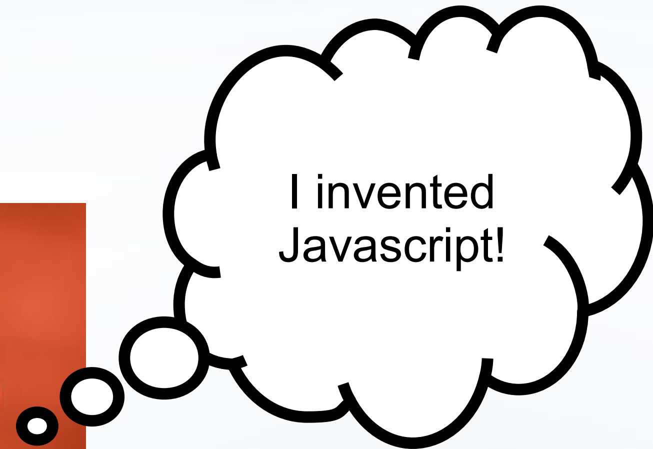
JAVASCRIPT

The Little Language That Can

espace®

What is Javascript?



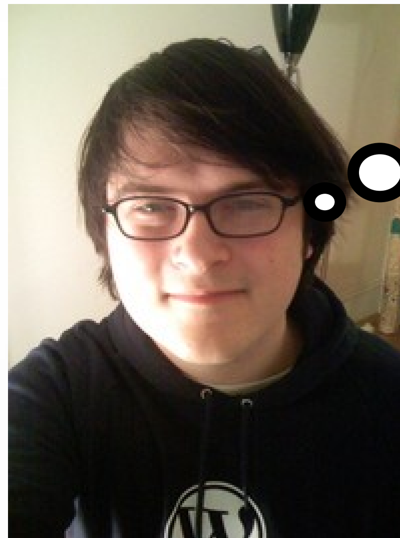
Brendan Eitch, CTO, Mozilla Corp.

espace®

I discovered Javascript (and JSON)

Douglas Crockford, Javascript Guru, Yahoo

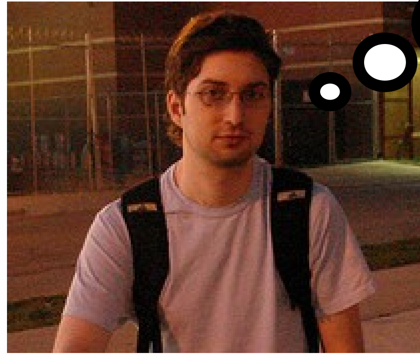
espace®



I am helping push Javascript forward

John Resig, JQuery author, Mozilla Corp.

I am making Javascript as much fun as Ruby



Sam Stephenson, Prototype.js author, 37Signals

espace®

And all I did
was this
lousy
presentation

Muhammad Ali, CTO, eSpace Technologies

Javascript Is An

Object oriented, *[like C++, Java, Eiffel, Smalltalk]*
prototype based, *[like IO, Lua, Self]*
dynamically typed, *[like Objective-C, Python, Ruby]*
***functional*,** *[like Lisp, Scheme]*
dynamic language *[like Perl, PHP, Groovy]*

espace®

Data Types

- Primitive types
  - boolean (true,false)
  - number (floating point double)
  - string (chars are strings with length = 1)
- Special values
  - null
  - undefined
- Everything else is an object

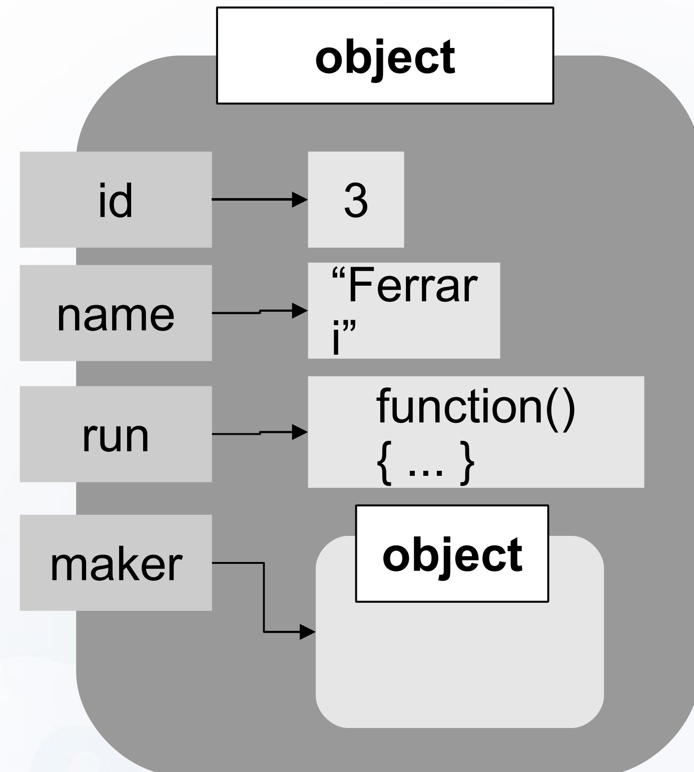
espace®

Everything is an object

- There is no distinction between objects and classes
- Objects inherit from objects
- Even functions are objects

All objects are hashes

- Objects are containers with slots
- Slots can contain anything
- Even functions
- Or other objects



espace®

Slots (properties) can be accessed

- Via dot notation

```
object.property
object.property = newValue
```

- Or via bracket notation

```
object[“property”]
object[“property”] = newValue
```

- The second method allows for using reserved words as property names.
- You can also consider it a call by name

espace®

Creating new objects

- Via the Object constructor function:
  - var object = new Object();
- Or via the Object literal:
  - var object = { };
- Both are the same
- The second allows for default properties:
  - var object = {name:'object', createdAt:new Date()}

espace®

Example

```
var tamer = {
        id : 1, title: “Manager”,
        fullTitle : function(){
                return “Opertations” + this.title;
        }
}
tamer.title              => “Manager”
tamer[“title”]           => “Manager”
tamer.fullTitle()        => “Operations Manager”
tamer[“fullTitle”]()     => “Operations Manager”
tamer[“fullTitle”]       => function
```

espace®

Traversing an object's properties

- Given an object *obj* the following code:

  for(var i in obj){ … }

  will traverse property names in obj, while:

  for each(var i in obj){ … }

  will traverse property values in obj.

- Example:

```
var properties = “”; //empty string
var plane = {name:'Boeing', model:'747'};
for(var p in plane){
        properties += p + “:” + plane[p] + “\n”;
}
```

  name:Boeing
  model:747

DEFINING NEW OBJECT TYPES

espace®

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars.

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Notice the use of this to assign values to the object's properties based on the values passed to the function.

Now you can create an object called mycar as follows:

```
mycar = new Car("Eagle", "Talon TSi", 1993);
```

espace®

An object can have a property that is itself another object. For example, suppose you define an object called person as follows:

```
function Person(name, age) {
   this.name = name
   this.age = age
}
```

and then instantiate two new person objects as follows:

```
rand = new Person("Rand McKinnon", 33);
ken = new Person("Ken Jones", 39);
```

espace®

Then you can rewrite the definition of car to include an owner property that takes a person object, as follows:

```
function Car(make, model, year, owner) {
  this.make = make; this.model = model;
  this.year = year; this.owner = owner
}
```

To instantiate the new objects, you then use the following:

```
car1 = new Car("Eagle", "Talon TSi", 1993, rand);
car2 = new Car("Nissan", "300ZX", 1992, ken);
```

Then if you want to find out the name of the owner of car2, you can access the following property:

```
car2.owner.name //returns "Ken Jones"
```

espace®

Class-Based vs. Prototype-Based Languages

Class-based object-oriented languages, such as Java and C++, are founded on the concept of two distinct entities: classes and instances.

A prototype-based language, such as JavaScript, does not make this distinction: it simply has objects.

A prototype-based language has the notion of a prototypical object, an object used as a template from which to get the initial properties for a new object.

Any object can be used as a prototype for another object.

OBJECTS IN PROTOTYPE BASED LANGUAGES

espace®

Class-based (Java)

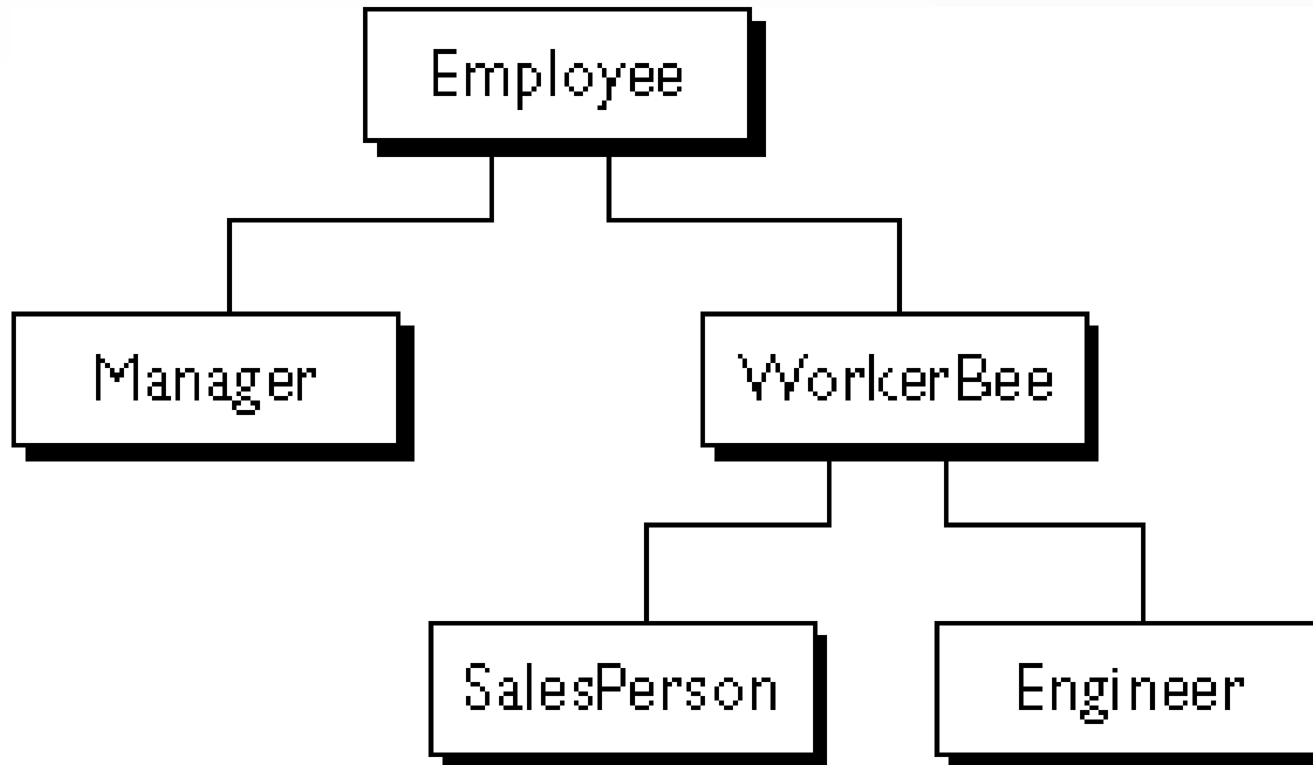
- **Class and instance are distinct entities.**
- **Define a class with a class definition; instantiate a class with constructor methods.**
- **Create a single object with the new operator.**
- **Construct an object hierarchy by using class definitions to define subclasses of existing classes.**
- **Inherit properties by following the class chain.**

Protoype-based (JavaScript)

- **All objects are instances.**
- **Define and create a set of objects with constructor functions.**
- **Create a single object with the new operator.**
- **Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.**
- **Inherit properties by following the prototype chain.**

espace®

The Employee Example



espace®

Creating the Hierarchy

Employee

```
function Employee () {
   this.name = "";
   this.dept = "general";
}
```

Manager

```
function Manager () {
   this.reports = [];
}
Manager.prototype=new Employee;
```

WorkerBee

```
function WorkerBee() {
   this.projects = [];
}
WorkerBee.prototype=new Employee;
```

SalesPerson

```
function SalesPerson () {
   this.dept = "sales";
   this.quota = 100;
}
SalesPerson.prototype=new WorkerBee;
```

Engineer

```
function Engineer () {
   this.dept = "engineering";
   this.machine = "";
}
Engineer.prototype=new WorkerBee;
```

JAVASCRIPT INHERITANCE

```
function Manager () {
this.reports = [];
}
Manager.prototype = new Employee;
function WorkerBee () {
this.projects = [];
}
WorkerBee.prototype = new Employee;
```

```
public class Manager extends
    Employee {
  public Employee[] reports;
  public Manager () {
    this.reports = new Employee[0];
  }
}
public class WorkerBee extends
    Employee {
  public String[] projects;
  public WorkerBee () {
    this.projects = new String[0];
  }
}
```

JAVASCRIPT INHERITANCE

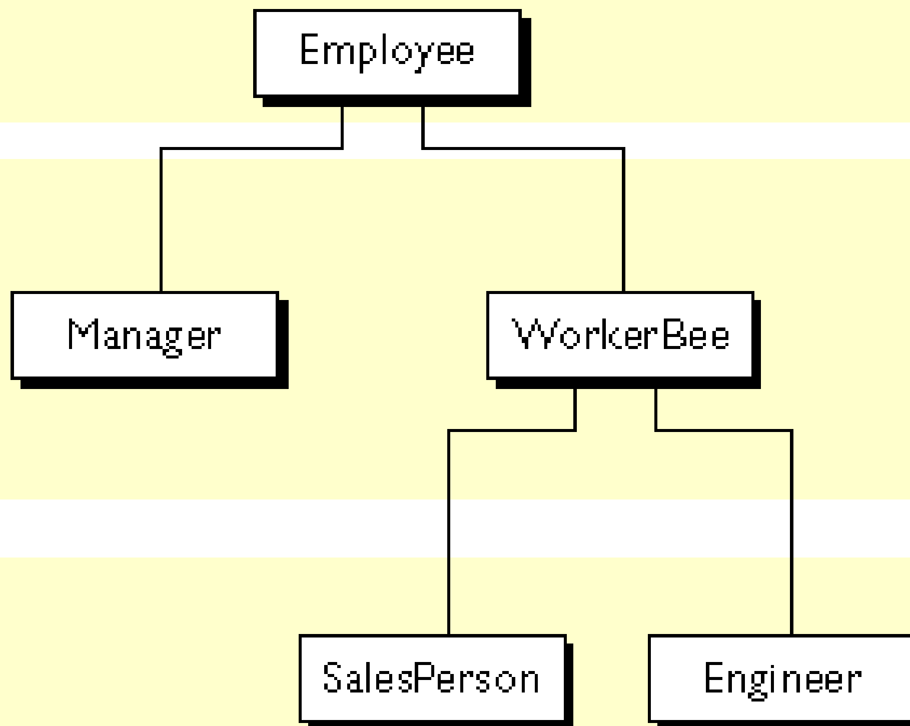
```
function SalesPerson () {
  this.dept = "sales";
  this.quota = 100;
}
SalesPerson.prototype = new WorkerBee;
function Engineer () {
  this.dept = "engineering";
  this.machine = "";
}
Engineer.prototype = new WorkerBee;
```

```
public class SalesPerson extends
    WorkerBee {
  public double quota;
  public SalesPerson () {
    this.dept = "sales";
    this.quota = 100.0;
  }
}
public class Engineer extends
    WorkerBee {
  public String machine;
  public Engineer () {
    this.dept = "engineering";
    this.machine = "";
  }}
```

JAVASCRIPT INHERITANCE

espace®

Object hierarchy



Individual objects

```
jim = new Employee
jim.name is ''
jim.dept is 'general'

sally = new Manager
sally.name is ''
sally.dept is 'general'
sally.reports is []

mark = new WorkerBee
mark.name is ''
mark.dept is 'general'
mark.projects is []

fred = new SalesPerson
fred.name is ''
fred.dept is 'sales'
fred.projects is []
fred.quota is 100

jane = new Engineer
jane.name is ''
jane.dept is 'engineering'
jane.projects is []
jane.machine is ''
```

espace®

Object hierarchy

Individual objects

Employee

```
function Employee () {
  this.name = "";
  this.dept = "general";
}
Employee.prototype.specialty = "none"
```

```
jim = new Employee
```

jim.specialty is 'none'

Manager

WorkerBee

```
function WorkerBee() {
  this.projects = [];
}
WorkerBee.prototype=new Employee;
```

```
mark = new WorkerBee
```

mark.specialty is 'none'

SalesPerson

Engineer

```
function Engineer () {
  this.dept = "engineering";
  this.machine = "";
}
Engineer.prototype = new WorkerBee;
Engineer.prototype.specialty = "code"
```

```
jane = new Engineer
```

jane.specialty is 'code'

espace®

Public

The members of an object are all public members. Any function can access, modify, or delete those members, or add new members.

There are two main ways of putting members in a new object:

In the constructor

This technique is usually used to initialize public instance variables.

The constructor's this variable is used to add members to the object.

```
function Container(param) {
    this.member = param;
}
```

So, if we construct a new object

```
var myContainer = new Container('abc');
```

then myContainer.member contains 'abc'.

espace®

Private

Private members are made by the constructor. Ordinary vars and parameters of the constructor becomes the private members.

```
function Container(param){this.member=param; var secret=3; var self=
this;}
```

This constructor makes three private instance variables: param, secret, and self. They are accessible to private methods. Private methods are inner functions of the constructor.

```
function Container(param) {
    function dec() {
        if (secret > 0) {secret -= 1;return true;} else {return false;}
    }
    this.member = param; var secret = 3; var self = this;
}
```

Private methods cannot be called by public methods. To make private methods useful, we need to introduce a privileged method.

espace

Privileged

A privileged method is able to access the private variables and methods, and is itself accessible to the public methods and the outside. It is possible to delete or replace a privileged method, but it is not possible to alter it, or to force it to give up its secrets.

Privileged methods are assigned with **this** within the constructor.

```
function Container(param) {
    function dec() {
        if (secret > 0) {secret -= 1;return true;} else {return false;}
    }
    this.member = param; var secret = 3; var that = this;
    this.service = function () {
        if (dec()) { return that.member; } else { return null; }
    };
}
```

espace®

Patterns

Public

```
function Constructor(...) {
  this.membername = value;
}
Constructor.prototype.membername = value;
```

Private

```
function Constructor(...) {
   var that = this; var membername = value;
   function membername(...) {...}
}
```

Privileged

```
function Constructor(...) {
   this.membername = function (...) {...};
}
```

HIGHER ORDER FUNCTIONS

espace®

- Functions act as first class citizens in the language
- Functions can be passed to functions
- Functions are actual Objects
- Functions are reflective

espace®

Example, The Array.sort method:

The code [2,3,1,4].sort() will result in [1,2,3,4]. But the sort method can do more. It allows a compare function as an optional argument. Suppose we have an array of objects. Each object has a date property, and we want to sort the objects by their date value

```
arrayOfObjects.sort(
    function (x,y) {
        return x.date-y.date;
    })
```

The compare function is called regularly during sorting. It must return a negative value when x comes before y, zero when they are equal and a possitive value when x comes after y.

espace®

Example, Returning Functions

```
function mime_parser(mime_type) {
  switch(mime_type){
        case “html/text” : return function(data){ /* process html */ };
        case “xml” : return function(data){/* process xml */};
        default: return function(data){/* default mime processor */}
    }
}
var process = mime_parser(“html/text”);
var processed_html = process(unprocessed_html);
```

CLOSURES

Javascript Functions are closures which means:

- They have access to the variables defined in their enclosing block (lexical scope)
- They retain their enclosing environment even when it gets out of scope

espace®

Example:

```
function plus(number) {
    return function(another_number){
        return number + another_number; /* number is private to plus */
    }
}
var plus2 = plus(2);
var x = plus2(5); /* x = 7 */

var plus3 = plus(3);
var y = plus3(6); /* y = 9 */

var z = plus2(plus3(1)); /* z = 6 */
```

espace®

Accessing public variables in a closure:

```
function amplify() {
    this.factor = 2;
    return function(signal){
        return signal * this.factor ; /* factor is a public variable */
    }
}
var amplifier = amplify();
var new_signal = amplify(signal); /* ERROR: this.factor is undefined */
```

The meaning of *this* has changed because of the change of scope.

espace®

Solution:

```
function amplify() {
    this.factor = 2;
    var self = this; /* the private variable self now points to this */
    return function(signal){
        return signal * self.factor ;
                /* we can access factor through self now */
    }
}
var amplifier = amplify();
var new_signal = amplify(signal); /* correct result */
```

espace®

What are they good for?

- Event Handlers
- Callback Functions (in async. environments like AJAX)
- Many novel uses....

CURRYING

- Currying is a useful technique, with which you can partially evaluate functions!
- It depends heavily on higher order functions and closures

Example

We need a function that adds two numbers, but when we pass only one number as an argument, it will return another function that accepts one argument and adds it to the previous number.

```
var result = add(2,3);/* result = 5 */
var add_more = add(2);
/* add_more is a function that stores the value 2 */
var result2 = add_more(3); /* result2 = 5 */
```

espace®

Implementing Add

```
function add(a, b) {
 if (arguments.length == 1) {
/* js gives us access to passed arguments */
        return function(c) { return a + c }
    } else if(arguments.length == 2) {
        return a + b;
    }
}
var result = add(2,3); /* result = 5 */
var add_more = add(2);
/* add_more is a function that stores the value 2 */
var result2 = add_more(3); /* result2 = 5 */
```

VISITORS, BETTER ITERATORS

espace®

Using function passing can prove very helpful for iterating over list elements. We will show some useful iteration functions that follow the visitor pattern.

Iterator One, apply something to each element

```
/* old way */

for (var i=0; i< list.length;i++) { /* do something to list[i] */ }

/* new way */

list.each(function(element){ /* do something to element */ })
```

Instead of iterating over the elements, a function is passed to the each method, which will iterate over the list and pass each element to the function

VISITORS, BETTER ITERATORS

```
/* implementation of each */
Array.prototype.each = function(visitor_function){
    for(var i=0; i < this.length; i++){
        visitor_function(this[i]);
    }
}
/* example: multiply each element by two and append them to a new array*/
var list2 = [ ];
list.each(function(element){
    list2.push(element * 2);
})
```

Instead of iterating over the elements, a function is passed to the each method, which will iterate over the list and pass each element to the function

espace®

Iterator Two, select some elements from a list

```
/* return a list of elements bigger than 5 */
list.select(function(element){ return element > 5 })
/* implementation of select */
Array.prototype.select = function(selector_function){
    var result = [ ];
    this.each(function(element){
        if(selector_function(element))result.push(element);
    })
    return result;
}
```

espace®

Iterator Three, reduce the list to a single value

```
/* sum all elements */
list.inject(0, function(a,b){ return a + b })
/* multiply all elements */
list.inject(1, function(a,b){ return a * b })
/* implementation of inject */
Array.prototype.inject = function(initial_value, injector_function){
    var result = initial_value;
    this.each(function(element){
        result = injector_function(result, element);
    })
    return result;
}
```

espace®

Summary

```
/* iterate over all elements */
list.each(function(element){ /* do something to element */ })
/* return a list of elements bigger than 5 */
list.select(function(element){ return element > 5 })
/* return a list of elements that are not arrays */
list.select(function(element){ return !element.constructor == Array })
/* return a list of female students */
student_list.select(function(student){ return  student.gendre==”female”})
/* sum all elements */
list.inject(0, function(a,b){ return a + b })
/* multiply all elements */
list.inject(1, function(a,b){ return a * b })
```

PARASITIC INHERITANCE

espace®

Another way to inherit form objects other than through their prototypes:

- Call the parent object's constructor in your own constructor.
- Modify the resulting object as you please.
- Return the modified object as if it was you.

```
function Car(model, make){
    this .model = model, this.make = make;
}
function SportsCar(speed){
    var car = new Car();
    car.speed = speed, car.turbo = true; /* parasitic traits */
    return car;
}
```

TRUE PROTOTYPAL OBJECT CREATION

Javascript uses the keyword new to create objects from constructor functions

```
new f()       //creates an object that inherits from f.prototype
```

This was done to be more familiar to classical programmers. But it deviates away from true prototypal methods.

In a pure prototypal language you just need to get a new prototype from an existing object. We can mimic that in Javascript by defining the following

```
function object(o) {
    function F() { }
    F.prototype = o;
    return new F();
}
var newObject = object(oldObject)
```

Aspect Oriented Programming:

- Concerned with different aspects of the program
- Not interested in a specific object behavior (mostly)
- Implemented using interception.
- Most commonly used as advising (before, after, around) methods

```
function User(){
    this.login = function(name, password){
        //login
    }
}
var user = new User();
function log_action(args){ /* ... */ }
Aspects.adviceBefore(User.login, log_action);
```

AOP

```
function User( ){
}
User.prototype.login = function(name, password){ /* do login */ }
function log_action(){ /* do logging */ }
function adviceBefore(obj, fn, advice){
    obj.prototype["__"+fn+"__"] = obj.prototype[fn];
    obj.prototype[fn] = function(){
        advice.apply(this,arguments);
        obj.prototype["__"+fn+"__"].apply(this,arguments);
    }
}
adviceBefore(User,"login",log_action);
user = new User();
user.login("ahmed","salem");
```

espace®

espace

```
function User( ){
}
User.prototype.login = function(name, password){ /* do login */ }
function log_action(){ /* do logging */ }
function adviceAfter(obj, fn, advice){
    obj.prototype["__"+fn+"__"] = obj.prototype[fn];
    obj.prototype[fn] = function(){
        obj.prototype["__"+fn+"__"].apply(this,arguments);
        advice.apply(this,arguments);
    }
}
adviceAfter(User,"login",log_action);
user = new User();
user.login("ahmed","salem");
```

espace

Used mainly for list generation (hence the name)

```
function square(obj) {
   for each ( var i in obj )
     yield i*i;
}
var someNumbers = {a:1,b:2,c:3,d:4,e:5,f:6};
var it = square(someNumbers);
  try {
   while (true) {
     document.write(it.next() + "<br>\n");
   }
 } catch (err if err instanceof StopIteration) {
            //we are done
 }
```

espace®

Short hand form of generators

```
var someNumbers = {a:1,b:2,c:3,d:4,e:5,f:6};
 var it = ( i*i for each (i in someNumbers) if (i%2==0))
  try {
   while (true) {
     document.write(it.next() + "<br>\n");
   }
 } catch (err if err instanceof StopIteration) {
          //we are done
 }
```

espace®

In place generation of arrays

```
var someNumbers = {a:1,b:2,c:3,d:4,e:5,f:6};
var array = [( i*i for each (i in someNumbers) if (i%2==0))]

/* array = [4,16,36] */
```

espace®

Brendan & Co. are working hard to implement:

- Program Units and Packages
- Optional Typing
- Tamarin Integration In Mozilla (ActionScript + SpiderMonkey)
- Other projects are targeting Tamarin to:
  - Integrate it with other browsers
  - Map other languages to it : Python and Ruby
- Enhance Javascript for Server Side usage
  - Rhino On Rails (effort @ Google)

espace®

Ultra Cool New Javascript Mascot
(proposed by Brendan)

espace®

Phew!

That was long!

espace®

WAIT!

There is more!

espace®

Joking

Enough for today

Thank You

Mohammad A. Ali
CTO, eSpace Technologies