Notes on JavaScript (aka ECMAScript) and the DOM

JavaScript highlights:

- Syntax and control structures superficially resemble Java/C/C++
- Dynamically typed
- Has primitive types and strings (which behave more like primitive types than objects) and reference types (objects, arrays, and functions). Numbers and arithmetic are floating-point and there is no explicit integer type.
- Runs in an interpreter with automatic memory management (garbage collection)
- Not a true OO language, but can create class-like constructs using *constructors* and *prototype objects*. Objects are really just name-value pairs (associative arrays)
- Functions (or methods) are "first-class" objects and can be stored in variables or as object properties. Functions do not have to be associated with objects, but when they do we usually call them methods
- The scope of a function (method) is a distinct object that can outlast the function invocation

JavaScript console or standalone interpreter

JavaScript is most often associated with web pages and is commonly run in a browser. However, it does not have to be. In fact there is a JavaScript interpreter now built into the JDK. If you have the JDK bin directory in your PATH, you can run the command **`jrunscript`** in a command shell to get an interactive javascript prompt. This is useful for experimentation, e.g.:

```
smkautz-macbook:javascript_examples smkautz$ jrunscript
js> var x = 42;
js> x
42.0
js> x == "42"
true
js> x === "42"
false
js> typeof(42)
number
js> typeof(42.0)
number
js> quit();
smkautz-macbook:javascript_examples smkautz$
```

You can also run scripts from the command line this way, e.g., if you had a file **count.js** containing the code

```
for(var i = 0; i < 10; ++i)
{
  print(i + "\n");
}
```

then you could run it with the jrunscript command:

```
smkautz-macbook:javascript_examples smkautz$ jrunscript count.js
0
1
2
3
4
5
6
7
8
9
smkautz-macbook:javascript_examples smkautz$
```

Client-side JavaScript

When running in a browser, there is always a "global object" of type Window. (The significance of the phrase "global object" will be more clear when we talk about scope chains later.) One important property (i.e. variable) of this object is the **`document`** property (of type Document). One easy thing to do with JavaScript is to generate html content with the method document.write(). Consider the html page **example1.html**

```
<head></head>
<body>
<script>
for(var i = 0; i < 10; ++i)
{
  document.write(i + "<br>");
}
</script>
</body>
```

A few things to notice about this simple example:

- The loop appearing between the <script> tags is JavaScript code. When put in the body like this, it is executed once when the page is loaded. Everything (e.g. values of variables) starts with a clean slate when the page is reloaded. This is not necessarily typical or particularly useful; most JavaScript defines data types and methods in the head or in an external script that is loaded in the head.

- The format of the for-loop is familiar, but note that the variable “i” is not declared with a type as you might expect. Any variable will happily store any type you put in it. You don’t actually even need the **`var`** declaration, but you should always use it: if you omit the var declaration, JS will silently create the variable for you but it will have global scope. (In this example, however, it makes no difference because the script tags always have global scope.)

- Try making a mistake, e.g., delete the parenthesis at the end of the for-loop and reload the page. It will be blank, because there is an error in the JS. In Chrome you can open up a

JS console via the View->Developer menu or the customization button, where the error will be reported. Correct the error and reload the page. You can right-click on the JS console to bring up an option to clear it. (This is an interactive console like the JDK one above, with an important difference: when you are stopped at a breakpoint, everything will be interpreted relative to the current execution context.)

- Next, open the Chrome "Developer Tools". Click the "Source" tab and click on the file example1.html. You may need to reload the page, but it should show up in the center pane. Set a breakpoint on the document.write line by clicking in the left margin and reload the page. Note you can step through the code using the step over button and see the lines appear one at a time.

Notes about Developer Tools: You can actually edit javascript and css code while executing it, but not the html files. However, if you have copies of files in a local directory it is easy to create a "workspace" and add the directory. Then you can use Developer Tools as an editor. See

http://www.html5rocks.com/en/tutorials/developertools/revolutions2013/#toc-workspaces

For more about debugging with Developer Tools see

https://developers.google.com/chrome-developer-tools/docs/javascript-debugging

Objects

You can create an object with the **`new`** operator. Objects can be given properties just by referring to them. You can use dot notation or associative-array notation, since an object is really just a collection of name-value pairs. You can also use JSON notation to set up a list of name-value pairs. E.g see **example2.html**:

```
<head>
</head>
<body>
<script>
// create an object and some properties
var obj = new Object();
obj.fred = 42;
obj.george = 137;
obj["hermione"] = "hello";

// same thing, using JSON notation
var obj2 =
{
  "fred": 42,
  "george": 137,
  "hermione": "hello"
}
// mix and match the associative array notation
```

```
document.write(obj["fred"] + "<br>");
document.write(obj.george + "<br>");
document.write(obj.hermione + "<br>");

document.write(obj2["fred"] + "<br>");
document.write(obj2.george + "<br>");
document.write(obj2.hermione + "<br>");

// we can also loop through the properties
for (var key in obj)
{
  document.write(obj[key] + "<br>");
}
</script>
</body>
```

You can define a data type by creating a constructor, which is just a function (more later).

Arrays

JS also has arrays, which are just like objects, except values are associated with numbers instead of strings. Arrays don't have a fixed predefined length, and don't have to be contiguous (you can still loop through the indices). E.g. **example 3.html**:

```
<head></head>
<body>
<script>
// create an array
var arr = new Array();
var obj = new Object();
obj.ginny = 137;

arr[0] = "This is a string";
arr[42] = obj;
document.write(arr[0] + "<br>");
document.write(arr[42].ginny + "<br>");

// this value is undefined
document.write(arr[2] + "<br>");

// you can also loop through the indices that have been assigned
for (var index in arr)
{
  document.write(arr[index] + "<br>");
}
</script>
</body>
```

null and undefined

As you can see the elements of an "array" don't have to be the same type. This example begs the question: what happens if we look at `arr[2]`? You might well ask. JS has two interesting

values, **`null`** and **`undefined`**, which are not the same. The meaning of **`null`** is similar to its meaning in Java: a special value you can assign to an object reference to specifically indicate that it does not refer to an actual object. On the other hand, **`undefined`** is the value a variable has if it has never been initialized.

Try adding the line **`document.write(arr[2]);`** to example3.html.

The operators == and ===

As in Java there is a == operator for equality, but also a different operator for a stricter version of "sameness", called the *identity* operator and denoted with a triple equals sign ===. For primitives and strings == and === have the same effect. Primitives and strings are compared by value, so unlike Java, you can use the == operator to compare strings, or even the === operator. (Weird exception: the value NaN is never equal or identical to any other value, not even itself.) Objects, arrays, and functions are compared by reference and are equal or identical only if they refer to the same object.

`null` and **`undefined`** are the same according to == but not ===.

The == operator performs type conversion when comparing values of different types. For example, 42 == "42" is true. The === operator returns false for different types.

The corresponding negations are != and !==.

Functions

Functions, or methods, in JS are "first-class" objects, meaning that a function can be stored in a variable, passed as a parameter, compared with ==, etc., just like any other values. They can be named or anonymous. In the next example we create two functions, one named f that returns a value and an anonymous function that is stored as a property of an object, and does not return a value. Note the parameters are not typed. Functions would normally be defined in the head or in an external script loaded in the head. See **example4.html**:

```
 <head>
<script>
  var i;
  function f(x, y)
  {
    return x + y;
  }
  var obj = new Object();
  obj.m = function(x)
  {
    i = x * x;
  }
</script>
</head>

<body>
<h1>Function example</h1>
```

```
<script>
  obj.m(5);
  document.write(i + "<br>");
  document.write(f(4, 3) + "<br>");
</script>
</body>
```

Constructors

You define a data type by defining a constructor for it. A constructor is just a function that initializes some properties of an object. It does not have a return statement. It is invoked using the keyword new. See **example5.html:**

```
<head>
<script>
function Point(first, second)
{
  this.x = first;
  this.y = second;
  this.dist = function()
  {
    return Math.sqrt(this.x * this.x + this.y * this.y);
  }
}

</script>
</head>
<body>
<h1>Constructor example</h1>
<script>
var p = new Point(3, 4);
document.write(p.x + "<br>");
document.write(p.y + "<br>");
document.write(p.dist());
var q = new Point(3, 4);
document.write(p.dist == q.dist);
</script>
</body>
```

Methods and pseudoclasses

Note that Point looks like the definition of a class. One of its attributes is a function, which can be invoked like a method in Java. However, the example above differs from a class specification in that there will be a different `dist()` function for each instance of point we create (which you can see from the output of the last two lines).

Each object has an internal reference to another object called the prototype object, and properties of the prototype object are "inherited" by the object, in the sense that they appear to be properties of the object. The prototype for an object is just the `prototype` property of its constructor function (which defaults to Object). So to create a class-like definition in which the methods are shared by all instances, you can make the methods properties of the prototype. See **example6.html**:

```
function Point(first, second)
{
  this.x = first;
  this.y = second;
}
Point.prototype.dist = function()
{
  return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

What happens is that when you try to invoke **`dist`** on a **`Point`** object, the interpreter will discover that **`Point`** has no property named **`dist`**, and will then look in its prototype.

It is important to note that ALL instances of Point will share the same prototype object. So the prototype is a good place to put methods, but not a good place to put attributes. There is a built-in asymmetry: when you try to *read* a property that isn't defined for an object, the interpreter will look in its prototype and use the value there if it is present. If you try to *write* a property that isn't defined for an object, the interpreter will simply define the property for the object, and ignore the value in the prototype from then on. More on this a bit later.

Variable scope and lifetime

Java is "block scoped". A local variable comes into existence at the point where it is declared and its scope extends to the end of the block in which it is declared. When execution reaches the end of the block, or when the method returns, the variable effectively ceases to exist. The way we think of this is that locals exist within an activation frame on the call stack.

<picture of call stack...>

JS is not block scoped. The scope of a variable is the entire function body in which it appears. This fact is really a consequence of the fact that local variables are not just slots in an activation frame on the call stack, but have a very different kind of existence than locals in Java. In fact, all JS variables are really *properties* of some object. Global variables are properties of the "global object" which, in the case of client-side JS code, is the Window. Local variables are properties of something called the "call object". The call object is created when a method is called. The strange thing is that the call object, and therefore local variables declared in a function, may continue to exist even after the method returns.

Functions definitions can be nested inside other functions. A nested function body can refer to any variable declared in any enclosing function as well as to global variables. Specifically, the call object for a nested function has a reference to the call object for an enclosing function, and so on up to the global object, forming a "scope chain" that is the execution context for the code in the inner function.

Now, if the nested function is only used inside of an enclosing function, there is nothing special about any of this. But functions are first class values, and references to functions can be stored in variables, or stored as properties of objects to used as methods. So a reference to a nested

function can persist even after the enclosing function returns. And as long as a reference to a function exists, *its scope chain continues to exist.*

Here is an example with a simple nested function.

```
function create(x)
{
  var ret = function(y) { return x + y; }
  return ret;
}
```

Note that the body of the nested function that is returned by create() depends on the local variable x. Now suppose we make a call such as

```
var f1 = create(1);
```

In the scope chain for f1, x has value 1, and that value persists as long as we hold the reference f1, even though x was a local variable in function create(), which has returned. See **example7.html**:

```
<body>
<script>
  var f1 = create(1);
  var f2 = create(2);
  document.write(f1(42) + "<br>");
  document.write(f2(42) + "<br>");
</script>
```

A function, together with a scope chain, is called a *closure*. Closures are frequently used in JS idioms and examples, especially when creating objects with non-public properties or dynamically creating event handlers (as in Ajax), so it is important to understand them.

Here is a similar example. The inner function returned by `mystery()` retains variable v in its closure(see **example7a.html**):

```
<script>
function mystery()
{
  var v = 42;
  return function()
  {
    v = v + 1;
    return v;
  }
}
</script>
</head>
<body>
<script>
```

```
// inner function retains local variable v in its closure
var f = mystery();
document.write(f() + "<br>");
document.write(f() + "<br>");
document.write(f() + "<br>");
document.write(f() + "<br>");
</script>
</body>
```

It is worth noticing that the call objects making up the scope chain for a given function depend only on where the function is *declared*, not on where it is *called*. JS functions are said to be "lexically scoped". By contrast, in Java, the variables that exist when a method executes are in the call stack and hence may be different depending on where the method is called.

Event handling

As noted before, JS embedded in script tags in the body is executed once when the page loads. This is ok for illustrating basic JS code but not actually very useful. For an interactive web application we need to be able to react to events generated from the browser. This is a bit like writing event handlers in Swing and registering listeners. Here is a simple example of an event handler for a button click (see **example8.html**):

```
<head>
<script>
function showEvent()
{
  alert("Button clicked!");
}
</script>
</head>
<body>
<input type="button" name="clickme" value="Click me" onclick="showEvent();"
/>
</body>
```

The string in quotes assigned to the `onclick` property is a sequence of JS statements; in this case just calling a function.

Some commonly used events to which you can attach handlers are:

- `onclick` (button-like elements and anchors)
- `onmousedown`, `onmouseup` (most document elements)
- `onmouseover`, `onmouseout` (most document elements)
- `onchange` (input, select, textarea)
- `onload` (body)

An onload event handler is executed after the page is fully loaded. An alternate mechanism for attaching event handlers, more cleanly separating JS from html, is to directly assign an event

handling function to the **`onclick`** property of the button. Since most event handlers manipulate page content, this would be done in an onload handler for the page so that it won't be triggered before the page is fully loaded. See **example9.html**:

```
<head>
<script>
function showEvent()
{
  alert("Button clicked!");
}

window.onload = function()
{
  var button = document.getElementById("thebutton");
  button.onclick = showEvent;
}
</script>
</head>
<body>
<input id="thebutton" type="button" name="clickme" value="Click me" />
</body>
```

Referencing external JavaScript files

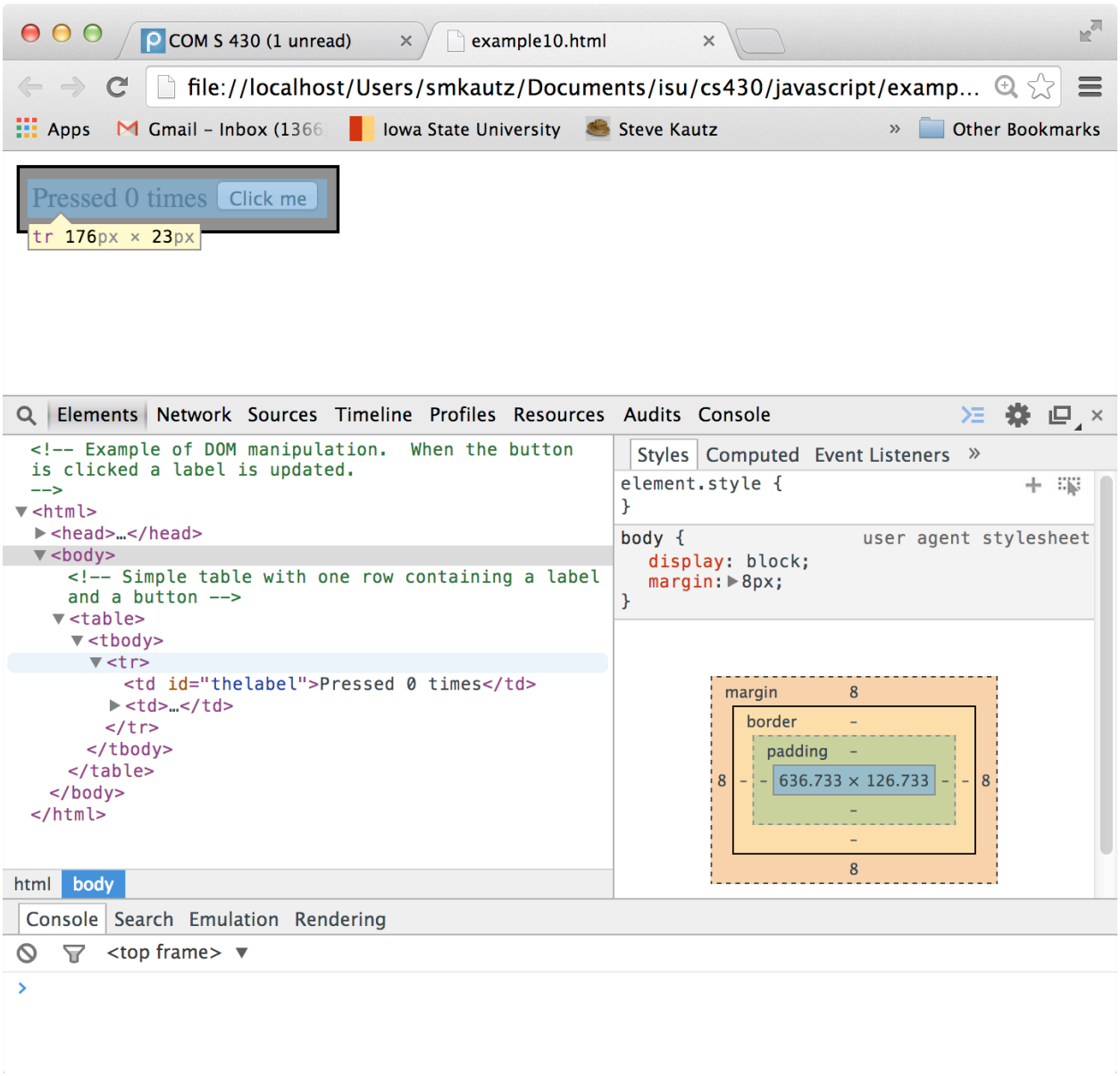
The code can be put in an external file, typically with a .js extension, and referenced in the <script> tag (see **example9a.html** and **example9_code.js**):

```
<head>
<script src="example9_code.js"></script>
</head>
<body>
<!-- note there is no javascript in the html -->
<input id="thebutton" type="button" name="clickme" value="Click me" />
</body>
```

Scripting the DOM

The presence of the annoying message box generated by the **`alert()`** function begs the question: how do you change the page in response to user events? We can't use **`document.write()`** in an event handler (which will attempt to write to a new, blank version of the page). The key is the DOM, or Document Object Model. The **`getElementById()`** method in the example above, in fact, returns a DOM element. Attaching the event handler by setting the onclick property of the returned object is a simple example of manipulating the DOM.

Idea: html describes a hierarchical (tree-like) structure, and the browser renders a page by first creating a model for this structure. The screenshot below is the view from the Elements tab in Developer Tools.



(See **`example10.html`** – the comments explain what's going on - and examine the details using Developer Tools. Examples 11, 12, and 13 further illustration DOM and css modification. Example 13 shows use of a timer for simple animation of text.)

Threading

Like the Swing framework, JS executes in a single-threaded environment. Page loading, event handlers, and timer events (and as we will soon see, Ajax callbacks) all execute in a single thread. Worker threads for background tasks can be created using WebWorker, similar to SwingWorker.

Inheritance

We saw with the `Point` example that if a property is not found in the prototype, the interpreter will look in the prototype's prototype, and on up the chain to Object. This looks a lot like like the dynamic binding mechanism in an OO language, and in fact it is through the use of prototypes that you can get something like inheritance in JavaScript. There are really two distinct aspects to inheritance.

1. Creating and initializing the properties of an object that were initially defined in the supertype.
2. Setting up the prototype chain so that methods defined in the prototypes will be found.

It should be emphasized that this JavaScript is extremely flexible and there are several ways to acheive the same results.

The first step makes sense when you think about the fact that a constructor has a slightly different role in JavaScript than in Java or C#. In Java, the attributes (instance variables) are predefined in the class definition, and all the constructor has to do is initialize them correctly. In JavaScript, every object is initially a kind of blank slate, and the properties have to be created as well as initialized.

Here is an example, see **animals.html**.

```
<html>
<head>
<script>

// A class-like definition
function Pet(n)
{
    // create and initialize a public property
    this.name = n;
}

// Public method added to prototype, same code for all instances.  By default
// the prototype is Object.
Pet.prototype.getName = function()
{
    // Properties need to be prepended with 'this'
    return this.name;
}

// Another class.  Our intention is to extend Pet, so we do two things...
function Dog(name, license)
{
  // (1) Explicitly invoke Pet constructor using call() or apply(), passing 'this'
  // as the first argument.  All properties of Pet will be created
  // for 'this' Dog object.  Note we are not using 'new' keyword, so the
  // invocation does NOT create a distinct Pet object.
  Pet.call(this, name);

  // create and initialize other properties of Dog
  this.license = "000" + license;
}
```

```
// (2) Create a Pet object to be Dog's prototype.  Whenever someone invokes a method
// or property on a Dog that isn't defined in Dog, the interpreter will look in
// the prototype.
Dog.prototype = new Pet();

// This is optional, but emphasizes that the per-instance properties of Pet
// will not ever be used as part of the prototype
delete Dog.prototype.name;

// Public method added to prototype
Dog.prototype.getLicense = function()
{
    return this.license;
}

// Public method added to prototype
Dog.prototype.speak = function()
{
    document.write("woof" + "<br>");
}

// A subtype of Dog
function Retriever(name, license)
{
    // (1) invoke Dog constructor, passing 'this'
    Dog.call(this, name, license);
}

// (2) Create a Dog for the prototype
Retriever.prototype = new Dog();

// Per-instance license property of prototype will not be used
delete Retriever.prototype.license;

// Override
Retriever.prototype.speak = function()
{
    document.write("raoou" + "<br>");
}

// Add a new method
Retriever.prototype.retrieve = function()
{
    return "Bird";
}

// Override Pet method
Retriever.prototype.getName = function()
{
    // Trying to call super.getName, have to explicitly invoke getName
    // on parent class prototype using call() or apply(), passing 'this'
    // as the first argument
    var firstName = Dog.prototype.getName.call(this);
    return firstName + " the Great";
}
</script>
</head>
<body>
<script>

// Try it out...
var d = new Dog("Ralph", "123");
```

```
var r = new Retriever("Clover", "456");

d.speak();                      // speak() method of Dog
document.write(d.getName() + "<br>");     // getName() method from Pet
document.write(d.getLicense() + "<br>");
document.write("Setting name property to Fluffy" + "<br>");
d.name = "Fluffy";              // This works b/c name is a public property
document.write("Name is now " + d.getName() + "<br>");     // Now name is Fluffy
//document.write(d.retrieve() + "\n");  // this would be a runtime error
document.write("<br>");

r.speak();
document.write(r.getName() + "<br>");
document.write(r.getLicense() + "<br>");
document.write(r.retrieve() + "<br>");
document.write("<br>");
</script>
</body>
</html>
```

Here is a variation. The **getName** method of **Pet** seems redundant, since the **name** property is actually public. It is also possible to define variables that act like private instance variables. The idea is to define a local variable within the constructor. A method defined within the constructor can refer to such variables and they continue to exist as part of its closure. This is in some sense "inefficient" because now you really do need a different copy of the method for each instance, i.e., you can't make the method a property of the prototype. Notice that the **name** property of **Pet** is no longer public. When we attempt to change the **Dog d**'s name to Fluffy, we can easily create a new name property for that object, but the **getName** method still returns the original value stored in its closure. See **animals2.html**.

```
<html>
<head>
<script>
function Pet(n)
{
    // acts like a private variable (technically could just use n)
    // that lives in the closure for the function getName
    var name = n;

    // public method, code is per-instance rather than part of prototype
    this.getName = function()
    {
        return name;
    }
}

function Dog(name, license)
{
  Pet.call(this, name);

  // acts like a private variable, initialized with a private method
  var licenseNumber = convert();

  // public method, code is per-instance rather than part of prototype
  this.getLicense = function()
  {
    return licenseNumber;
  }
```

```
  // acts like a private method
  function convert()
  {
    return "000" + license;
  }
}
Dog.prototype = new Pet();
Dog.prototype.speak = function()
{
    document.write("woof" + "<br>");
}

function Retriever(name, license)
{
    Dog.call(this, name, license);
}
Retriever.prototype = new Dog();
Retriever.prototype.speak = function()
{
    document.write("raoou" + "<br>");
}
Retriever.prototype.retrieve = function()
{
    return "Bird";
}
Retriever.prototype.getName = function()
{
    var firstName = Dog.prototype.getName.call(this);
    return firstName + " the Great";
}
</script>
</head>

<body>

<script>
var d = new Dog("Ralph", "123");
var r = new Retriever("Clover", "456");

d.speak();
document.write(d.getName() + "<br>");
document.write(d.getLicense() + "<br>");
//document.write(d.retrieve()  + "<br>");  // runtime error
document.write("Setting name property to Fluffy"  + "<br>");
d.name = "Fluffy";
document.write("Now name is " + d.getName() + "<br>");
document.write("<br>");


r.speak();
document.write(r.getName()  + "<br>");
document.write(r.getLicense()  + "<br>");
document.write(r.retrieve() + "<br>");
document.write("<br>");
</script>

</body>
</html>
```