OpenJDK

OpenJDK FAQ
Installing
Contributing
Sponsoring
Developers' Guide

Mailing lists
IRC · Wiki

Bylaws · Census
Legal

JEP Process

search

Source code
Mercurial
Bundles (6)

Groups
(overview)
2D Graphics
Adoption
AWT
Build
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
Internationalization
JMX
Members
Networking
NetBeans Projects
Porters
Quality
Security
Serviceability
Sound
Swing
Web

Projects
(overview)
Annotations Pipeline 2.0
Audio Engine
Build Infrastructure
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Device I/O
Font Scaler
Framebuffer Toolkit
Graal
Graphics Rasterizer
HarfBuzz Integration
IcedTea
JDK 6
JDK 7
JDK 7 Updates
JDK 8 · Java SE 8
JDK 8 Updates
JDK 9
JavaDoc.Next
Jigsaw
Kulla
Lambda
Locale Enhancement
Memory Model Update
Modules
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: PowerPC/AIX
SCTP
Sumatra
ThreeTen
Type Annotations
XRender Pipeline
Valhalla

JEP 169: Value Objects

Owner	John Rose
Created	2012/10/22 20:00
Updated	2014/09/23 18:58
Type	Feature
Status	Draft
Component	hotspot
Scope	SE
Discussion	mlvm dash dev at openjdk dot java dot net
Effort	L
Duration	L
Priority	4
Issue	8046159

Summary

Provide JVM infrastructure for working with immutable and reference-free objects, in support of efficient by-value computation with non-primitive types.

Goals

- Support user-defined and library-defined abstract data types with performance profiles similar to Java primitive types.
- Bring the semantics of `int` and `java.lang.Integer` closer together.
- Allow representation of ubiquitous types currently not well supported on the JVM, including complex numbers, vector values, and tuples.
- Increase shareability of Java data structures.
- Provide clear and explicit semantics for shared read-only array data.
- Enable functional-style computation with pure data, for optimized parallel computations.
- Increase safety and security and decrease "defensive copying" in applications which must share structured data across trust boundaries.

Non-Goals

This work is not intended to change the Java Language Specification or JVM bytecode instruction set.

Motivation

In Java, primitive types are a principal factor in coding for reasonable performance. For example, programmers expect an array of `int`s to be cheaper to work with than a `List` of `Integer` objects, and they code accordingly.

In modern JVMs, object allocation is inexpensive, with a cost comparable to out-of-line procedure calling. But even this cost is often a painful overhead when compared to individual operations on primitive values. Thus, Java programmers face a binary choice between existing primitive types (which avoid allocation) and other types (which allow data abstraction and other benefits of classes). When they need to define small composite values such as complex numbers, pixels, or pairs of return values, neither approach serves. This dilemma often has no good solution, and the workarounds distort Java programs and APIs. Consider, for example, the lack of a good complex number type for those who program numeric algorithms in Java.

Unlike the days when Java was designed, modern hardware now routinely operates on values larger than 64 bits, which we may collectively call "vectors". It is difficult to work with vector values from Java because they cannot be directly represented in Java code without creating a temporary object. A mutable object (often an array) can be created to hold the components of such a vector value, but this is merely a workaround, which does not qualify as direct representation of the value, since the same object will (in general) successively hold a series of values. Alternatively, an immutable object can be created, which qualifies as a direct representation, but each new value requires creating a new object. The costs are often high enough to discourage programmers from using the direct representation.

There are optimizations which can eliminate object allocation in some regions of code. For example, objects can sometimes be "scalarized" into their component fields, if an escape analysis can be successfully carried out. However, such optimizations are limited in scope and applicability. For out-of-line calls, objects must still be boxed into memory, as long as existing Java reference semantics are enforced. Without new rules relaxing reference semantics, local escape analyses cannot be relied on to remove boxing overheads from objects, no matter how similar they are to primitive types.

We need new rules which will allow some Java objects to be routinely represented as unboxed groups of scalar values and/or vector registers, and (when that fails) to represent values directly and efficiently as immutable boxed values.

Description

A new operator `lockPermanently` will be defined which takes an `Object` and marks it as immutable and unaliasable.

The use of this operator is formally similar to `Object.clone`, except that the original object will be returned in a new locked state. Similar to `clone`, the operator will be idempotent.

It may be expedient to place limits on the subsequent use of inputs to the locking operator, although this might require changes to the verifier.

In general, a permanently locked object cannot be subjected meaningfully to any operation that depends on the reference identity of the object. An operation depends on reference identity if the operation produces different results depending on whether it applies to the original object or one of its clones. Thus:

- Fields and array elements cannot be changed.
- Synchronization cannot be performed.
- Methods for waiting or notifying cannot be invoked.
- An identity hash code cannot be queried.
- Pointer equality checks should not be performed.

Except for pointer equality checks, forbidden operations will throw some sort of exception. How to control pointer equality checks is an open question with several possible answers.

Permanent locking will apply to the wrapper types `Integer`, `Boolean`, etc. The standard primitive boxing methods `valueOf` will produce permanently locked objects.

Permanent locking will apply to all array types. An overloaded method such as `Arrays.lockedCopyOf` is likely to be supplied.

There will be restrictions on the static form of classes that support locking. (Examples: all fields must be `final` and/or a marker interface must be declared.) There will be a design pattern for value-oriented classes (like `Complex`) which support locking.

When an object is locked, the JVM is allowed to box and unbox the object much more freely. Specifically, the JVM will be free to insert virtual *reboxing* operations at any method invocation for any boxed object which is a method argument or return value. A reboxing operation potentially substitutes one reference for an equivalent one. Reboxing can produce either completely new copies of an object, or reuse an old one. The reuse can be global, and therefore JVMs are allowed (but not required) to perform interning.

Once the JVM is free to insert virtual reboxing operations, it can create customized calling sequences for methods that operate on value-oriented classes, which transmit the components of the values, delaying (and usually eliminating) boxing operations.

Note that returning a value type with multiple components (such as a tuple) is indistinguishable (at the machine code level) to a multiple-value or struct return.

Specialized value-oriented calling sequences can be confined to compiled code. The interpreter can continue to operate as it does today.

Additional design notes are kept in the MLVM repository.

Alternatives

Programmers currently must use what Rich Hickey calls place-oriented programming in mutable arrays or buffer objects, when working with composite values. They could simply continue this.

It should be emphasized that place-oriented programming in Java is an anti-pattern, because it blurs the correspondence between Java variables and application values. If a value is implicitly defined by being stored in two or more Java variables, there is no way to work with it directly as as single variable, method argument, or return value. The programmer is required to manage aliasable "places" to store the value components, as well as the values stored in them. Any optimizing compiler is hard-pressed to distinguish the two.

Local escape analysis could be extended (with heroic efforts) to interprocedural escape analysis, allowing methods to pass object fields in registers, delaying object creation perhaps indefinitely. Without new rules for permanently locked objects, the existing rules for field mutability and object identity tracking would require complex, probably unworkable, additional data to be passed between methods along with the object field values.

We could add new explicit tuple types to the JVM, providing a more explicit third option between classes and primitives.

In the case of arrays, we could introduce a new set of types for representing immutable array values, instead of providing the means for locking arrays. This would be confusing to programmers and lead to extra copying between working mutable array values and new immutable arrays.

We could also use the existing rules (rather delicate rules in the Java Memory Model) for *de facto* immutable arrays, which state that an array accessed only by a final field may be regarded as stable, and that changes are race conditions. These current rules require routine defensive copying of arrays before storing them in final fields of objects which might be shared. The current proposal is better, because it makes the stabilization step (often a memory fence) to be more explicit. It also defends more vigorously against race conditions, by throwing an exception when a write is attempted to a permanently locked array. Finally, defensive copies can be eliminated in the current proposal.

The concept of permanent locking could be applied to class (or other types) instead of individual objects. This might simplify some JVM optimizations, although in practice some classes (like `Integer`) will be 99.99% locked.

Advantages of per-instance locking are:

- A clear story about mutability during initialization.
- Simple interoperability with existing Java array types.
- Compatibility with reference-oriented use of wrappers (`new Integer(n)` vs. `Integer.valueOf(n)`).

JITs can use optimistic techniques to specialize code for value objects even when some uses are reference-oriented.

(For more discussion of mutability during initialization of read-only objects, see larval objects.)

Note: None of these alternatives seem to promise a mechanism for routinely unboxing composite values (or even `Integer` values) across method call boundaries.

Testing

- Existing tests for array operations should be adapted to test for correct enforcement of locking.
- Existing tests for object field operations (where applicable) should be adapted to test for correct enforcement of locking.
- Likewise, existing tests for synchronization, identity hashing, and pointer comparison should be adapted.
- Concurrency tests should verify that locking an object properly flushes its fields for publication.
- Performance tests should verify that loops producing sequences of composite values do not create a significant GC load.
- Beta versions of the system should be supplied to developers and end users to ensure that new restrictions on wrapper values (`Integer`, etc.) are not violated by existing applications. Workarounds may be needed.
- Sample value types should be implemented and used experimentally to validate the mechanism.