CS520 – JavaScript the Programming Language

Pallavi Chilappagari Natasha Gaitonde
{pallavic, gaitonde}@email.arizona.edu

Department of Computer Science
The University of Arizona, Tucson

JavaScript is one of the most popular scripting languages used today. Anyone who has browsed the web has seen it run behind the scenes, since it is embedded in most HTML web pages that perform processing depending on human interaction. There is an umbrella of technologies called DHTML (Dynamic HTML) and JavaScript is at the core of DHTML. It's most popular use is to allow executable content to be included in HTML pages. Our intention is to expose the lesser knows features of this fully fledged programming language and to show that it can be used for a lot more than just scripting.

WHAT'S INA NAME?

Developed at Netscape by Brendan Eich 1995, under the name Mocha, it was renamed to JavaScript in an attempt to ride the Java wave that was making ripples across the world at the time. Since then, it has been standardized by the European Computer Manufacturers Association (ECMA). The standards define ECMAScript, which was based on JavaScript 1.1. Netscape's JavaScript and Microsoft's JScript are the most popular flavors of JavaScript. Right now JavaScript 1.8 is the current version developed and maintained by Mozilla.

Although JavaScript is derived from a lesser known language called Self, syntactically it resembles C, C++ and Java and this makes it easy to learn. What makes JavaScript different from what we consider as mainstream programming languages is that it is dynamically typed. As a consequence, the type of a variable depends on the value assigned to it.

MAIN FEATURES

1. JavaScript is a lightweight, interpreted programming language that runs in a JavaScript Engine
2. It is weakly and dynamically typed. The typeof operator can find the dynamic type of a variable
3. It supports Object Oriented Programming using Prototype Inheritance
4. It has several basic types: Number, String, null, undefined, Boolean, etc. Besides these, everything else is an object
5. All objects are subclasses of a super Object class, just like in Java. Hence even Functions are Objects, which is the true beauty of this language
6. JavaScript has Closures using Inner Functions
7. It closely integrates with the Document Object Model (DOM) in HTML pages

SCOPE RULES

Unlike C and C++, JavaScript does not have Block Level Scoping. The keyword var must be used to declare a local variable, else the global one is referred. A variable declared in a function will be visible throughout the function.

```
var x = 10;                 //Global x

function print_me(){

       alert(x);            //Displays undefined, since there is a local x and it
                            //hasn't been initialized yet

       var x = "Hello";     //Local x, will hide global x in print_me() function

       alert(x);    }       //Displays Hello, the local x

alert(x);                   //Displays 10, the global x
```

In JavaScript the let construct allows local scoping.

```
a=6;      b=9;
let (a = a + 20, b = 77)          //local a=26, local b=77
{
             print(a+b + "\n");} //Displays 103, sum of local a and b
print(a+b + "\n");               //Displays 15
```

FUNCTIONS

- **Higher Order Functions**: In JavaScript Functions are first class data values. Functions can be passed as arguments, returned from functions, assigned to variables, objects, etc.
- **Function Pointers**: Since functions are objects, they can be assigned to variables. After the assignment, these variable act as pointers to the function.
- **Function Declaration**:

```
//Function Constructor
var square = new Function ("x","return x*x;");
//Lambda function
var cube = function(x) { return x*x*x;};
```

- **Function Arguments**: The arguments passed to a function are in an Array called arguments. This array has a callee property, which is the arguments array of the calling function. Hence this allows us to debug a function call stack that is arbitrarily long.
- **Methods and Constructors**: Since Functions are objects, they make Object Oriented programming possible in JavaScript. When used as Methods of a class type, there is an in-built reference to the "this" object, which is the invoking object.
- **Parameter Passing:** All objects are passed by reference and primitive types by value.
- Functions can return multiple values. E.g. `return ([a,b]);`

OBJECTS IN JAVASCRIPT

Objects are like associative arrays where members are key-value pairs. They are always passed by reference to functions. Some important points about Objects:

1. **new:** To create a new instance of an object the keyword new on a Function object is needed
2. Special Objects in JavaScript:
    - **Date:** Handling user inputs for date fields is really easy, since Date is a data type.

- **Null:** null is a value for an object. If something has not been declared or not yet assigned a value, it is not null but undefined. Hence error checking is important for uninitialized values, since they are handled gracefully with an undefined value and no error. Null can be used to indicate a special object value, like the list element of a linked list.

- **Array:** Arrays are Untyped and can act as containers of values. Arrays have the length property, and a useful sort function. They are indexed by zero-based integers and can be automatically extended in size.

- **RegExp:** Makes it easy to perform pattern matching using a PERL like approach.

3. **Constructor functions:** Custom or user defined objects are created using constructor functions. Members that belong to the constructor function, belongs to the class.

4. Every object has a Prototype member object, and inherits all the members of its prototype. It is useful to store instance variables and methods in the prototype of a constructor object. This is how inheritance is achieved in JavaScript.

5. **Runtime addition of members:** Like Ruby, members of JavaScript Functions can be added after declaration.

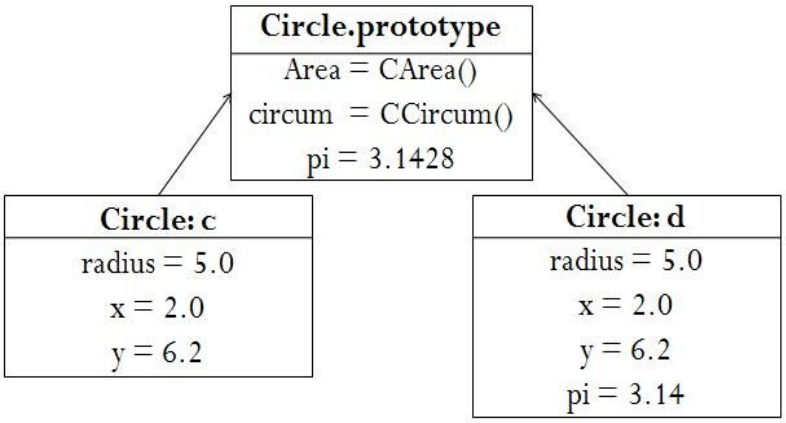
PROTOTYPE INHERITANCE

In JavaScript, Inheritance is achieved by using the Prototype property of a custom object. The Prototype acts as a Blueprint for all objects of a class and is a good place for values and functions that will be common to all objects of the Constructor class.

- **Encapsulation:** When a property value is read, first the Constructor object is checked, if the member is not found, the Prototype for that Constructor is consulted. In this way Constructor members hide the Prototype members.

- When we want to update a member value, JavaScript will not update the Constructor prototype. This is because the write will affect all the objects whose prototype is set to this object and this is not what we want.

Eg: The d Circle object hs it's own value of pi and object c will continue to see it's Prototype's value of 3.1428.



ITERATORS

JavaScript 1.7 introduced Generators and Iterators as a nice way of accessing every element of a data Collection. This is not unique to JavaScript, but as part of a programming language, it's a handy tool.

```
function fib() {
  var i = 0, j = 1;              //Local variables
  while (true) {
         yield i;                //Will return value of i to calling
         var t = i;
         i = j;
         j += t; }}

var g = fib();
for (var i = 0; i < 10; i++) {
  document.write(g.next() + "<br>\n");} //successive calls to g.next()
                                        //will iterate all values of i
```

EXCEPTION HANDLING

JavaScript 1.5 has standardized Exception handling along with try, catch and finally blocks. There are some inbuilt Exceptions, in addition we can create custom Exception objects and handle them. Finally is a great place for clean up code, if an exception has been raised. This is similar to Java's finally clause. Also try..catch blocks can be nested.

ARRAY COMPREHENSIONS

Similar to what we say in Python, JavaScript 1.7 has introduced a neat way to initialize arrays on declaration. Initialization can be done conditionally as follows:

```
function generate_sequence (start, end){
  for (let i = start; i < end; ++i)
  {
          yield i;
  }}

//initialize every even index from 0-200
var evens = [i for each
                  (i in generate_ sequence (0, 200))
                          if (i % 2 == 0)];
```

CLOSURES

JavaScript allows inner functions and these functions have access to all the local variables, parameters and other inner functions of the function in which they were created. This leads us to the concept of closures. A closure is an object made up of two things: a function and the environment in which it was created.

Typically, a closure is formed when an inner function is made available outside the function in which it was created, so that it can be executed even after the outer function returns. At this point, the inner function still has access to all the parameters and local variables and of the outer function.

```
function exampleClosure(num){
        var exponent = 2;
        function power_exponent(){
              return (num*exponent);}
return exampleReturned;}
var globalVar = exampleClosure(2);
var product=globalVar();
```

The inner function 'power_exponent' computes the product of the formal parameter 'num' and the local variable ‘exponent’ of the outer function 'exampleClosure'. Since JavaScript has first-class functions, functions can be returned and stored in variables. ‘globalVar’ now has a reference to the inner function

‘power_exponent’ (the return value of exampleClosure()) and can be executed by calling globalVar(). Thus, a closure has been formed and consequently, 'product' has the value 4 (2*2) at the end of execution of the above block of code.

As the example illustrates, even after the outer function returns, the inner function ‘power_exponent’ can still access the local variable ‘exponent’ and the formal parameter ‘num’ of the outer function. This is done by binding the execution context of the inner function to the environment in which it was created. In other words, the activation record for the outer function is not de-allocated as long as there is a reference to its inner functions.

An interesting application of closures is that it can be used to emulate private methods in JavaScript. Unlike Java, JavaScript does not provide an explicit way of declaring private methods. Private methods are useful for restricting access to code and managing the global namespace more efficiently.

```
var int_counter = (function() {
  var counter = 0;
  function changeVal(val) {
          counter += val;  }
        return {
           increment: function() {
                 changeVal(1);
          },
            decrement: function() {
                 changeVal(-1);
          },
          value: function() {
             return counter;
          }  }   })();

alert(int_counter.value());

int_counter.increment();

int_counter.increment();
```

The trailing parenthesis at the end of the function definition above tells JavaScript to execute the function as soon as it has been parsed. As a result, ‘init_counter’ does not refer to the function itself but to the return value of the function. In this case, the return value is a wrapper to the three functions ‘increment’, ‘decrement’ and ‘value’. Since these are inner functions, these have access to the variable ‘counter’ and the method ‘changeVal()’. These inner functions can now be invoked through the following statements which result in closures.

```
init_counter.value()
init_counter.increment()
init_counter.decrement()
```

The function ‘value()’ returns the current value of the variable ‘counter’. increment() and decrement() increment and decrement the value of ‘counter’ by 1 respectively, by accessing the method ‘changeVal()’. Thus, the only way to access the method ‘changeVal()’ is through the inner functions returned from the anonymous function. The access to ‘changeVal()’ is thus restricted and in effect, it is a private method now.

Often closures are created accidentally and if they are written just to access inner functions without paying attention to memory management issues, they can result in memory leaks (we'll see an example in the next section).

GARBAGE COLLECTION

JavaScript is a garbage collected language and it uses the mark-and-sweep algorithm for garbage collection. Since JavaScript is an interpreted language, it always runs in a host environment (commonly the browser). The way in which the browsers handle the allocation of DOM (Document Object Model) objects can have a significant impact on memory management in JavaScript. Browsers use reference counting for garbage collection and the disadvantage of reference counting is that it cannot handle cyclic references. It is quite common in web applications to have cyclic references between JavaScript variables and DOM objects. This could lead to memory leaks in some cases as illustrated below.

```
<script type="text/javascript">
document.write("Circular references between JavaScript and DOM!");
  var ob;
  window.onload = function()
 {
      obj=document.getElementById("DivElement");
      document.getElementById("DivElement").expandoProperty=obj;
      obj.bigString=new Array(1000).join(new Array(2000).join("XXXX"));
};
```

```
</script>
<div id="DivElement">Div Element</div>
```

At the end of the execution of above code, a cyclic reference is formed between the DOM element ‘DivElement’ and the JavaScript object ‘obj’. The DOM object 'DivElement' is never reclaimed because its reference count never falls to 0 (there’s a live reference through ‘obj’). The JavaScript object ‘obj’ is not reclaimed because of a live reference to it (through the ‘DivElement’ object).

Thus, though JavaScript itself has an efficient garbage collection technique that handles cyclic references well, the incompatible garbage collection schemes used by browsers can lead to memory leaks.

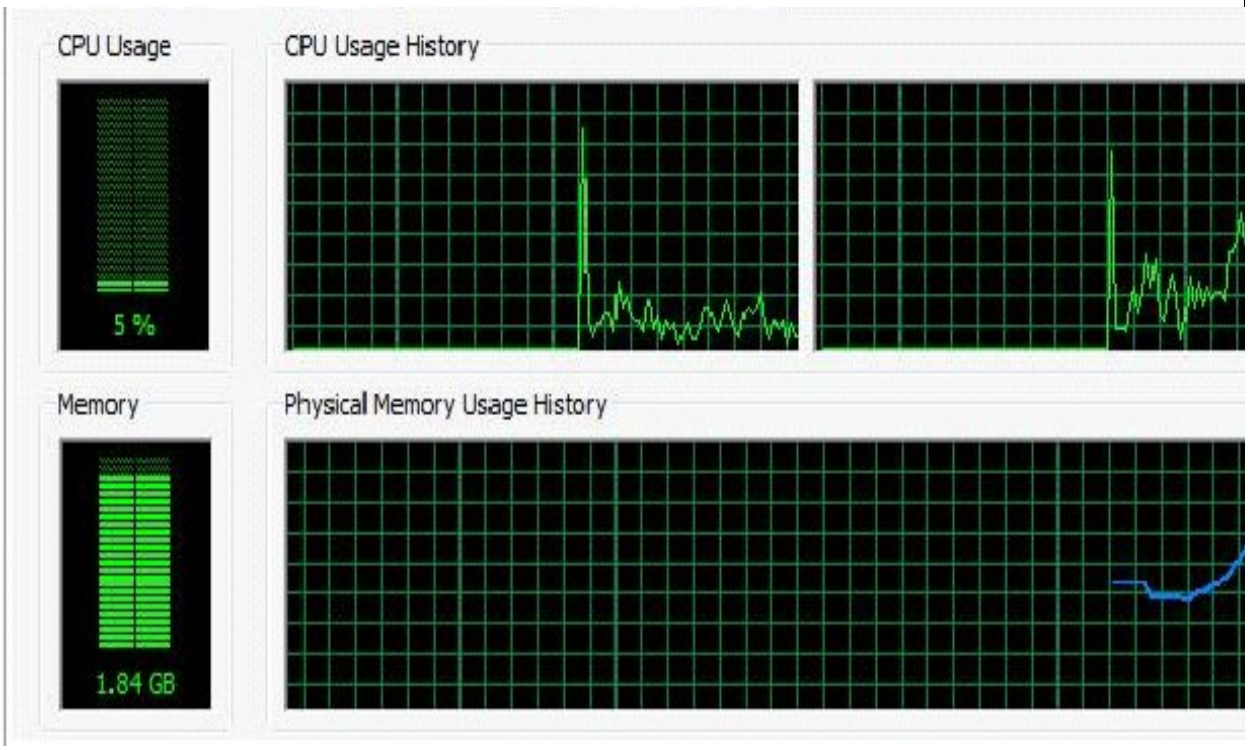
Closures are a common source of memory leaks in JavaScript. The following example demonstrates memory leak caused due to a closure.

```
<html>
<head>
<script type="text/javascript">
    function LeakMemory(){
        var parentDiv = document.createElement("div");
        parentDiv.onclick=function(){     foo();       };

        parentDiv.bigString =
          new Array(1000).join(new Array(2000).join("XXXXX"));
    }
</script>
</head>
<body>
<input type="button" value="Memory Leaking Insert"
onclick="LeakMemory()" />
</body>
</html>
```

Every time the button ‘Memory Leaking Insert’ is clicked, the function LeakMemory() is invoked. The anonymous function assigned to ‘parentDiv.onclick’ is a closure over ‘parentDiv’ and this function can access the ‘parentDiv’ object (by definition of closure). ‘parentDiv’ in turn refers to a DOM object (‘div’). Thus, a cyclic reference is formed between DOM and JS world, which results in a memory leak.

The following figure is a snapshot of the memory usage graph, when the above code is executed in a browser. It can be seen from the graph that the memory usage increases every time the button is clicked and eventually the browser hangs or crashes. This is highly undesirable given the interactive nature of web applications.



Cyclic references might not be obvious in all cases and closures are known to be good at hiding such references. Thus, it is important to write leak free code and de-allocate or nullify objects explicitly when they are no longer required to avoid cyclic references.

REFERENCES

1. http://developer.mozilla.org/en/docs
2. http://www.jibbering.com/faq/faq_notes/closures.html
3. http://www.javascriptkit.com/javatutors/closuresleak/index.shtml
4. http://www.codeproject.com/KB/scripting/leakpatterns.aspx
5. David Flanagan, *JavaScript: The Definitive Guide*, Third Edition
6. Don Gosselin, *JavaScript: Comprehensive*