



Javascript

MIS 4530

Dr. Garrett

JavaScript – the *J* in Ajax

- JavaScript is an interpreted programming language that is predominantly used to add dynamic behavior to Web pages. All modern Web browsers include an interpreter that is capable of executing JavaScript code.
- Ecma International standardized the language as ECMA-262, which is a vendor-neutral language specification called *ECMAScript*. Mozilla's JavaScript and Microsoft's JScript (mostly) now conform to ECMA-262 Edition 3. Despite the name of the standard the name *JavaScript* is commonly used.
- JavaScript is a full-featured language that is loosely-typed and includes object-oriented capabilities. JavaScript has support for different datatypes, various statements including if/else, switch, and different kinds of loops, notation for defining functions that will accept arguments and return a value, exception-handling, notation for defining and creating objects, and support for inheritance.
- In JavaScript you do not specify a type when declaring variables, objects map property names to arbitrary property values, and inheritance is prototype-based – there is no formal notion of a *class* in JavaScript

Identifiers and Comments

- Identifiers are used to name variables, functions, and labels. The first character of an identifier must be a letter, underscore (`_`), or dollar sign (`$`). Subsequent characters can be a letter, a digit, an underscore, or a dollar sign. You can't use any of the JavaScript reserved words as identifiers.
- Single-line comments begin with two forward slashes (`//`). Multi-line comments are opened with a forward-slash followed by an asterisk (`/*`) and closed with an asterisk followed by a forward-slash (`*/`).

```
// This comment is restricted to one.
```

```
/* This comment can span  
multiple lines. */
```

```
/* This comment ends here */ causing an error here */
```

Datatypes and Values

- JavaScript supports the datatypes *Null*, *Undefined*, *Number*, *Boolean*, *String*, and *Object*. They can be divided into two categories, primitive types and reference types. Object is the only reference type, the rest are primitive types. The primitive types Number, Boolean, and String have associated global wrapper objects.
- Because JavaScript is loosely typed, any datatype can be used in place of any other datatype and JavaScript will automatically perform a conversion.

- *Null* – Represents a null, empty, or non-existent reference.
- *Undefined* – Returned when you use a variable that has been declared but never assigned a value or you use an object property that does not exist. Values `null` and `undefined` are not identical because they are two distinct values, however they do evaluate to equal because they both indicate an absence of value.
- *Number* – The set of values defined by the double-precision 64-bit format of the IEEE 754 standard, and the special NaN ("Not-a-Number") and `Infinity` (positive and negative) values. This single datatype encompasses both integer and floating-point values. Supports hexadecimal (`0x1F`), octal (`037`), and exponential notation (`5.43e12` or `5.43E12`).

- *Boolean* – Has exactly two values `true` and `false`, and can be converted to and from other datatypes.
- *String* – The datatype for all string values. String values can be defined using single quotes (`'hello'`) or double quotes (`"hello"`). JavaScript supports escape syntax (`'that\'s cool'`), and Unicode (e.g., `\uXXXX`). Strings are not of type *Object* yet you can use object notation. For example:

```
"hello".length
```

```
"hello".charAt(0)
```

Strings can be concatenated using the `+` operator. For example:

```
"Hello " + 5 + " times."
```

Strings in JavaScript are *immutable* and compared by value.

- *Object* – Unordered collections of values, called properties, where each property may contain a primitive value, object, or function. Everything in a JavaScript object is a property. When a function is assigned to a property of an object it is called a method. To create an object you either use a *constructor function* or literal notation.

```
o = new Object(); // Creates an empty object.
```

```
o = {}; // Also creates an empty object.
```

Object literal notation is simply a set of comma-separated name/value pairs inside curly braces.

```
myObj = {  
  name: "Richard Allen",  
  address: { city: "Atlanta", state: "GA" },  
  getName: function() { return this.name; }  
}
```

You can *dynamically* add and delete properties of an object.

Primitive versus Reference Types: JavaScript primitive types are manipulated by value and reference types are manipulated by reference. “By value” means that the value is copied when passed as a function argument or assigned as a variable. “By reference” means that a new reference is made to the value instead of copying the value. The only reference type in JavaScript is Object, which means that all objects including built-in objects, user defined objects, and special types of objects like arrays and functions are manipulated by reference.

Primitive Datatype Wrapper Objects: JavaScript defines objects for the primitive types Boolean, Number, and String that wrap the primitive value and provide properties and methods to manipulate the value. They are automatically created for you by JavaScript when necessary – when you use a primitive value in an Object context. For example, when you have code like `"12, 51, 31".split(", ")`, JavaScript creates an instance of the String object to handle the `split()` method call and then discards the object when done. You use the `new` operator to create an explicit instance of a wrapper object.

```
b = new Boolean(false); // Construct a Boolean object.
if (b) { // Object references evaluate to true.
    ...
}
```

Built-In Objects and Host Objects: The JavaScript interpreter creates a unique global object before executing any JavaScript code. Creating a top-level variable in JavaScript creates a property of the global object. The global object contains several properties including built-in objects and host objects supplied by the host environment. The host environment for client-side JavaScript is the Web browser. ECMAScript standardizes several built-in objects, which are collectively called the JavaScript core – the wrapper objects (Number, Boolean, and String), the Object object, Array, Date, and Math objects, a regular expression object (RegExp), a Function object, an Error object (and several derivatives), and many built-in functions like `parseInt()` and `parseFloat()`. In top-level code you can refer to the global object using the `this` keyword, or the global `window` or `self` properties.

Variables: Use the `var` keyword to associate a name with a value. Datatypes are not specified. The initial value, if not specified, is `undefined`. Variables declared outside a function using the `var` keyword become properties of the global object. If the `var` keyword is not used the variable is created as a property of the global object. Notice in the examples that variables can be reassigned any datatype.

```
var myVar = 0;           // Declares myVar and assigns it a Number
myVar = "hello";       // myVar is assigned a String
var myVar = true;      // myVar is assigned a Boolean
myVar = { index: 0, count: 2 }; // myVar is assigned an Object
```

Statements: Statements in JavaScript can optionally end in a semicolon or a line break. Eliminating semicolons is considered sloppy programming and can lead to bugs.

```
a = 1
b = 2;
```

Scope: Variables in JavaScript are only visible from within the object in which they were declared. Variables with global scope (called global variables) are variables that were declared as properties of the global object (either implicitly or explicitly) and are visible anywhere in your JavaScript program. Variables with local scope (called local variables) are declared within a function or as properties of some object other than the global object and are only visible within that object. If you declare a local variable in a function with the same name as an existing global variable, the local variable takes precedence over the global variable. You should *always* use the `var` keyword to declare variables. JavaScript does not have block-level scope, so all variables declared anywhere in a function are defined throughout the function, even before the function is executed.

```
var scope = "global"; // Declares global variable.
function showScope() {
    var scope = "local"; // Declares local variable.
    alert("Scope is " + scope);
}
showScope(); // Displays "Scope is local".

function showScope() {
    scope = "local"; // Declares global variable!
    alert("Scope is " + scope);
}
scope = "global";
alert("Scope is " + scope); // Displays "Scope is global"
showScope(); // Also displays "Scope is global"!

// Define an object with a local variable.
var myObject = { scope: "local to object" };
// To access the object's variable you must qualify it.
alert(myObject.scope); // Displays "local to object"
```

Functions: Defines a block of code that can be invoked any number of times; are first class objects that can be assigned to variables and passed around; can optionally specify any number of parameters and return a single value. The following example essentially defines the global variable `sayHello` and assigns it a function object.

```
function sayHello() {  
    return "hello";  
}
```

Or define an anonymous function using a function literal:

```
var sayHello = function sayHello() {  
    return "hello";  
}
```

Optional and variable-length arguments: Functions can be passed any number of parameters even if not declared in the function definition. Parameters declared but not passed are assigned the value `undefined`. Functions have a built-in `arguments` property that references an array-like **Arguments object**.

```
function add() {  
    var result = 0;  
    for (var i = 0; i < arguments.length; i++) {  
        result += arguments[i];  
    }  
    return result;  
}
```

The `arguments` object also provides the property `callee` that refers to the function currently being executed.

```
function(x) { return x * arguments.callee(x - 1); }
```

Lexical Scoping and Closures: Functions use the scope in which they are defined, not the scope in which they are executed – *lexical scoping*. When a function is defined the current scope chain is saved and becomes part of the internal state of the function. The scope chain includes, in order, the function's Arguments object, parameters, local variables, and then any containing objects from inside out in order of containment. The last item in the scope chain is the global object. JavaScript functions are *closures* – the combination of the code to be executed and the scope to be executed in.

```
function createFunction(x) {  
    return function() { return x; };  
}  
var fiveFunction = createFunction(5);  
var tenFunction = createFunction(10);  
alert(fiveFunction()); // Displays 5  
alert(tenFunction()); // Displays 10
```

apply () and call (): When a function is a property of an object it is called a method. All JavaScript functions have built-in methods, `apply ()` and `call ()`, that allow a function to be invoked as if it were a method of another object, in other words, they change a function's scope chain. The method parameters are: the object on which the function should be invoked – becomes the value of `this`; and the parameters that should be passed to the function – `call ()` accepts a variable list of parameters, while `apply ()` accepts an array.

```
var Richard = { firstName: "Richard" };
var Kelly = { firstName: "Kelly" };
var displayName = function(lastName) {
    alert(this.firstName + " " + lastName);
};
displayName.call(Richard, "Allen"); // "Richard Allen"
displayName.call(Kelly, "Allen"); // "Kelly Allen"
```

Arrays: A special form of object that provides the ability to reference its properties using a zero based nonnegative integer index and a special `length` property. The `length` property holds the current length of the array and can be set to a different value to increase or shorten the length of the array. Arrays can contain any combination of datatypes, and dynamically grow to hold new elements that are added to them. Can use the `Array()` constructor:

```
var myArray = new Array(50); // Length of 50
myArray.length = 0; // Length is now 0
myArray[0] = 0; // Dynamically grows
myArray[1] = 1;
myArray[3] = 3; // Length is now 4 with 1 undefined
```

Otherwise, the array literal notation is much more concise:

```
var emptyArray = []; // Create empty array
var multitypeArray = [true, 5, "hi", { x:1 }, [1, 2]];
var sparseArray = [1,,,5]; // Length of 5 with 3 undef
```

Regular Expressions: Can be created using the `RegExp()` constructor or a regular expression literal notation. The following examples create and use a regular expression object that matches the word "ajax", case insensitive.

```
var regExp1 = /\bajax\b/i;
var regExp2 = new RegExp("\\bajax\\b", "i");

// Displays "com.hello.common"
alert("com.ajax.common".replace(regExp1, "hello"));

// Displays "com.blah.common"
alert("com.AJAX.common".replace(regExp1, "blah"));

// Displays "com.hello.common"
alert("com.ajax.common".replace(regExp2, "hello"));

// Displays "com.blah.common"
alert("com.AJAX.common".replace(regExp2, "blah"));
```

Error Handling and Error Objects: Error objects are thrown by the JavaScript interpreter when a runtime error occurs – examples include `Error`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. Error objects contain a `message` property. You can throw exceptions by using the `throw` keyword followed by an expression that evaluates to any type.

```
throw new Error("My error message");  
throw "An error message";
```

You can define an exception handler using the `try/catch/finally` statement.

```
try {  
    return (pts.y2 - pts.y1) / (pts.x2 - pts.x1);  
} catch (e) {  
    alert("Error: " + e.message);  
}
```

Constructor Functions and Simulated Classes

- JavaScript doesn't yet support true classes, however, you can simulate classes using *constructor functions* and *prototype* objects.
- A constructor function looks like any other function in JavaScript except it typically adds properties to `this` and has no return statement. Example:

```
function Person(id, firstName, lastName) {  
    this.id = id;  
    this.firstName = firstName;  
    this.lastName = lastName;  
}
```

- In JavaScript, `this` always refers to the “owner” of the function being executed. When used in a top-level function, `this` refers to the global object.

- To create an instance of a class in JavaScript you must use the `new` operator. The `new` operator creates a new object with no properties and then invokes the specified function, passing the new object as the value of the `this` keyword. The constructor function can then use the `this` keyword to initialize the new object.

```
var john = new Person(1, "John", "Doe");
```

- If the constructor function returns a value, that value becomes the result of the `new` operator and the newly created object is discarded.
- The `new` operator also sets the *prototype* of the object, which is the value of the `prototype` property of the constructor function. Every function in JavaScript automatically gets a `prototype` property when the function is defined.

- The `prototype` property is initially assigned an object with a single `constructor` property that refers to the associated constructor function. Every property of the `prototype` object can be accessed as if it is a property of the object created by the `new` operator. Example:

```
Person.prototype.compareByLastName = function(other) {
    if (!other || !(other instanceof Person)) {
        throw new Error(
            "Invalid argument passed to compareByLastName(): "
            + other);
    }
    if (this.lastName > other.lastName) {
        return 1;
    }
    if (other.lastName == this.lastName) {
        return 0;
    }
    return -1;
}
var john = new Person(1, "John", "Doe");
var jane = new Person(2, "Jane", "Doe");
alert(john.compareByLastName(jane)); // Displays 0.
```

Instance Properties and Methods: In JavaScript you can say that the properties assigned to `this` in a constructor function are *instance properties* of the class defined by the constructor function. These properties are assigned to the object created by the `new` operator, which you can think of as the *instance* of the class. The “class” is defined by the constructor function (the term “class” is used loosely here).

You can define *instance methods* in JavaScript by assigning a method to the `prototype` property of a constructor function (see `compareByLastName()` in previous slide). However, in JavaScript, functions are objects, so a function can also be assigned to an instance property in a constructor function, thereby creating an instance method of the class defined by the constructor function (see next slide).

```
function Person(id, firstName, lastName) {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
    this.compareByLastName = function(other) {
        if (!other || !(other instanceof Person)) {
            throw new Error(
                "Invalid argument passed to compareByLastName():"
                + other);
        }
        if (this.lastName > other.lastName) {
            return 1;
        }
        if (other.lastName == this.lastName) {
            return 0;
        }
        return -1;
    }
}
var john = new Person(1, "John", "Doe");
var jane = new Person(2, "Jane", "Doe");
alert(john.compareByLastName(jane)); // Displays 0.
```

Class Properties and Methods: To create class properties and methods in JavaScript you assign properties directly to the constructor function (again, “class” is used loosely here).

```
function PwdUtils() {  
  
    PwdUtils.SPECIAL_CHARS = "~`!@#$%^&*()_ -+={}[]|\\:\\"; '<>, .?/";  
  
    PwdUtils.validatePassword = function(pwd) {  
        for (var i = 0; i < PwdUtils.SPECIAL_CHARS.length; i++) {  
            if (pwd.indexOf(PwdUtils.SPECIAL_CHARS.charAt(i)) > -1)  
            {  
                return;  
            }  
        }  
        alert("Passwords must contain one character from "  
            + PwdUtils.SPECIAL_CHARS);  
    };  
  
    PwdUtils.validatePassword("hello"); // Displays error message.
```

Private Members: JavaScript does not have visibility modifiers that allow you to make members of a class private. But you can achieve private members with a technique using closures.

```
function Person(firstName, lastName) {  
    // Private instance members.  
    var firstName = firstName;  
    var lastName = lastName;  
  
    // Priviledged methods that have access to private instance members.  
    this.getFirstName = function() { return firstName; }  
    this.setFirstName = function(name) { firstName = name; }  
    this.getLastName = function() { return lastName; }  
    this.setLastName = function(name) { lastName = name; }  
}
```

```
var john = new Person("John", "Doe");  
alert(john.firstName); // Displays undefined.  
alert(john.getFirstName()); // Displays John.  
alert(john.getLastName()); // Displays Doe.  
john.setFirstName("Jane");  
alert(john.getFirstName()); // Displays Jane.
```

- **Inheritance:** The `constructor` property of the prototype object references the constructor function with which the prototype is associated. Objects created from a constructor function using the `new` operator *inherit* all properties from the prototype object of that constructor function – even if the properties were added *after* the instance was created.

When a property of an object is referenced, JavaScript checks to see if that property is a member of the object. If not, then the prototype object is checked for the property. Because the prototype is an object, it too can have an associated prototype, forming a chain of prototypes, which is the inheritance hierarchy. Each prototype in the chain is checked. If the property is not found, `undefined` is returned.

```
function User(id, firstName, lastName, userName) {
    // Invoke the Person constructor function, but use the
    // new object assigned to User's this keyword. Hence,
    // the use of the call() function. This is called
    // constructor chaining.
    Person.call(this, id, firstName, lastName);

    // Initialize User properties.
    this.userName = userName;
}

// We must assign the User prototype an instance of Person
// to subclass Person. Otherwise we will subclass Object.
User.prototype = new Person();

// We must reassign the constructor property to the User
// function otherwise the constructor for User objects
// will be the Person function.
User.prototype.constructor = User;

var john = new User(1, "John", "Doe", "john.doe");
alert(john instanceof Person); // Displays true.
alert(john instanceof User);   // Displays true.
alert(john.firstName); // Displays John.
```