Oracle Blog

John Rose @ Oracle

John Rose's weblog at Oracle

« JavaOne in 2010 | Main | a modest tool for... »

larval objects in the VM

By john.rose on Oct 02, 2010

Or, the ascent from squishy to crunchy.

I want to talk about initialization of immutable data structures on the JVM. But first I want to note that, in good news for Latin lovers (and those with Latin homework), Google has made Latin translation available ad stupor mundi.

That has nothing to do with Java, but it reminds me of a Java design pattern that deserves to be unmasked and recognized: The larval stage. The idea (which will be instantly familiar to experienced programmers) is that every Java data structure goes through a construction phase during which it is incomplete, and at some point becomes mature enough to publish generally. This pattern is most readily seen in Java constructors themselves. The `this` value during the execution of an object's constructor in general contains uninitialized fields. Until the constructor completes normally, the object may be regarded as in a *larval* stage, with its internal structure undergoing radical private changes. When the constructor completes, the object may be viewed, by contrast, as having entered a permanent *adult* stage. In that stage, the object is "ready for prime time" and may be exposed to untrusted code, published to other threads, etc.

This pattern is so important that we amended the 1.1 version of Java (here is an old link to my paper) to include the concept of *blank finals* to enable the adult stage of an object to be immutable. Later on, the JSR 133 Expert Group enhanced the Java memory model to guarantee that larval stages of immutable objects could not be made unintentionally visible to other threads. The result is that immutable Java objects (starting with `Integer` and `String`) can be easily defined and safely used even in massively multi-threaded systems. Especially on such systems, immutable data structures are of great importance, because they allow threads to communicate using the basic capabilities of the multi-processor memory system, without expensive synchronization operations. (This isn't the best imaginable way for processors to communicate, but it is the communication channel to which chip designers give the most attention. The Von Neumann haunting appears to be permanent.)

Something is missing, though. The support I just described applies only to objects which are able to

enter the adult stage when their constructor completes. This means that the complete information content of an object must be supplied as arguments (or some other way) to the constructor. If an immutable object's contents are built up incrementally in a variable number of optional steps, the construction of the object is better expressed using a builder pattern. In this pattern, a builder object acts as a front-end or mask for the object under construction. The builder object has an API which accepts requests to add information content to a larval object (which may or may not actually exist yet), and is eventually asked to unveil the adult object. The first instance of this pattern in Java was the trusty StringBuffer, which collects `append` requests and eventually produces the completed string in response to the `toString` request. More recently, Google has made an admirable investment in builder-based APIs for immutable collections of various sorts.

Still, from my viewpoint as a JVM tinkerer, something else is missing. It seems to me that the most natural way to express the creation of an immutable object is what I see when I read the machine code for creating an `Integer` or `String`: First you allocate a blank object, then you fill it in, then you publish it. Just before you publish it, you take whatever steps needed to make sure everybody will agree on its contents. (This is sometimes called a fence, which is a subject of multiple posts.) After you publish the object, nobody changes it ever. (Well, in some cases, maybe there's an end to the epoch. But that is another story.) The only changes ever made to that block of memory are performed by the garbage collector, until (and unless) the block gets reused for another object.

In other words, using the larval/adult metaphor for this pattern, the object building process starts out with an incomplete larval object, which is masked behind (or cocooned within) a module (wish I could say monad here) that sets up the object, and when the object is mature, eventually hatches it out into the open as an adult. The organization of the code which sets up the object is *not* in general as simple as a constructor body.

In order to make this work better in more cases, I want to give the builder object fully privileged access to the object in its larval state. And I want to these full privileges to extend to the initialization of finals, a privilege which is currently given only to constructors. The increased flexibility means that the final fields will be initialized by multiple blocks of code. The blocks of code may even repeatedly initialize finals (as the underlying array in a `StringBuffer` is subject to repeated extension or truncation). The blocks of code may be invoked by untrusted code (via the builder API, not directly). Eventually the builder declares that the object is done. Just before it publishes the object, it flushes all the writes. The flush sometimes appears as a memory fence operation in the machine code. This part is especially problematic, since the current Java and JVM specifications only guarantee correct flushing of writes for variables used to reach final fields. The guarantees for non-final fields and array elements are weaker, and the interactions are subtle.

What would this look like in an amended Java and JVM? Maybe non-public access to finals could be relaxed to allow writing, so that if a builder object has privileged access to do so, it can write the finals of a larval object. There would have to be an explicit "hatching" step, I think, to clearly mark the pre-publication point at which writing must stop and memory must be flushed. One or more new keywords could be used to indicate statically which finals are writable, which methods or classes can do the writing, and where the writing must stop. There is probably a way to express it all without keywords, too, or a combination of keywords ("volatile final", for for those looking to recoil from rebarbative syntax). The surface syntax is less important than the pattern. The pattern must prevent you from applying a larval operation to an adult operation, both in the language and in the JVM. That might be an innocent mistake, or a deliberate attack; in either case it must be provably impossible. The important thing to recognize is that there are separate larval and adult sets of operations (APIs) and only the larval ones are allowed to change finals.

But a static pattern cannot ensure such safety, unless we allow another new thing. That is some kind of type-state or changeable class, which expresses the transition from larval to adult stage. A direct and flexible way of making this distinction would be to allow a Java object to have two types over its lifetime, a larval type with an extended set of initialization operations, and an adult type. The type

change operation from larva to adult would be a low-level JVM operation which would do several things:

- mark the object permanently as adult
- forbid all future attempts to invoke methods classified as larval
- forbid all future changes to finals
- flush all pending memory changes relevant to the object

In all cases to date, the de facto larval API has privileged elements, like the package-private `String` constructor used by `StringBuffer`. Most uses of the larval type-state I am suggesting would probably continue to be restricted to the internals of a specific module. Explicit larval objects would tend to allow builder objects to be simpler, since the builder could drop information into the larval object as soon as it is known, rather than (as at present) keep shadow copies until the constructor is finally invoked.

With better protection, based on type-state, it might be reasonable in some cases to make larval APIs public. For a large-scale example, a transactional database might support a public API for creating a larval view of a new version of the database, which would allow free mutation of the database contents. The adult form would be a read-only view of some other version.

In the small scale (which is more typical), when a Java API designer creates a tuple-like class for something like complex numbers, there are always conflicting impulses between making the structure immutable, so that it can be safely used across threads, or else making the structure mutable (and often with public fields), so that the objects can be used directly as scratch variables in a computation. If the choice is made for mutability, every object must be defensively copied before it handed to untrusted code. If the choice is made for immutability, a fresh object must be made for every intermediate value of a computation. Years ago, the tilt was towards mutability, probably under the assumption that objects were expensive to allocate. This tilt might be visible in the choice to make Java arrays only mutable, and in the mutability of the `Date` class. (In the worst case there are mostly-immutable data structures, such as `java.lang.reflect.Method`, which has one mutable bit.) For `Complex`, there are mutable sketches floating around the net, but the Apache commons design is immutable. What I want to point out here is that, if we had type-state with control of mutability, programmers would get both from two stages of the same type: The larval form could have public mutable fields, useful as temporaries, while the adult form would be immutable and safe to throw around. Defensive copying would be rare to nonexistent, and failures to copy could be detected by checks on type-state.

All this begs the question of what type-state looks like in the JVM. That is a discussion for another day, but I will drop a hint, with another biological metaphor: If a *class* is a standard unit of taxonomy, what would we say if we had to suddenly distinguish objects of the same class? Well, we would have to invent a subsidary unit of taxonomy, such as the *species*. There are several potential uses of such a refined distinction: Storing the type parameters for reified generics, tracking life cycle invariants (as here with larva vs. adult), and optimizing prototype based languages. At the JVM level, all these things are an extra indirection between object and class, and could be formalized and made available to the library implementor.

A final word about terminology: The term “larva” comes from a Latin word which can mean a mask; a “larvatus” is something that is masked. Creepily, the word also refers to witches, skeletons, and (as with modern languages) worms. I suppose that if we invent larval data structures, the name will remind us to keep them well covered, at least until their little exoskeletons have hardened. More specifically, unfinished data structures should be carefully masked. Or, as an ancient Roman software engineer might have remarked, *Collectio imperfectum larvatum sit*.

Category: JVM

Tags: none

Permanent link to this entry

« JavaOne in 2010 | Main | a modest tool for... »
Comments:

Very nice proposal. The amount of boilerplate this could reduce and the encouragement to use immutable data structures a lot more than nowadays would be definitely beneficial to the language and platform.

Posted by **Igor Minar** on October 03, 2010 at 01:55 AM PDT #

This would make a difference to distributed code.

Typically classes compiled with Generics can't be mixed when they have been compiled separately, since the type casts don't match.

It potentially makes a huge difference to Serialization, by guaranteeing immutability of references and referent objects after deserialization. It might also be possible to relieve some of the issues of classes being tied to Serialized form, since the larval stage is akin to a constructor.

Posted by **Peter** on October 03, 2010 at 05:58 PM PDT #

Hmm ... would that allow to create cyclic immutable structures?

Like a tree where the immutable parent knows the immutable children, and the immutable children know their immutable parent?

That would be really great for some data structures.

Posted by **guest** on October 03, 2010 at 10:16 PM PDT #

StringBuffer no longer uses a package private String constructor, as it no longer shares the array with the constructed String.

Posted by **Bas de Bakker** on October 03, 2010 at 10:59 PM PDT #

Idea (stupid?): Mark the objects permanently as adult could be handled similar to StringBuilder, but use generics and java 7 dynamic features to require less changes to the language.

Larva<MyObject> builder = makeLarva(MyObject.class, constructor arguments...)

builder.setMyFinalMember(...)

return builder.toObject()

Larva class would have privileged access similar StringBuilder. A new keyword would be needed to mark methods as able to change final members, only accessable by Larva.

Posted by **Marcus Vesterlund** on October 04, 2010 at 07:08 PM PDT #

Maybe you want something like the Ruby freeze method,

http://ruby-doc.org/core/classes/Object.html#M000356

Fields which are freezable can be set only until the object is frozen,after which they are final. Freezing an object would also have memory model implications.

Posted by **Sean** on October 05, 2010 at 12:53 AM PDT #

Would I be, somewhat, correct when I say that this appears to allow of a way to create strictly immutable lazy values? At the cost of a megamorphic callsite that is.

At the larval state the eventual value hasn't been computed yet, and the hatch method, say, get, forces/runs the associated code, fills in the final, and the reference/thunk is now a constant, where, yet another, get simply returns the value. This get is now the adult get and not the larval get.

Posted by **Maarten Daalder** on October 05, 2010 at 01:01 AM PDT #

John, I think this change would be a HUGE welcome. I'm currently working on a banking-related application and I see so many business model classes with lots and lots of fields that simply cannot be made final because not the whole object is initialized at one time. Information about these fields is available at later stages. With a builder-like facade to these objects that has access to blank final fields could reduce a lot of boilerplate, make the classes immutable and lead to an overall cleaner design.

Posted by **kodeninja** on October 06, 2010 at 01:08 AM PDT #

Hello,

I just want to drop a note to my research on Intervals. It allows support for "temporarily mutable" objects like the larval objects you refer to, but using a rather different approach than Type State.

The basic idea is that users can denote the span of time in which the object is mutable as part of its type. Once that span of time has expired, the object can no longer be changed. One nice feature of this approach is that aliasing is no problem, as the type of the object never changes --- instead, the compiler is aware of whether a particular method occurs during the mutable span or not, and so the permitted operations change depending on when a method will execute.

Some more details are available here: http://harmonic-lang.org/news/larval-objects-using-interv.html or in our recent paper: http://www.lst.inf.ethz.ch/research/publications/SPLASH_2010.html

Posted by **Niko Matsakis** on October 21, 2010 at 03:45 AM PDT #

I have had similar ideas, from (may be) another starting point.

So, I am quite happy you are pushing your proposal.
Thank you !

1) my main starting point was about set() methods
A lot of developers use get/set methods for maximizing initilization flexibility, and then, they somewhat bypass constructor-based initialization.

It implies multiple drawbacks:
a) all these quite empty set() methods are somewhat odious
b) all other classes have access to these set() methods and then, its breaks encapsulation
c) a solution could be to centralize initialisation into a builder class, but still, these set() methods are needed

Then, imho, I thought we need a way to build more easily builder classes, with the following features:
- the builder should have full access to all fields and methods (or maybe a more restricted access driven by annotations),

- when a builder is returning an object, only a subset of fields and methods should be available. And the returned object could be immutable, if no modifying access is available.

From a *naive* point of view, it seems "all about" mainly changing the virtual object table ~ something like upcasting, without a way to get back.

Note: writing about such builder classes remind me the 'friend' keyword in C++, missing in Java. Imho, the pb we face in Java comes partly from this missing C++ feature.

2) Another point I thought is about DI frameworks.
Basically, the Spring framework do provide such builder classes, but they are hidden through dynamic scripting.

Then, I think too, a builder class proposal (~ larval objects proposal ?) could be related to frameworks like Spring. Or, to say things differently, in case of a successful larval objects propsal, if one can associate it with dynamic scripting/interceptor, good parts of Spring feature benefits could be brought in Java platform itself.

Are you planning experiments for the larval object proposal ?
What are the next steps if planned ?

Posted by **Dominique De Vito** on October 25, 2010 at 06:49 AM PDT #

Count me in as well... the amount of manual effort expended constructing and maintaining builders (especially for incrementally-initialized classes with a lot of fields) is really frustrating. Anything to get this done cleanly and interacting nicely with the JMM is welcome!

Posted by **Jonathan Anderson** on November 02, 2010 at 11:04 AM PDT #

There's a very nice feature making its way into the Clojure language, called Pods:
http://kotka.de/blog/2010/12/What_are_Pods.html

Posted by **guest** on September 16, 2011 at 08:10 AM PDT #

see also http://clojure.org/Transients

Posted by **mnicky** on September 17, 2011 at 05:11 AM PDT #

Post a Comment:

- Name: guest
- E-Mail:
- URL:
- Notify me by email of new comments
- Remember Information?
- Your Comment:
- HTML Syntax: NOT allowed

- Please answer this simple math question

1 + 78 =

- Preview Post

About

John R. Rose

Java maven, HotSpot developer, Mac user, Scheme refugee.

Once Sun and present Oracle engineer.

Search

Enter search term:

Search only this blog

Recent Posts

- two thoughts about career excellence
- value types in the vm, infant edition
- the isthmus in the VM
- value types and struct tearing
- celestial harmony
- solar eclipse backyard adventure
- Monday at Microsoft Lang.NEXT
- the OpenJDK group at Oracle is growing
- value types in the vm
- sweet numerology keyfob

Top Tags

- communityone
- invokedynamic
- javaone
- jvmlangsummit

Categories

- General
- JVM
- Java
- Personal

Archives

« December 2014

Sun	Mon	Tue	Wed	Thu	Fri	Sat
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20

21 22 23 24 25 26 27
28 29 30 31

Today

News

- CNN
- NY Times

Blogroll

- Dave Johnson
- Lance Lavandowska
- Matt Raible

Menu

- Blogs Home
- Weblog
- Login

Feeds

RSS

- All
- /General
- /JVM
- /Java
- /Personal
- Comments

Atom

- All
- /General
- /JVM
- /Java
- /Personal
- Comments

The views expressed on this blog are those of the author and do not necessarily reflect the views of Oracle. Terms of Use | Your Privacy Rights | Cookie Preferences