JavaScript Risks

Daniel Jackson

For now, all rights reserved. Daniel Jackson, 2012.
JavaScript is...
JavaScript is...

dynamic
› no separate compile phase
› code created & modified on the fly
JavaScript is...

dynamic
› no separate compile phase
› code created & modified on the fly

weakly typed
› automatic memory management
› primitive ops check types of args
JavaScript is...

dynamic
› no separate compile phase
› code created & modified on the fly

weakly typed
› automatic memory management
› primitive ops check types of args

map/string based
› object is map from string to values
JavaScript is...

dynamic
› no separate compile phase
› code created & modified on the fly

weakly typed
› automatic memory management
› primitive ops check types of args

map/string based
› object is map from string to values

messy
› evolved quickly, often botched
JavaScript is...

dynamic
› no separate compile phase
› code created & modified on the fly

weakly typed
› automatic memory management
› primitive ops check types of args

map/string based
› object is map from string to values

messy
› evolved quickly, often botched

all this typical for scripting languages (Ruby, Python, Perl, etc)
dynamic features
## dynamic vs. static languages

<table>
<thead>
<tr>
<th>conventional phasing</th>
<th>conventional languages</th>
<th>dynamic languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>compile time</td>
<td>Fortran</td>
<td>Smalltalk</td>
</tr>
<tr>
<td>process decls</td>
<td>Algol</td>
<td>LISP</td>
</tr>
<tr>
<td>layout code &amp; data</td>
<td>Pascal</td>
<td>Scheme</td>
</tr>
<tr>
<td>runtime</td>
<td>CLU</td>
<td>Perl</td>
</tr>
<tr>
<td>run code</td>
<td>Ada</td>
<td>Lua</td>
</tr>
<tr>
<td>modify data</td>
<td>ML</td>
<td>Ruby</td>
</tr>
<tr>
<td></td>
<td>C/C++</td>
<td>Javascript</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>Python</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
dynamic features
dynamic features

no “declarations”
› just like other statements
› so order matters!
dynamic features

no “declarations”
› just like other statements
› so order matters!

no object templates
› grow & shrink, no fixed slots
dynamic features

no “declarations”
› just like other statements
› so order matters!

no object templates
› grow & shrink, no fixed slots

runtime typing
› no errors found until it runs
dynamic features

no “declarations”
› just like other statements
› so order matters!

no object templates
› grow & shrink, no fixed slots

runtime typing
› no errors found until it runs

reflection & eval
› program ⇔ data
dynamic features

no “declarations”
› just like other statements
› so order matters!

no object templates
› grow & shrink, no fixed slots

runtime typing
› no errors found until it runs

reflection & eval
› program ↔ data

> f = 'g'
"g"
> o = {}
0bject
> o.f = 3
3
> o.g = 4
4
> o[f]
4
modifying built-ins

redefining + in Ruby

```ruby
>> 1 + 2
3
>> class Fixnum
   def +(x)
      5
   end
end
==> nil
>> 1 + 2
5
```

redefining Array.toString in JavaScript

```javascript
> a = [1,2,3]
[]
> a.toString()
"1,2,3"
> Array.prototype.toString = function (){return "hah!";}
function (){return "hah!";}
> a.toString()
"hah!"
```
modifying built-ins

redefining + in Ruby

```ruby
>> 1 + 2
3
>> class Fixnum
   def +(x)
      5
   end
end
==> nil
>> 1 + 2
5
```

can’t make robust abstract type

> client can always break it

redefining Array.toString in JavaScript

```javascript
>> a = [1,2,3]
[ ]
>> a.toString()
"1,2,3"
>> Array.prototype.toString = function (){return "hah!";}
function (){return "hah!";}
>> a.toString()
"hah!"
```
modifying built-ins

redefining + in Ruby

Can’t make robust abstract type
› Client can always break it

Modifying global namespace
› In JS, just another object
› Assignment to globals is risky (more later)

Redefining Array.prototype.toString in JavaScript

Monday, February 27, 2012
reflection & strings

...

<FORM NAME="Calc">
<INPUT TYPE="text" NAME="Input" Size="16">
<INPUT TYPE="button" NAME="one" VALUE=" 1  "
OnClick="Calc.Input.value += '1'">
<INPUT TYPE="button" NAME="three" VALUE="3"
OnClick="Calc.Input.value += '3'">
...

<INPUT TYPE="button" NAME="plus" VALUE="+
OnClick="Calc.Input.value += '+'">
<INPUT TYPE="button" NAME="DoIt" VALUE="=
OnClick="Calc.Input.value = eval(Calc.Input.value)">
</FORM>

A Javascript/HTML calculator

reflection & strings

lethal combination
› strings everywhere
› eval command

...<FORM NAME="Calc">
<INPUT TYPE="text" NAME="Input" Size="16">
<INPUT TYPE="button" NAME="one" VALUE="1" OnClick="Calc.Input.value += '1'">  
<INPUT TYPE="button" NAME="three" VALUE="3" OnClick="Calc.Input.value += '3'">  
...
<INPUT TYPE="button" NAME="plus" VALUE="+" OnClick="Calc.Input.value += '+'">  
<INPUT TYPE="button" NAME="DoIt" VALUE="=" OnClick="Calc.Input.value = eval(Calc.Input.value)"/>
</FORM>

A Javascript/HTML calculator

weakly typed
# types of types

<table>
<thead>
<tr>
<th>approach</th>
<th>examples</th>
<th>catch errors at compile-time?</th>
<th>prevent memory corruption?</th>
<th>check function arguments?</th>
<th>no implicit conversions?</th>
</tr>
</thead>
<tbody>
<tr>
<td>safe, static</td>
<td>Ada, Java, Pascal, Scala</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>safe, dynamic</td>
<td>Scheme, Smalltalk</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>weakly typed dynamic</td>
<td>Javascript, Python</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>unsafe</td>
<td>C, C++</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
heterogeneous types

```r
> favorites = [33, "hello", true]
[33, "hello", true]
```
heterogeneous types

what’s typed?
› not variables!
› not structures!

```plaintext
> favorites = [33, "hello", true]
[33, "hello", true]
```
heterogeneous types

what’s typed?
› not variables!
› not structures!

heterogeneous lists
› very convenient

> favorites = [33, "hello", true]
[33, "hello", true]
variable type
variable type

display = '';  
function press (key) {  
    if (key === '=')  
        display = eval(display);  
    else  
        display = display + key;  
    console.log(display); 
}  

a (bad) calculator in JavaScript
a (bad) calculator in JavaScript

display = '';  
function press (key) {
    if (key === '=')
        display = eval(display);
    else
        display = display + key;
    console.log(display);
}
variable type

display = '';  
function press (key) {
   if (key === '=')
      display = eval(display);
   else
      display = display + key;
   console.log(display);
}

a (bad) calculator in JavaScript

a sample run
variable type

```javascript
display = '';  
function press (key) {  
    if (key === '=')  
        display = eval(display);  
    else  
        display = display + key;  
    console.log(display);  
}  

a (bad) calculator in JavaScript

lesson: don’t use dynamically varying type unless you really need it
```

a sample run

```javascript
> press (1);
1
> typeof (display);
"string"
> press ('+');
1+
> press (2);
1+2
> press ('=');
3
> typeof (display);
"number"
> press ('+');
3+
> press (3);
3+3
> press ('=');
6
> press (4);
10
```
bitten by dynamic fields
bitten by dynamic fields

class Circle {
    int radius;

    public static void main(String[] args) {
        Circle c = new Circle();
        c.radius = 5;
    }
}
bitten by dynamic fields

```java
class Circle {
    int radius;

    public static void main(String[] args) {
        Circle c = new Circle();
        c.radius = 5;
    }
}

c.radius cannot be resolved or is not a field
```
bitten by dynamic fields

```java
class Circle {
    int radius;

    public static void main(String[] args) {
        Circle c = new Circle();
        c.radius = 5;
    }
}
```

circle = {radius: 3}
  Object
> circle.radius = 4
  4
> circle.radius
  4
> "radius is " + circle.radius
  "radius is undefined"
> circle.radius = 5
  5
> circle.radius
  4

c.circle cannot be resolved or is not a field
implicit conversions

JavaScript multiplication

> [ ] * 2
0
> [1] * 2
2
> [1,2] * 2
NaN
implicit conversions

what’s going on here?
› multiplication op converts types
› sometimes convenient
› but not worth it...

JavaScript multiplication

> [] * 2
 0
> [1] * 2
 2
> [1,2] * 2
 NaN
modifying immutables?

```javascript
> o = {}
Object
> o.f = 3
3
> o.f
3
> x = 1
1
> x.f = 3
3
> x.f
undefined
```
modifying immutables?

add a property to a number?

✓ fails silently

```javascript
> o = {}
Object
> o.f = 3
3
> o.f
3
> x = 1
1
> x.f = 3
3
> x.f
undefined
```
two primary types
maps & strings
maps & strings

in many languages

→ object = map[string, value]
maps & strings

in many languages
› object = map[string, value]

in Javascript
› arrays & functions too
› even the stack frame
maps & strings

in many languages
› object = map[string, value]

in Javascript
› arrays & functions too
› even the stack frame

in Python
› object has __dict__
maps & strings

in many languages
› object = map[string, value]

in Javascript
› arrays & functions too
› even the stack frame

in Python
› object has __dict__

```
p = {x: 1, y: 2}
Object

> p['x']
1

> p['y']
2

> this['p']
Object
 1.x: 1
 2.y: 2
 3.__proto__: Object

> this['p']['x']
1
```

JS objects

Monday, February 27, 2012
maps & strings

in many languages
› object = map[string, value]

in Javascript
› arrays & functions too
› even the stack frame

in Python
› object has __dict__

```python
>>> class Point:
...   pass
>>> p = Point()
>>> p.x = 1
>>> p.y = 2
>>> p.__dict__
{'y': 2, 'x': 1}

>>> p = {x: 1, y: 2}
Object
>>> p['x']
1
>>> p['y']
2
>>> this['p']
Object
1.x: 1
2.y: 2
3.__proto__: Object
>>> this['p']['x']
1

JS objects

Python objects

Monday, February 27, 2012
messy features
funny values
funny values

in JavaScript, many “falsy” values
  include: '', 0, null, undefined, NaN
funny values

in JavaScript, many “falsy” values include: ‘’, 0, null, undefined, NaN

NaN means “not a number”

> typeof(NaN)
"number"

> NaN === NaN
false

> NaN !== NaN
true
funny values

in JavaScript, many “falsy” values include: ‘’, 0, null, undefined, NaN

NaN means “not a number”

> typeof(NaN)
"number"
> NaN === NaN
false
> NaN !== NaN
true
funny values

in JavaScript, many “falsy” values include: '', 0, null, undefined, NaN

NaN means “not a number”

> typeof(NaN)
"number"

> NaN === NaN
false

> NaN !== NaN
true
funny values

in JavaScript, many "falsy" values
› include: ' ', 0, null, undefined, NaN

NaN means "not a number"
› typeof(NaN)
"number"
› NaN === NaN
false
› NaN !== NaN
true
funny values

in JavaScript, many "falsy" values
  include: '', 0, null, undefined, NaN

NaN means "not a number"
  typeof(NaN)
  "number"
  NaN === NaN
  false
  NaN !== NaN
  true
funny values

in JavaScript, many “falsy” values
› include: ‘‘, 0, null, undefined, NaN

NaN means “not a number”
› typeof(NaN)
"number"
› NaN === NaN
false
› NaN !== NaN
true
funny values

in JavaScript, many “falsy” values include: ‘’, 0, null, undefined, NaN

NaN means “not a number”
	> typeof(NaN)
"number"
	> NaN === NaN
false
	> NaN !== NaN
true
whitespace matters :-( 
whitespace matters :-(

› in Python, by design
› in CSS, because of units (eg ‘20px’ vs ‘20 px’)
› in JavaScript, due to semicolon insertion
whitespace matters :-(

› in Python, by design
› in CSS, because of units (eg ‘20px’ vs ‘20 px’)
› in JavaScript, due to semicolon insertion

```python
return {
    status: true
};
```

returns undefined
whitespace matters :-(

- in Python, by design
- in CSS, because of units (eg ‘20px’ vs ’20 px’)
- in JavaScript, due to semicolon insertion

```
return {
    status: true
};
```

returns undefined

```
return {
    status: true
};
```

returns object with status
strategy for ugly features
strategy for ugly features

good & bad strategy
› discover ugly feature
› bad: learn all its details and use it
› good: find way to avoid it
strategy for ugly features

good & bad strategy
› discover ugly feature
› bad: learn all its details and use it
› good: find way to avoid it

element
› == is ugly, so always use ===
strategy for ugly features

good & bad strategy
› discover ugly feature
› bad: learn all its details and use it
› good: find way to avoid it

eexample
› == is ugly, so always use ===

```bash
> 0 == ''
true
> 0 == '0'
true
> '' == '0'
false
```
strategy for ugly features

good & bad strategy
› discover ugly feature
› bad: learn all its details and use it
› good: find way to avoid it

equal sign example
› == is ugly, so always use ===

> 0 == ''
true
> 0 == '0'
true
> '' == '0'
false
> 0 === ''
false
> 0 === '0'
false
> '' === '0'
false
summary
the good parts
the good parts

In Javascript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of Javascript is so effectively hidden that for many years the prevailing opinion of Javascript was that it was an unsightly, incompetent toy. My intention here is to expose the goodness in Javascript, an outstanding dynamic programming language...
In Javascript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of Javascript is so effectively hidden that for many years the prevailing opinion of Javascript was that it was an unsightly, incompetent toy. My intention here is to expose the goodness in Javascript, an outstanding dynamic programming language...

Deep down, Javascript has more in common with Lisp and Scheme than with Java. It is Lisp in C’s clothing.

—Douglas Crockford in Javascript: The Good Parts
good, bad & awful
good, bad & awful

**good**

first class functions

closures

properties

prototypes

immutables
good, bad & awful

good

- first class functions
- closures
- properties
- prototypes
- immutables

bad, can work around

- variable scoping
- function decls
- ==
good, bad & awful

**good**
- first class functions
- closures
- properties
- prototypes
- immutables

**bad, can work around**
- variable scoping
- function decls
- ==

**awful, stuck with these**
- new & this
- implicit conversions
- semicolon insertion
- floating point nums
- 16 bit unicode

Monday, February 27, 2012
use the good, shun the bad
use the good, shun the bad

It is rarely possible for standards committees to remove imperfections from a language... But you have the power to define your own subset.

Douglas Crockford, in Javascript: The Good Parts
use the good, shun the bad

It is rarely possible for standards committees to remove imperfections from a language... But you have the power to define your own subset.

Douglas Crockford, in Javascript: The Good Parts

The Skater’s Principle of Language Use:
Stay in the middle, where the ice is thicker.

Michael Jackson, in Software Requirements & Specifications
use the good, shun the bad

It is rarely possible for standards committees to remove imperfections from a language... But you have the power to define your own subset.

Douglas Crockford, in *Javascript: The Good Parts*

*The Skater’s Principle of Language Use:*
Stay in the middle, where the ice is thicker.

Michael Jackson, in *Software Requirements & Specifications*

There is danger and misery at the edges.

Douglas Crockford, in *Javascript: The Good Parts*