Ecma/TC39/2010/018

Module proposals

Some aspects to consider

Ihab Awad
Google

The Web is big

You just won’t believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think the cereal aisle at Safeway has a lot of different choices, but that’s just peanuts to the Web.

-- with apologies

Levels of generativity

To what extent is a module system *generative*?

Or: To what extent can a client *sense* that two *similar* module loading commands did or did not load exactly the same thing?

Similar → There are many ways to refer to code:

"http://foo.org/*ver*/utils.js for any *ver* ≥ 3"

"http://mirrors.com/foo/utils-v3.9.js"

A notation

Just for expository purposes

loadit("foo") -- loads module code, does not run it, returns reference

loadinstance("foo") -- loads module code, runs it and returns instantiated objects

Gen level G0

Most generative

Module code never exposed as 1st class

Module state (instances) created afresh each time a module loading command issued

```
loadinstance("foo") !== loadinstance("foo")
```

Gen level G1

Module code exposed as first-class

Module state (including internal types) always generative

```
loadit("foo") === loadit("foo")
```

```
loadit("foo").make(3, 4) !==
   loadit("foo").make(3, 4)
```

`loadit("foo")` only exposes standardized `make` interface; no internal types or anything else is available prior to instantiation

Gen level G2

Module's programmer-defined internal types available

Instance data is generative

```
loadit("foo").X === loadit("foo").X

new loadit("foo").X(3, 4) instanceof
    loadit("foo").X

new loadit("foo").X(3, 4) !==
    new loadit("foo").X(3, 4)
```

Gen level G3

Module instances are singletons

```
loadinstance("foo") === loadinstance("foo")
```

The danger ...

The greater the "gen level" :) the more ways there are for the programmer to sense -- and *depend upon* -- whether we've given them the "same" stuff ...

... and therefore the greater the programmers' dependency on the algorithm we use to locate modules and decide whether to go get a new copy of something or whether the one we already have will do.

(Recall: the Web is big.)

Modules starting with Java

Imagine that we *start* with Java and build a good module system ....

- What would we change in Java?
- How would we build our system?

Global mutable namespace

This is Public Enemy #1 for Java

Otherwise stated:

1. Classes self-declare their names; but
2. Clients of the classes cannot remap the names

```
package org.util; class Foo {}
```

→ org.util.Foo "used up" for [non reflective] Java

→ Lots of otherwise avoidable machinery in OSGi

Fixing the problem

Candidate solution before going any further:

1. External name locates class [file]; and
2. Importing binds external name to an identifier

```
// Direct URI reference
import "http://foo.org/Util.class" as UtilA;
// Some sort of "catalog" entry
import "util" as UtilB;
import "bar" as Bar;
```

Semantics of names

Request to *some* systems for retrieval of class stuff ...

... that's where the Bigness comes in.

Question: To what extent should we rely on the way these systems work?

Some definitions

Class/Module: Synonyms in our example

Strategy: How to find a class on the (*BIG*) Web (URIs, checksums, signatures, ...)

Short name: A string like "foo" or "org.util.Bar" that can appear in an import

Catalog: A mapping from short names to strategies

Bundle: Archived sources for classes + a catalog

Static state (singletons)

Traditional Java has static (ambiently shared) state ...

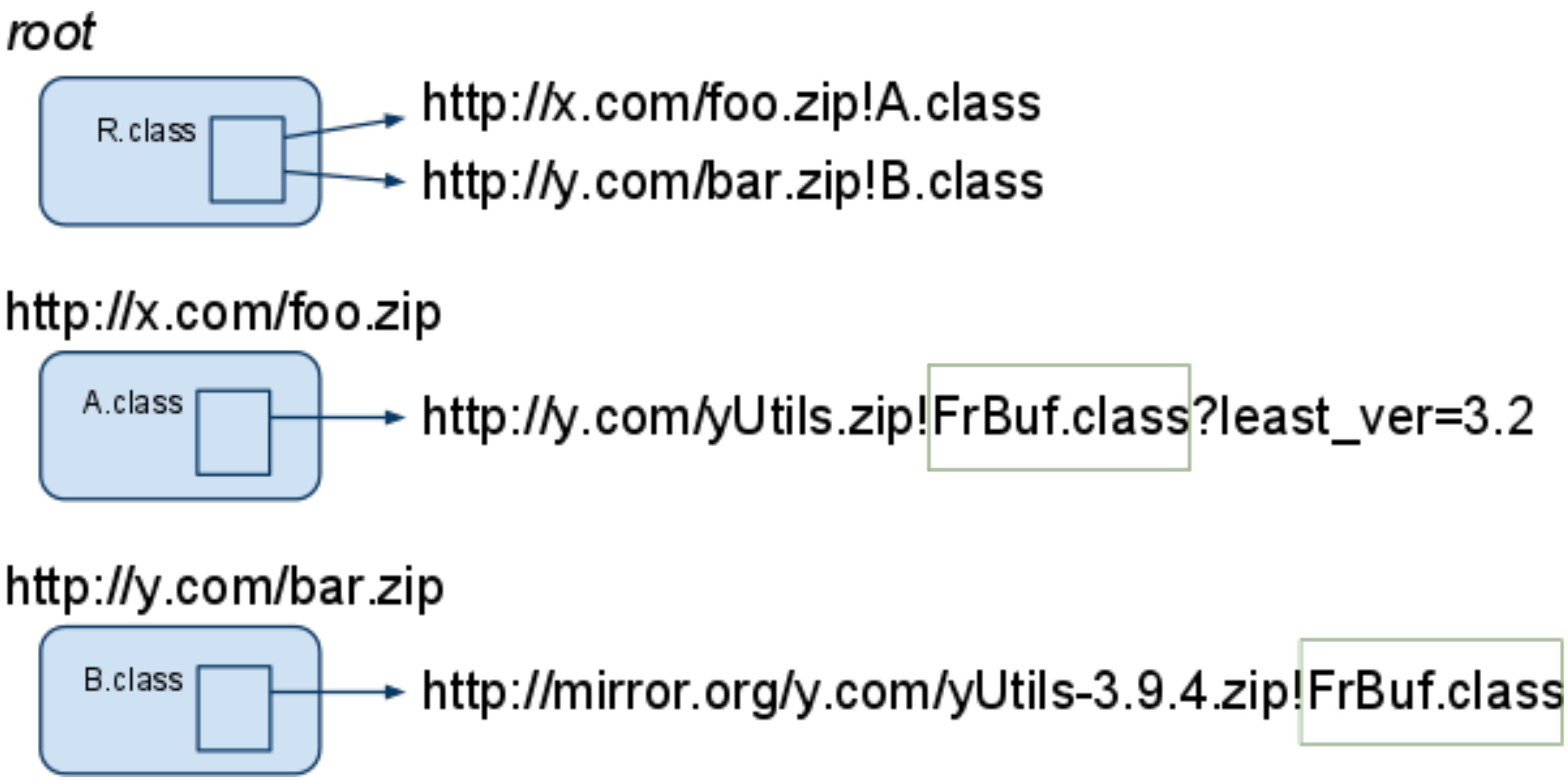
Mutable: arbitrary "application" shared state

Immutable: types, enum value

(This means Java is G3.)

It is crucially important whether two pieces of an application get the *same* static state

Hypothetical bundles



root

R.class

http://x.com/foo.zip!A.class

http://y.com/bar.zip!B.class

http://x.com/foo.zip

A.class

http://y.com/yUtils.zip!FrBuf.class?least_ver=3.2

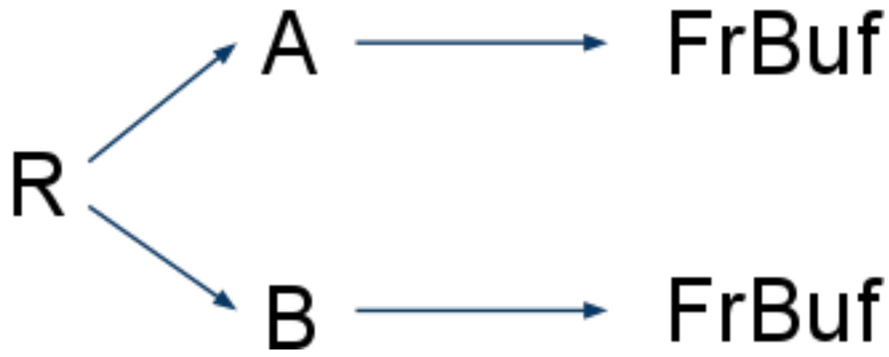
http://y.com/bar.zip

B.class

http://mirror.org/y.com/yUtils-3.9.4.zip!FrBuf.class

Class FrBuf contains mutable shared state
(e.g., shared frame buffer)

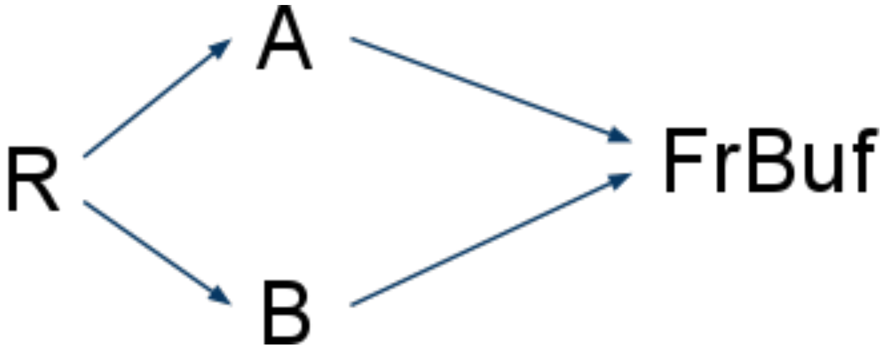
Idea 1: Separate instances



But: The programmers expected FrBuf to contain important shared state.

Why should packaging of source control the instance graph in this way?

Idea 2: Same instance

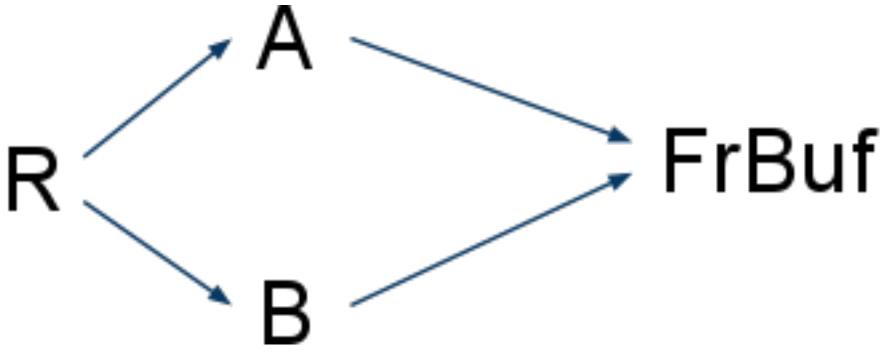


But: a minor change to the strategies in the bundle of A could suddenly cause us to revert to separate instances. Surprising.

Important shared information should not be subject to such fragility.

Idea 3: Same instance via remap

Now the bundle of R *remaps* the strategies of the bundles of A and B to *always* match.



Now the bundle of R is strongly dependent on the bundles of A and B; the author of R must always track its dependencies and do remapping work.

Conclusion

The Web is big.

Reduce the "stickiness" of dependencies (your G level).