HTMLDocument
• Always refers to the document root
• Properties
  • offsetWidth/offsetHeight – dimensions of document
• Methods
  • getElementById(id) – return an Element object matching the given id
  • createElement(tagName) – create an Element instance, which may be later added to the document tree
window Object

- Type Window
- Refers to the actual browser window
- Properties
  - screen – provides information about the user's display
- Methods
  - resizeTo(x, y) – resizes the window
DOM Element Objects

- Actual HTML/XML elements

- Properties
  - childNodes – array of element's children, including text nodes if any
  - attributes – array of attribute-value pairs
  - innerHTML – inner content of an element, read/write

- Methods
  - appendChild(node) – Inserts a node at the end of the current node
DOM Text Nodes

- Text content is treated as a node in the tree
- Typically seen as a child of another Element object
CSS Properties

● All CSS properties can be manipulated through the DOM

● Use the style property
  ● CSS properties are camelCase

● `document.body.style.fontFamily = "Verdana";`
DOM Events

- JavaScript can be triggered in response to events
- Done by implementing *event listeners*, called when events fire
- Event triggers
  - User-driven (mouse/keyboard events)
  - Document/Window (load, resize, scroll...)
  - Form submissions
Event Handling

• Find the element object
• Register the event handler
  • element.onclick = foo;
  • Many other triggers exist...
• Write the event handler
  • function foo(e) {
    if (!e) var e = window.event; // Required for IE
    /* Do stuff */
  }
  • Could be an anonymous function!
<aside> Inline Event Handling

- Some may have seen something like `<body onload="foo()">`
- Run into same maintenance issues as with CSS!
- Event handling within the script itself is always preferred
Event Objects

- Whenever an event fires, information about the triggering event is passed to the event handler via an Event object.
- Properties of this object can be checked for more information if needed:
  - `type` – Type of triggering event (mouseover, click, etc)
  - `timeStamp` – When the event occurred
- Not all are compatible cross-browser!
Event Propagation

- If an element and its child share an event handler, which fires first?
- Used to depend heavily on the browser...
  - Netscape: “The parent!” (Event Capturing)
  - Microsoft: “No, the child!” (Event Bubbling)
  - W3C: “...Fine. You're both right!”
W3C Event Propagation

- *Event Capturing Phase* happens first
  - Resolve in parent->child order
  - Events are *captured* as they descend the DOM
- Event capturing ends when the target node is reached
- *Event Bubbling Phase* happens next
  - Resolve in child->parent order
  - Events *bubble up* through the DOM
W3C Event Propagation

- The developer can choose whether their event fires during event capturing or event bubbling
- By default, `element.onclick = foo;` syntax uses *event bubbling*