JavaScript Risks

Daniel Jackson

For now, all rights reserved. Daniel Jackson, 2012.
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**

“declarations”
› just like other statements
› so order usually matters!

no object templates
› grow & shrink, no fixed slots

runtime typing
› no errors found until it runs

reflection & eval
› program ↔ data

```javascript
> f = 'g'
"g"
> o = {}
Object
> o.f = 3
3
> o.g = 4
4
> o[f]
4
```

hidden eval here!
modifying built-ins

- modifying global namespace
  - in JS, just another object
  - assignment to globals is risky (more later)

- can’t make robust abstract type
  - client can always break it

redefining + in Ruby

redefining Array.toString in JavaScript
reflection & strings

lethal combination
› strings everywhere
› eval command

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</td>
<td>Javascript, Python</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>dynamic</td>
<td></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

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

heterogeneous lists
› very convenient
variable type

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

a (bad) calculator in 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

lesson: don’t use dynamically varying type unless you really need it
bitten by dynamic fields

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

> 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
implicit conversions

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

JavaScript multiplication

```javascript
> [ ] * 2
0
> [1] * 2
2
> [1,2] * 2
NaN
```
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

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

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

in Python
› object has __dict__

in Ruby
› object is not a hash

Python objects

JS objects
messy features
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 :-(

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

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

returns undefined

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

returns object with status
strategy
solvents for home use

- isopropyl alcohol
  useful for cleaning things?
- hydrofluoric acid
  useful for dissolving rocks?

distinguish language features
  › risky but helpful: dynamic types? growable objects?
  › more trouble than they’re worth: implicit conversions
strategy for ugly features

two strategies
› bad: learn all its details and use it
› good: find way to avoid or protect against

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

```javascript
> 0 == ''
true
> 0 == '0'
true
> '' == '0'
false
> 0 === ''
false
> 0 === '0'
false
> '' === '0'
false
```
your own subset!

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
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...

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**
- 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