JavaScript in the Enterprise

Attila Szegedi, Chief Software Architect
Adeptra Inc.

Programming language choice is an implementation detail.

Why use JavaScript at all?

Rapid development

- Diminished by lack of tooling
- Not necessarily a benefit if the operational rollout is heavyweight.

Tailored environment

- JavaScript provides a very minimal environment
- You get to tailor it to your organization's needs
- First-class functions and dynamic access to them prevent undesired access to broader APIs.
  - I.e. no stray Thread.sleep() calls possible!
- Sandboxing improves security
- Easy to implement domain specific APIs

People aspect

- Wide hiring pool for people with JavaScript skills.
- They’re used to HTML DOM and browser APIs though.
- Still less of a transition than switching from Java.

Scalability

- Use an interpreted dynamic language for scalability?

Scalability problem

- Hundreds of thousands of tasks executing concurrently.
- Most of the time, they're blocked waiting for something
  - human action
  - web service response
  - time window
- Not an issue with client side JS

Architectural solutions

- One thread per task
- State machines (messaging middleware falls under this)
- Continuation-passing style
- Stack-based continuations

State machines

- Right solution at the right granularity
  - High system level
  - Coarser granularity
  - Ideally, around a message passing middleware

Continuation-passing style

- Viable in closure-friendly languages
  - JavaScript qualifies as one
  - Java doesn’t qualify
    - the amount of visual noise is staggering!
- Fact that execution is suspended still appears in API
  - Developers need to be aware of it

Stack-based continuations

- My personal favorite (on right system level)
- Suspension of execution, transfer to a different processing node, etc. completely hidden from API
- Not standard in JavaScript, though
- Mozilla Rhino on JVM supports them

Example with a state machine

```
    ...
    httpRequestToMyCompanyId = doHttpRequest("http://www.mycompany.com", "GET",
        headers);
    return;
}

function onHttpResponse(event) {
    if(event.requestId == httpRequestToMyCompanyId) {
        if(event.statusCode == 200) {
            ....
        }
    }
}
```

Example with CPS

```
...
doHttpRequest("http://www.mycompany.com", "GET", headers,
    new function(response) {
        if(response.statusCode == 200) {
            ...
        }
    });
```

Much better, as there's no longer need for explicit correlation.

Example with stack continuations

```
...
var response = doHttpRequest("http://www.mycompany.com", "GET", headers);
if(response.statusCode == 200) {
    ...
}
```

As if you were writing vanilla procedural code. Also, can use try-catch exception handling for IO failures.

Example with Narrative JS

```
function f(n) {
    return doHttpRequest->("http://www.mycompany.com", "GET", headers);
}
```

```
function f(n){var njf1=njen(this,arguments,"n");nj:while(1){switch(njf1.cp){case
0:njf1.pc(1,null,
doHttpRequest,["http://www.mycompany.com","GET",headers]);case
1:with(njf1)if((rv1=f.apply(c,a))==NJSUS){return fh;}return njf1.rv1;break nj;}}}
```

Needs separate NJS->JS compilation and a small runtime library.

Example with Narrative JS

You still need to use an explicit “yield” operator.

Continuations benefits

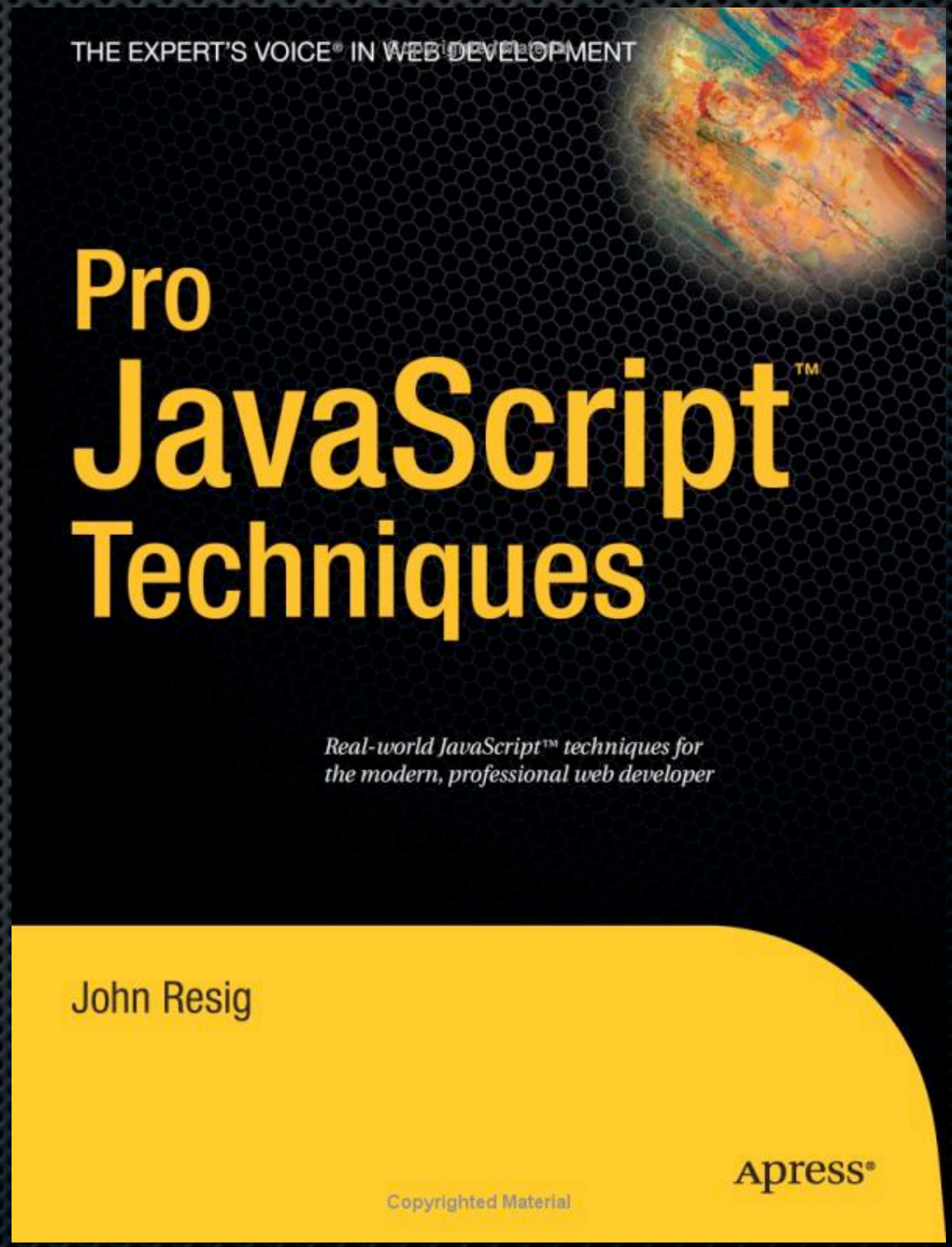
- Scalable code
- Easy to write code
- Deal breaker for us
- Demo

Organizational aspects

- Hiring pool
- Separate engineering teams for different levels of system
  - You don’t need language separation for this, but
  - “tailored environment” encourages it.
  - Also, higher level code then has fewer assumptions about runtime environment.

Less rosy organizational aspects

- Keeping high code quality is a challenge.
- “Everything public and global” disease.
- Your developers mus understand JS runtime
- Fortunately there’s a cure.



Lexical scoping for hiding

```
(function() {
    ...
})();
```

Lexical scoping for hiding

```
(function() {
   this.foo = function() {
       bar();
   };

   function bar() {
       print("bar invoked");
   }
})();
```

```
js> foo();
bar invoked
js> bar();
js: "<stdin>", line 16: uncaught JavaScript runtime exception: ReferenceError:
"bar" is not defined.
    at <stdin>:16
```

Namespacing

```
/** @namespace */
var MYMODULE = {};

(function() {
    MYMODULE.foo = function() {
        bar();
    };

    function bar() {
        print("bar invoked");
    }
})();
```

Private fields in a constructor

```
function User(name, age) {
    var year = (new Date()).getFullYear() - age;
    this.getYearBorn = function() {
        return year;
    }
    this.name = name;
}
```

```
js> var bob = new User("Bob", 27);
js> bob.getYearBorn();
1982
js> bob.year;
js> bob.name;
Bob
```

Adopt JSDoc

```
/**
 * @constructor
 * @param name {string} name of the user being created
 * @param age {number} age of the user being created
 * @return a new User object
 */
function User(name, age) {
    var year = (new Date()).getFullYear() - age;
    this.getYearBorn = function() {
        return year;
    }
    this.name = name;
}
```

Why am I telling you this?

- You'll have a big body of code in JavaScript
- You will want to have it maintainable

Horror code 1: dead stores

```
var map = new java.util.HashMap();
map = someFunctionThatReturnsMap();
```

- Uses a Java class
- Creates a map that is immediately thrown away
  - Compensating for lack of type declarations?

Horror code 2: terminology mismatch

```
var fareType;
switch(trip.fareType) {
    case "O": {
        fareType = "oneWay";
        break;
    }
    case "D": {
        fareType = "dayReturn";
        break;
    }
    case "M": {
        fareType = "monthReturn";
        break;
    }
    default: {
        fareType = "unknown";
    }
}
```

Horror code 2: terminology mismatch

```
var fareType = {
    O: "oneWay",
    D: "dayReturn",
    M: "monthReturn"
}[fare.fareType] || "unknown";
```

```
var ft = fare.fareType;
var fareType =
    ft == "O" ? "oneWay" :
    ft == "D" ? "dayReturn" :
    ft == "M" ? "monthReturn" :
                "unknown";
```

Horror code 3: fallback to Java

```
if(x.toString().equals(“”) ...

if(x == “”)
```

```
var list = new java.util.ArrayList();
list.add(x);
list.add(y);
list.get(1)

var list = [];
list.push(x);
list.push(y);
list[1]
```

Code quality

- Test-driven methodology helps.
  - Drives architecture toward smaller, independent units
- Code reviews for revealing working but smelly code.
- Documentation
- Unit testing as part of build
- Static analysis as part of build
  - JSLint, Fortify, Yasca, …

Modularity

- You need to create a script loading mechanism
- Proprietary “include()” function is sufficient
  - `include("com/mycompany/workflow/event.js");`
- But watch out for standardization efforts
  - Eclipse OSGi-like module system for JS at http://wiki.eclipse.org/E4/JavaScript

Include in Rhino

```
public class MyHostObject extends ScriptableObject {

    private final ScriptStorage scriptStorage;

    public String getClassName() {
        return "MyHostObject";
    }

    public void jsFunction_include(Scriptable scope, String scriptName) {
        Context cx = Context.getCurrentContext();
        Script script = scriptStorage.getScript(scriptName, cx);
        script.exec(cx, scope);
    }
}
```

```
ScriptableObject.defineClass(topScope, EngineApi.class);
MyHostObject hostObject = (MyHostObject)cx.newObject(topScope,
    “MyHostObject”);
ScriptableObject.putProperty(topScope, "hostObject", hostObject);
```

Quick’n’dirty config system

```
var voice="John";
var language="English";
include(“config.js");
```

config.js:

```
voice="Emma";
```

- Can use complex JSON-like config entries
- Be aware it allows for arbitrary code execution.

Threading

- JavaScript has no standard threading notion
- Programs are single-threaded by default
- You're best off if you can fit your processing into this model
  - batch processing
  - single-threaded event handlers

Shared objects

- Service-level objects
  - RESTful caches (i.e. NetKernel)
    - No need to parse that XSLT file 1M times a day
  - Named objects (i.e. through JNDI)
    - Services (i.e. async HTTP initiator)
    - Stateless, or at least immutable by scripts

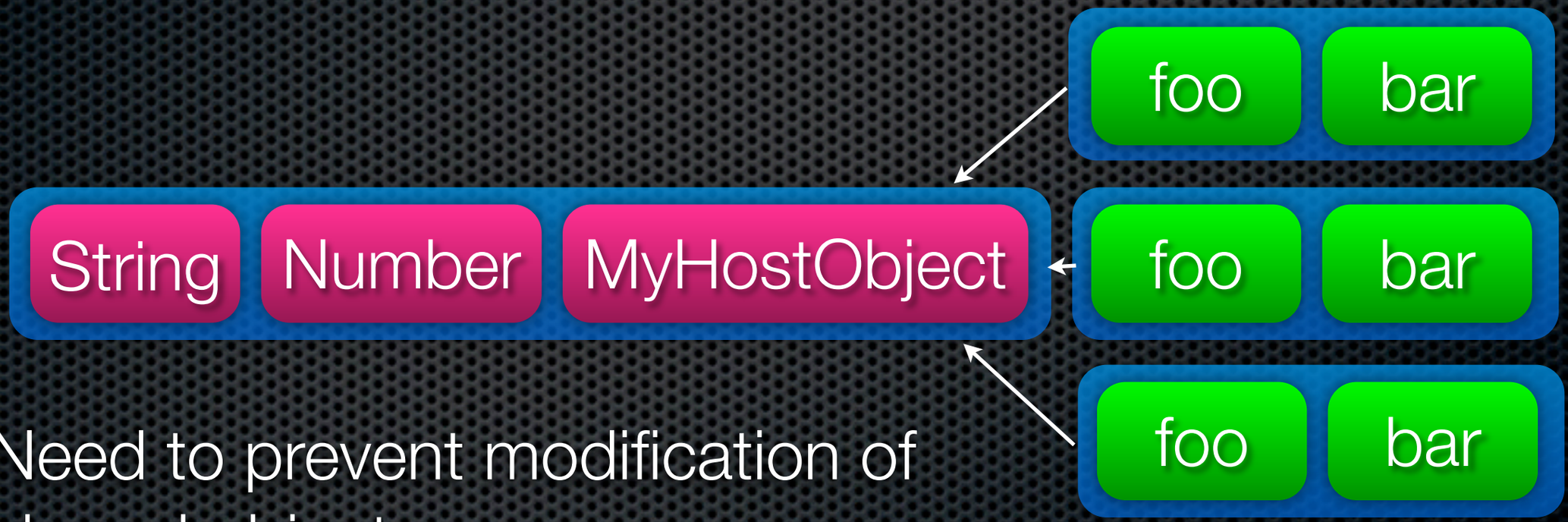
Shared standard objects

String Number MyHostObject foo bar

String Number MyHostObject foo bar

String Number MyHostObject foo bar

Shared standard objects



- Need to prevent modification of shared objects.
- Subtly changes runtime semantics

Precompilation

- Same script expected to be executed many times
- Prepare it into as efficient runtime representation as possible on first use

```
Script script = scripts.get(name);
if(script == null) {
    URL url = getScriptUrl(name);
    Reader r = new InputStreamReader(url.openStream(), "utf-8");
    try {
        script = cx.compileReader(r, url.toExternalForm());
    }
    finally {
        r.close();
    }
    scripts.put(name, script);
}
script.exec(cx, topScope);
```

Other enterprise uses

- Expression language for advanced users
- Logic spanning multiple HTTP requests (“web flow”)

Expression language

- Don’t write your own language
- JavaScript can still provide daunting to a manager writing an occasional Excel function

Expression language

```
function countList(list, condition) {
    return reduceLeft(filter(list, condition), 0, function(x) { return ++x });
}

function filter(list, condition) {
    var newList = [];
    for(var i in list) {
        var e = list[i];
        if(condition(e)) {
            newList.push(e);
        }
    }
    return newList;
}
```

```
countList(cars, function(x) { return x.year > 2006 && x.price < 10000 }) > 0
```

Expression language

countList(cars, year > 2006 and price < 10000) > 0

- Any manager who ever used Excel formulas can work with this.
- Drawback: it isn’t JavaScript though - not yet.

Expression language

countList(cars, year > 2006 && price < 10000) > 0

Expression language

```
function countList(list, condition) {
    return reduceLeft(filter(list, condition), 0, function(x)
{ return ++x });
}
countList.isLastArgumentFunction=true;
```

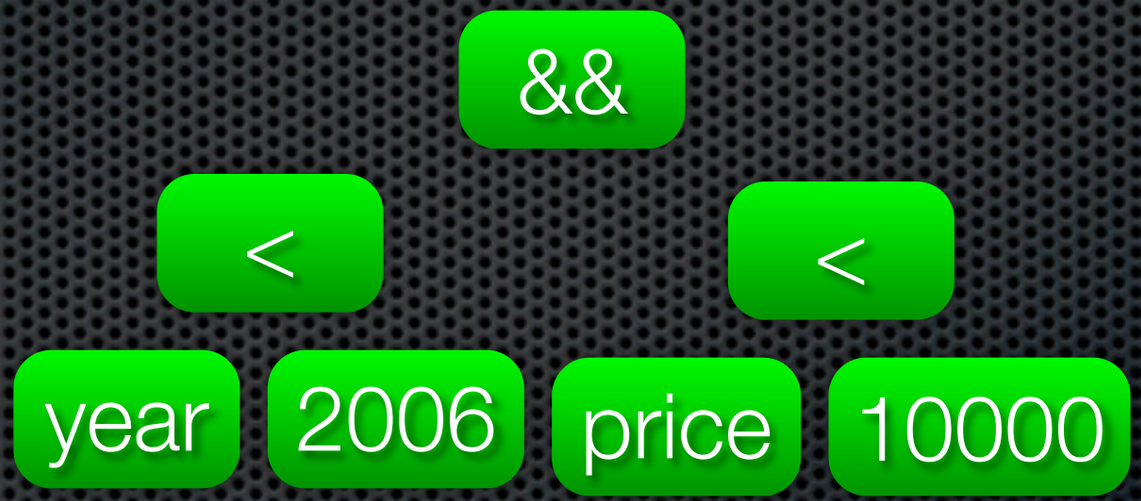
countList(cars,

```
function(x) { return
  x.year > 2006 &&
  x.price < 10000 }
```

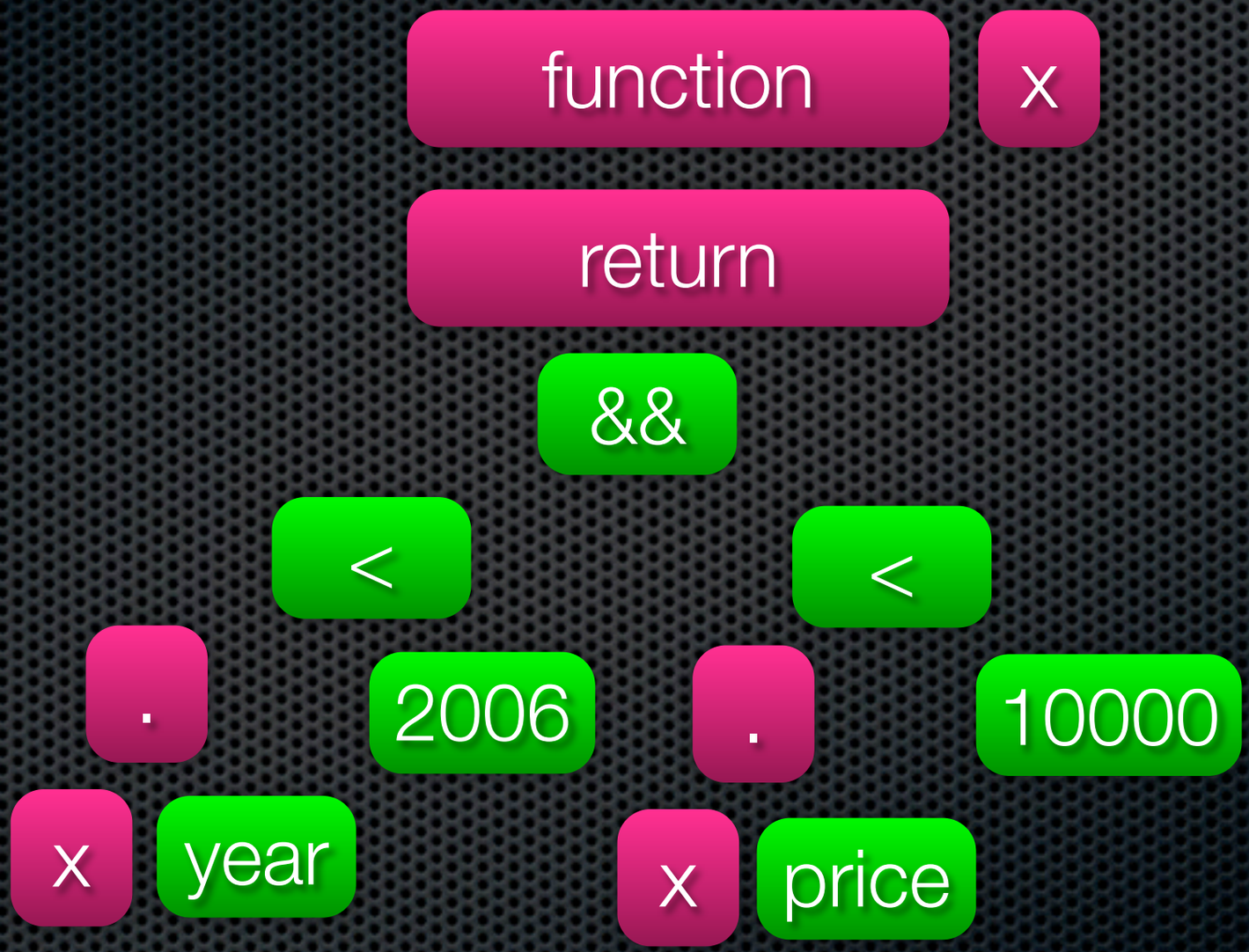
) > 0

- Some transformation required, but still...
- ... much less effort than writing your own expression language.

Expression language



Expression language



Is it worth it?

- You need to write a pre-parser to replace “and” and “or”
- You need to write a post-parser AST editor to:
  - prohibit looping constructs etc.
  - allow lifting of expressions into functions
- You can write your public functions in JS
- You still needn’t write a full parser/evaluator

Web flow

- Lots of web flow solutions are implemented as state machines, i.e. Spring Web Flow.
- They also often use XML as their state-machine definition language.

Web flow

```
<view-state id="enterBookingDetails">
    <transition on="submit" to="reviewBooking" />
</view-state>

<view-state id="reviewBooking">
    <transition on="addGuest" to="addGuest" />
    <transition on="confirm" to="bookingConfirmed" />
    <transition on="revise" to="enterBookingDetails" />
    <transition on="cancel" to="bookingCancelled" />
</view-state>

<subflow-state id="addGuest" subflow="createGuest">
    <transition on="guestCreated" to="reviewBooking">
        <evaluate expression="booking.guests.add(currentEvent.attributes.guest)" />
    </transition>
    <transition on="creationCancelled" to="reviewBooking" />
</subfow-state>

<end-state id="bookingConfirmed" >
    <output name="bookingId" value="booking.id" />
</end-state>

<end-state id="bookingCancelled" />
```

Web flow in JavaScript

- Rhino-in-Spring:

```
var addresses = {};
addresses.shippingAddress = getAddress("index", {});
addresses.billingAddress  = getAddress("billingAddress", addresses.shippingAddress);
respondAndWait("confirm", addresses);
respond("thankyou");
```

Web flow in JS

- “Subflows” come naturally - they are functions (subroutines)
- Logic is expressed as any other application logic
  - Control flow structures we know and love

Conclusion

- JavaScript has very good expressiveness, access controls, security, and tailoring capabilities
- Continuations for ultimate scalability
  - Both in backend and in webapps
- Easy to hire for
- Need to pay attention to code quality control

Thank you!