Javascript : the language

accu2009 conference
25th April 2009
Tony Barrett-Powell

Introduction

- Exploring aspects of the language
- Considering styles
  - Procedural
  - Object-oriented
  - Functional
- Some bad parts of the language along the way
- Some of the tools available

Who am I?

- Programmer at Oracle
  - Business Intelligence Tools
  - C++, Java and Javascript
- ACCU
  - Committee member
  - Web site editor

Motivation

- A lot of moaning about Javascript
  - From a strong typing point of view
- Very widely used language
- Suspicion that it is being used from wrong point of view
- A way to solve the problem of code that is repeated in applications with web front-end

A potted history

- Developed at Netscape
  - To provide what java didn't
  - Called mocha, livescript then javascript
- Copied by Microsoft as jscript
- Standardised in an attempt to regain control
  - ECMAscript
  - Errors in jscript kept in standard
- New ambitious standard abandoned for something less so – now in final draft

Basics

- Dynamically typed
- Objects are containers
- Functions are first class entities (lambda)
- Prototype based
- Simple linkage
- Interpreted
- Thread neutral
- Garbage collected

Types

- Number
  - floats no integer type
- Strings
  - immutable, not UTF-16
- Boolean
- Objects
- null and undefined

Procedural javascript

- How much javascript is written
- Example:

```
function add (x, y) {
  return x + y;
}

var a = add(1,2);          // a = 3
var b = add('1','2');      // b = '12'
var c = add('3',3);        // c = '33'
```

Scope

- Limited number of scopes in javascript
- {} is not a scope
- Example:

```
Function foo () {
   var i = 0;
   {
     var i = 1;
   }
   return i;
}
```

objects

- Javascript is a prototype based language
  - (self, Javascript and a few others)
  - There are no classes
  - Objects are dynamic
    (they can be modified at runtime)
  - Containers of name/value pairs
  - Objects are created based on other objects
  - Javascript has pseudo-classical features

Object creation

- Examples:
  ```
  var foo = {};
  ```
- And the equivalent
  ```
  var foo = new Object();
  ```
- These are created based on the Object object
  - Provides minimal methods
    - toString() / toLocaleString()
    - ValueOf()
    - hasOwnProperty() / IsPropertyOf() / propertyIsEnumerable()

Object augmentation

- Example:
```
var foo = {};
```
- Adding a member:
```
foo.x = 0;
foo['y'] = 1;
```
- Adding a method:
```
foo.add = function (x, y) {
  return x + y;
};
```

Object literal

- Instead of augmenting the object it can be defined as a literal:

```
var foo =
   { x   : 0,
     add : function (x, y) {
             return x + y;
           }
   };
```

- This is the basis of JSON

Object diminution

- What has been added can be taken away:

```
delete foo.x;
delete foo['y'];
delete foo.add;
```

pseudo-classical

- The designers of javascript seemed to have been uncomfortable with the prototype based approach
- They added features that look like class based languages:
  - new keyword
  - Constructor functions
  - instanceof operator
  - Prototype member of user defined functions
- Not really one or the other

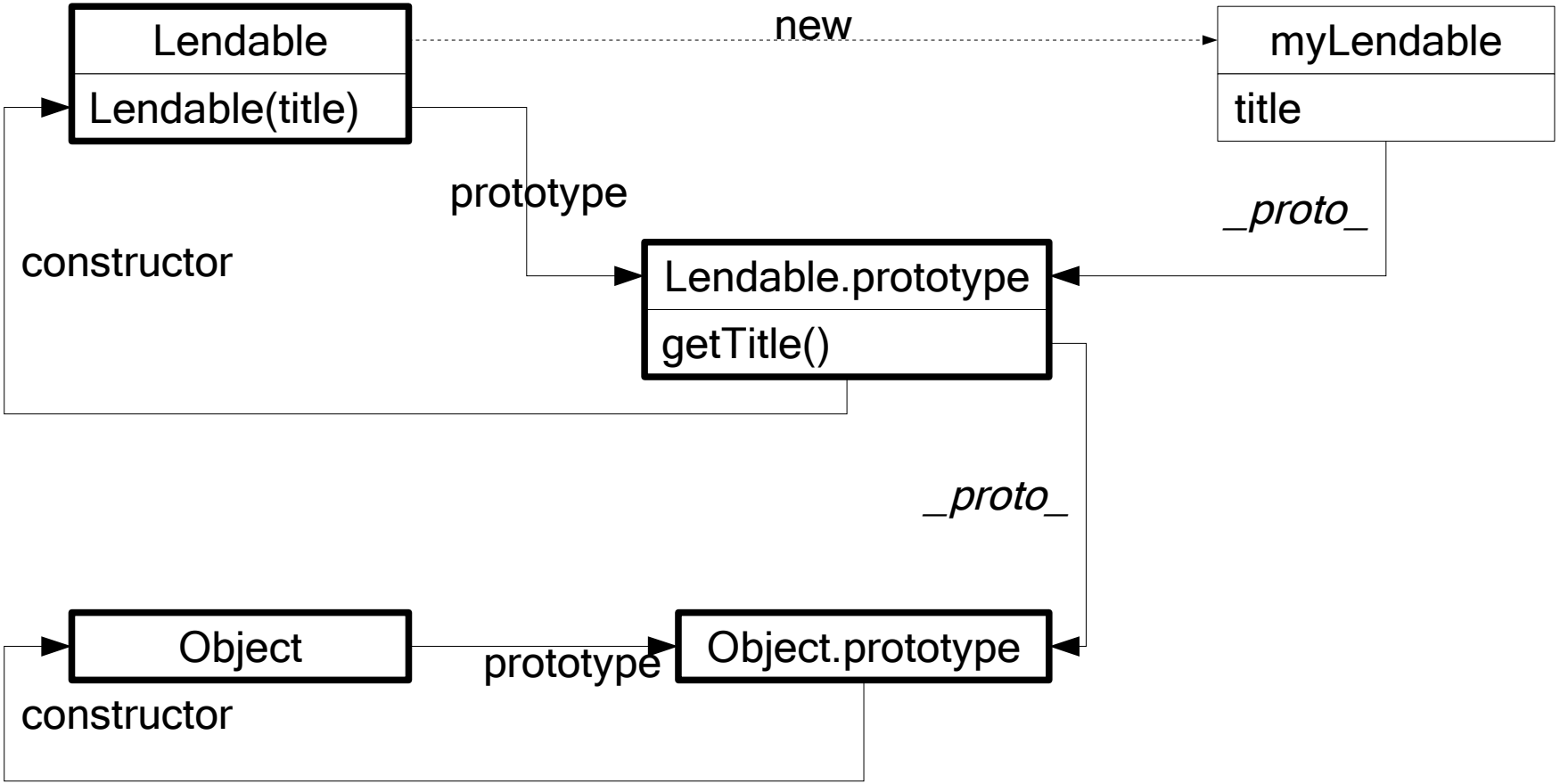
A javascript 'class'

- Example:

```
function Lendable(title) {
  this.title = title;
  return this; // optional
}
Lendable.prototype.getTitle() {
  return this.title;
};

var myLendable = new
Lendable('Javascript - The Book');
```

Looking under the covers



Issues

- Calling a constructor without new
  - this is the global object
  - The global object is returned
  - No error
  - Tools combined with coding style can help
- The object is unlike one produced in a classical language
  - Data is public

Name lookup

- When a method is invoked on myLendable
  - It is looked up in the Lendable object
  - It is then looked up in the Lendable.prototype object
  - And finally in Object.prototype object
- This is called the prototype chain
- A method or member can be added to any of these objects at runtime
- Name lookup occurs at the point of invocation

Mutating state

- A property can exist in the prototype object
  - These are read only from the instance
  - Mutation of these properties creates the property in the instance
  - Deletion of the property in the instance exposes the property in the prototype

Inheritance

- Given we have classes (of sorts) we should be able to support inheritance
  - Questionable in a dynamic language
    - If it does - it is
  - Only needed for code re-use
    - Is this enough?
    - Is it a good idea at all?
- We'll look at it anyway

Pseudo-classical inheritance

- Inheritance isn't straight forward
- We can simulate class inheritance by replacing the prototype of a class with an instance of the base class:

```
function Book(isbn, title) {
   ...
}
Book.prototype = new Lendable();
```

Pseudo-classical inheritance

- But
  - The constructor attribute is pointing at Lendable
  - We have to assign the new prototype before adding any methods to the prototype
  - We haven't inherited the members
- We can all this by creating a prototype object and adding a call in the child's constructor

Pseudo-classical inheritance

- So we handle all these with the following code:

```
function obj() {}
obj.prototype = Lendable.prototype;
Book.prototype = new obj();
Book.prototype.constructor = Book;
```

- And add a call to the base constructor:

```
function Book(isbn, title) {
   ...
   Lendable.call(this,title);
}
...
```

Pseudo-classical inheritance

- We can make this a function to reuse it:

```
extend = function(subClass, baseClass) {
   function inheritance() {}
   inheritance.prototype =
             baseClass.prototype;
   subClass.prototype =
             new inheritance();
   subClass.prototype.constructor =
             subClass;
   subClass.baseConstructor = baseClass;
   subClass.superClass =
             baseClass.prototype;
}
```

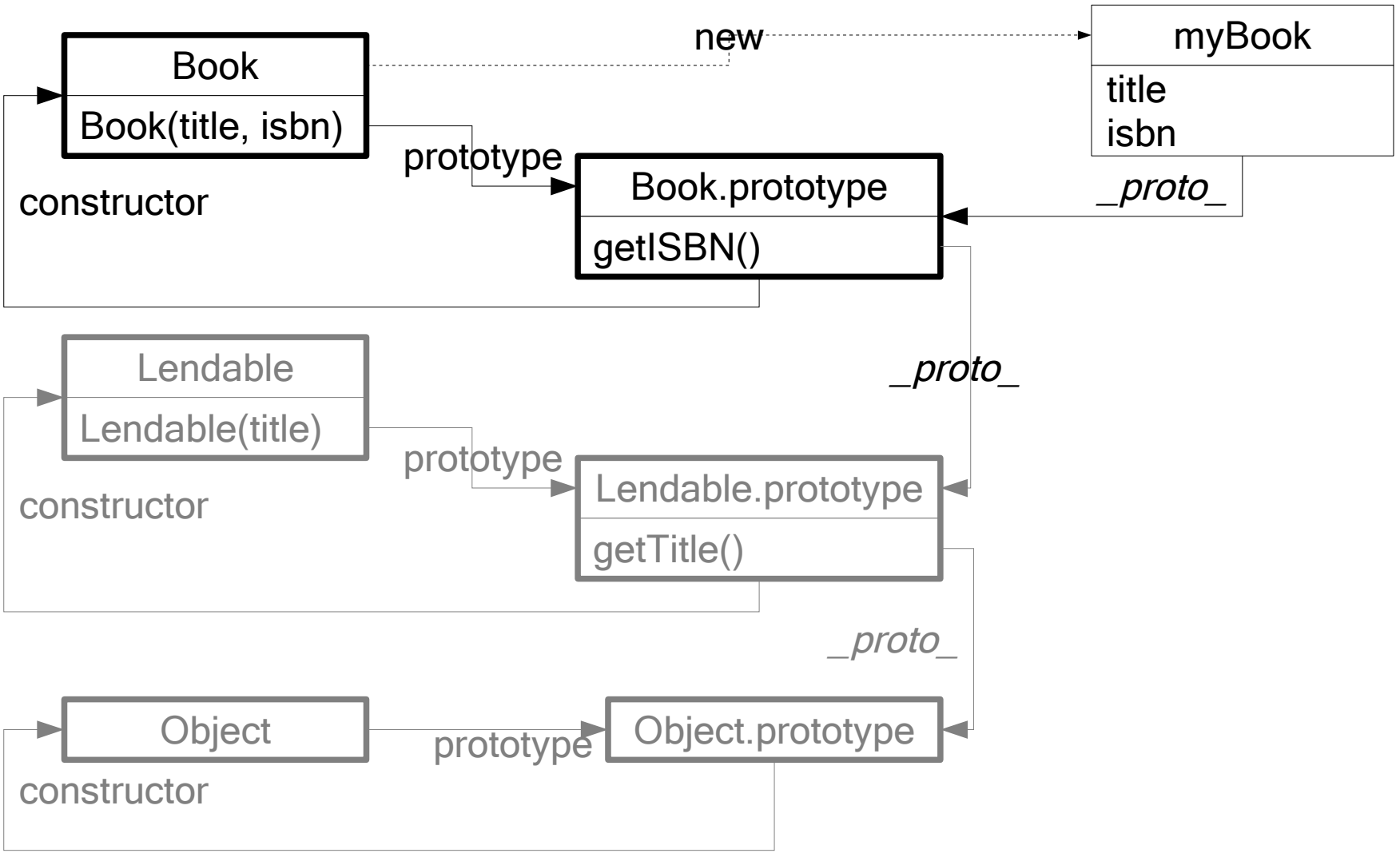
Pseudo-classical inheritance

- And use it like:

```
Book = function(isbn, title)
{
  Book.baseConstructor.call(this,
                            title);
  this.isbn = isbn;
}
extend(Book, Lendable);

...
```

Pseudo-classical inheritance



Pseudo-classical

- But still not very classical:
  - No data hiding
    - Members are public
- We can fix this too, but we need to understand closures…

Closures

- We can nest functions
- Nested functions
  - Can access variables from the outer scope
    - Lexical scoping
  - Can be referenced beyond the lifetime of the outer scope
- Let's look at some code...

Closures

- An example closure:

```
function scope(param1) {
  var var1 = 10;
  function closure(param2) {
    return param1 + var1 + param2;
  }
  return closure;
}

var closure = scope(5);
closure.call(11); // equals 26
```

Closures

- We can also construct closures as a return from a factory function:

```
var closure = function() {
   var var1 = 10;
   return {
      closure: function() {
         return var1;
      }
   }
}();
```

Closures

- Variables from the scope have value in the closure as at the end of the outside scope:

```
function scope() {
  var var1 = 10;
  function closure() {
    return var1;
  }
  var1 = 21;
  return closure;
}

// closure() returns 21;
```

Private data

- We can use closures to create private data in our classes
- If we start with the data:

```
function Lendable(title) {
  var privateTitle = title;
}
```

- privateTitle is inaccessible after the constructor has exited
  - But we can capture it in a closure

Private data

- Adding a function:

```
function Lendable(title) {
  var privateTitle = title;
  this.getTitle = function()
  {
    return privateTitle;
  }
}
```

- The only problem is that each instance of Lendable gets its own copy of getTitle()

Private methods

- In a similar way we can have private methods:

```
function Lendable(value) {
  this.val = value;
  var that = this;
  function privateCalc() {
    return (that.title % 2);
  }
  this.getResult = function() {
    return privateCalc();
  }
}
```

Private methods

- Each instance has a copy of privateCalc() and getResult()
- In the private data and private method examples, the public method is termed 'privileged' as it is able to access private data
- The syntax is subtle
- 'that' required for workaround to specification issue

Classical approach

- We've looked at javascript's pseudo classical nature
  - We've got a long way
    - constructors, methods, members
    - inheritance
    - data hiding
  - Making it more palatable for programmers from classical languages
  - Shows javascript adaptability
  - But I think there might be a better way

prototypes

- Javascript is a prototype based language
- It is dynamically typed
  - We don't need type to determine whether an object is suitable for use
  - We don't need to check if an object is of a particular type
  - If it does, it is
- We may still want to re-use or share code
- We could do this without using the pseudo-classical approach

Differential inheritance

- As a concept prototype based programming
  - Is classless
  - Uses cloning and modification of existing objects (prototypes) to achieve inheritance
    - This generally known as differential inheritance
  - This style comes from Self, though there are other languages including Javascript

Differential Inheritance

- We can achieve this in javascript by simply replacing an objects prototype with an instance of the parent object
  - We did something like this in the pseudo classical approach
- This time we use objects not 'classes'

```
var lendable = {
   'title' : 'Javascript Language';
   'getTitle' : function()
                {return this.title;}
};
```

Differential Inheritance

- We can define book:

```
var book = {};
book.prototype = lendable;
// augment book
```

- This can be codified into a function:

```
function object(o) {
   function f() {}
   f.prototype = o;
   return f;
}
var book = object(lendable);
```

Differential Inheritance

- This is much simpler than the pseudo classical approach
  - No need for new anywhere (or to forget)
- There are downsides:
  - Can't use instanceof to determine type
  - Prototype chain
    - name lookup only works with public names
    - Could get long, not so good for performance
      - But most javascript applications use the network so this is generally moot
  - State dependency

Another approach

- We don't need to use the prototype chain at all
- Equally valid in a prototype based language as differential inheritance
- Like the prototype pattern in the GoF?
  - Without the classes of course
- Douglas Crockford calls this parasitic inheritance

Parasitic inheritance example

- Looking back at Lendable:

```
function Lendable(title) {
  var lendable = {};
  lendable.title = title;
  lendable.getTitle =
    function {
      return this.title;
    };
  return lendable;
}
var lendable = Lendable();
```

Parasitic inheritance example

- Then book:

```
function Book(isbn,title) {
  var book = Lendable(title);
  book.isbn = isbn;
  book.getISBN =
    function {
      return this.isbn;
    };
  return book;
}
var book = Book();
```

Parasitic inheritance

- This is more in keeping with the conceptual basis of the language
- The new object is everything the parent was and more
- No state dependency
- Problem: methods are not shared

Parasitic Inheritance

- Is good
  - expect if you want to have a million objects of the same 'type'
  - Memory usage more than the pseudo-classical approach
- We need a hybrid approach for this
  - Use a shared prototype with the public methods
  - Create the objects with this prototype

Summary of OO

- Styles of OO
  - Pseudo classical
  - Prototype based
- We don't need the class style in Javascript
  - Except for class (Java, C++?) programmers
- Parasitic Inheritance possibly more in tune with the language

Linkage

- All the examples so far have added names to the global namespace
- The global namespace is basically a global variable
  - Linkage through global variable
  - Bad for mashups and the like
- Use functions to provide namespaces to stop pollution of global namespace

Functional

- We know that in javascript functions are first class entities
  - We've already used them to create closures
  - We've returned these closures from functions
- We can pass functions as arguments to other functions
  - These functions don't need names
  - Lamda

Functional

- An example from Eloquent JavaScript:

```
function printArray(a) {
  for (var i=0; i<a.length; i++) {
    print(a[i]);
  }
}
function sumArray(a) {
  var sum = 0;
  for (var i=0; i<a.length; i++) {
    sum += a[i];
  }
  return sum;
}
```

Functional

- In each example the for loop is repeated
  - The variation point is the function applied to each element
- So we can extract this:

```
function forEach(a, action) {
  for (var i=0; i<a.length; i++) {
    action(a[i]);
  }
}
```

Functional

- And rewrite the examples:

```
function printArray(a) {
  forEach(a, print);
}

function sumArray(a) {
  var sum = 0;
  forEach(a, function(number) {
    sum += number;
  });
  return sum;
}
```

Functional

- It doesn't appear to give much advantage for these simple examples
- But it does reduce the noise
  - no more loop or index
- A function which operates on another function is called a higher order function
  - Closer to the problem
  - Less mechanism
- “More software, less code”

Functional

- The forEach function used by sumArray was an example of the functional programming *reduce*
  - from Lisp
- Reduce looks like this in javascript:

```
function reduce(fn, a, base) {
  var s = base;
  for (var i=0; i<a.length; i++) {
    s = fn(s, a[i]);
  }
  return s;
}
```

Functional

- We can write SumArray using it:

```
function sumArray(a) {
  return reduce(function(s, number) {
              return s + number;
            },
            a, 0);
}
```

- Even less code

Functional

- If there is reduce then there must be map:

```
function map(fn, a) {
  var ret = [];
  for (var i=0; i<a.length; i++) {
    ret.push(fn(a[i]));
  }
  return ret;
}
```

- You could assign back to a: `a = fn(a[i]);`
  - But that wouldn't be functional
    - side effects

Functional

- So we can use map to multiply all array entries by 2:

```
function doubleArrayElements(a) {
  return map(function(number) {
            return number * 2;
          },
          a);
}
```

Functional

- What about recursive unnamed functions?
- If we wanted to calculate the fibonacci number for each array entry (say for an interview):

```
function calculateFabonacci(a) {
 return map(function(n) {
        f = arguments.callee;
        var s=0;
        if (n=0) {return s;}
        else if (n=1) {return s+=1;}
        else {return (f(n-1) + f(n-2));}
      },
      a);
}
```

Functional

- From these simple examples we can see the ability to reduce code and improve expressiveness
- Not bad for a simple language
  - Or maybe it is just misunderstood

Tools

- A brief look at some tools that help the javascript programmer
  - editors
  - debuggers
  - libraries
  - testing

JSLint

- A tool by Douglas Crockford
- Both a lint and layout checker
- Immediately annoying and useful in equal measure
- Helps avoid bad parts of javascript
- JSLintForJava from google allows javscript lint to be used from Ant

Editors

- Many editors provide highlighting for javascript
- Notable editors
  - Eclipse with JSEclipse (used by colleague)
  - NetBeans (used by another colleague)
  - Scintilla - has good highlighting, folding etc
- No real killer IDE

Debuggers

- Mainly browser based:
  - Mozilla/Firefox
    - Firebug
    - Javascript debugger
    - Venkman (dormant)
  - IE
    - Microsoft script debugger (Office and Visual Studio)
  - Chrome
    - Has one built in (not so good yet)
- Also one for Rhino (Mozilla Rhino debugger)

Libraries

- Main libraries:
  - Prototype (for dynamic web applications)
  - Dojo Toolkit (similar to Prototype)
  - X (widgets and more)
- As many AJAX libraries as can be imagined
- Not forgetting:
  - GWT (which makes this session a bit redundant)

Testing

- Selenium (we use this for system testing)
  - Integrates with ant and junit
- Rhinounit (unit testing on Rhino)
  - Also integrates with ant and junit
- JSUnit
  - Looks good but I've had trouble making it work
- JSNUnit
  - For .NET, but I haven't tried it
- JSCoverage/JSMock

Javascript

- Dynamically typed
- Objects are containers
- Lambda (functions are first class entities)
- Prototype based
- Simple linkage
- Interpreted
- Thread neutral
- Garbage collected