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
VisualVM
Zero

JEP 193: Enhanced Volatiles

Author	Doug Lea
Owner	Paul Sandoz
Created	2014/01/06 20:00
Updated	2014/11/19 10:25
Type	Feature
Status	Draft
Scope	SE
JSR	TBD
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	M
Duration	L
Priority	2
Reviewed by	Dave Dice, Paul Sandoz
Endorsed by	Brian Goetz
Release	9
Issue	8046183
Depends	JEP 188: Java Memory Model Update

Summary

Define a solution to invoke the equivalents of `java.util.concurrent.atomic` or `sun.misc.Unsafe` operations on object fields and array elements.

Goals

The following are required goals:

- Safety. It must not be possible to place the Java Virtual Machine in a corrupt memory state. For example, a field of an object can only be updated with instances that are castable to the field type, or an array element can only be accessed within an array if the array index is within the array bounds.
- Integrity. Access to a field of an object follows the same access rules as with `getfield` and `putfield` byte codes in addition to the constraint that a `final` field of an object cannot be updated. (Note: such safety and integrity rules also apply to `MethodHandles` giving read or write access to a field.)
- Performance. The performance characteristics must be the same as or similar to equivalent `sun.misc.Unsafe` operations (specifically, generated assembler code should be almost identical modulo certain safety checks that cannot be folded away).
- Usability. The API must be better than the `sun.misc.Unsafe` API.

It is desirable, but not required, goal that the API be as good as the `java.util.concurrent.atomic` API.

Motivation

As concurrent and parallel programming in Java continue to expand, programmers are increasingly frustrated by not being able to use Java constructions for arranging atomic or ordered operations for the fields of individual classes; for example atomically incrementing a `count` field. Until now the only ways to achieve these effects were to use a stand-alone `AtomicInteger` (adding both space overhead and additional concurrency issues to manage indirection) or, in some situations, to use atomic `FieldUpdaters` (often encountering more overhead than the operation itself), or to use JVM Unsafe intrinsics. Because intrinsics are preferable on performance grounds, their use has been increasingly common, to the detriment of safety and portability. Without this JEP, these problems are expected to become worse as atomic APIs expand to cover additional access consistency policies (aligned with the recent C++11 memory model) as part of Java Memory Model revisions.

Description

This proposal focuses on the control of atomicity and ordering for single variables. We expect the resulting specifications to be amenable for extension in natural ways for additional primitive-like value types or additional array-like types, if they are ever defined for Java. However, it is not a general-purpose transaction mechanism for controlling accesses and updates to multiple variables. Alternative forms for expressing and implementing such constructions may be explored in the course of this JEP, and may be the subject of further JEPs.

The target solution may require library enhancement, hotspot enhancements and compiler support. Language syntax enhancements may also be considered, especially if they enhance compile-time type checking and are complementary with existing syntax.

We model the operations on reference and primitive types via an abstract class called `VarHandle` that contains one signature-polymorphic-like method for each operation. Such a single class can support access to static fields, instance fields and array elements for both reference types and primitive types. As such `VarHandle`s can be considered "`MethodHandle`s for data".

A possible `VarHandle` class is:

```
public abstract class VarHandle {
    public abstract Object get(Object... args);
    public abstract Object getRelaxed(Object... args);
    public abstract Object getAcquire(Object... args);
    public abstract Object getSequential(Object... args);

    public abstract void set(Object... args);
    public abstract void setRelaxed(Object... args);
    public abstract void setRelease(Object... args);
    public abstract void setSequential(Object... args);

    public abstract Object getAndSet(Object... args);
    public abstract boolean compareAndSet(Object... args);
    public abstract boolean compareAndSetAcquire(Object... args);
    public abstract boolean compareAndSetRelease(Object... args);
    public abstract boolean weakCompareAndSet(Object... args);
    public abstract boolean weakCompareAndSetAcquire(Object... args);
    public abstract boolean weakCompareAndSetRelease(Object... args);

    public abstract Object getAndAdd(Object... args);
    public abstract Object addAndGet(Object... args);
    public abstract Object getAndIncrement(Object... args);
    public abstract Object incrementAndGet(Object... args);
    public abstract Object getAndDecrement(Object... args);
    public abstract Object decrementAndGet(Object... args);
}
```

A instance of `VarHandle` covering a field or array element is obtained by using equivalent lookup mechanisms, governing access control, as for looking up a `MethodHandle` to a field of a class or array elements of an array.

Such lookup to obtain a `VarHandle` covering a field named `i` of type `int` on a receiver class Foo might be performed as follows:

```
class Foo {
    int i;

    ...
}

...

class Bar {
    static final VarHandle VH_FOO_FIELD_I;

    static {
        try {
            VH_FOO_FIELD_I = VarHandles.lookup().
                in(Foo.class).
                findField(Foo.class, "i", int.class);
        } catch (Exception e) {
            throw new Error(e);
        }
    }
}
```

The number of arguments, the argument types and return type of a 'VarHandle' instance are governed by what variable the `VarHandle` covers and what method is being invoked.

For example, a `compareAndSet` on the previously looked up `fooi` handle requires 3 arguments, an instance of receiver Foo and two ints for the expected and actual values:

```
Foo f = ...
boolean r = VH_FOO_FIELD_I.compareAndSet(f, 0, 1);
```

In contrast a `getAndSet` requires 2 arguments, an instance of receiver Foo and one int that is the value to be set:

```
int o = (int) VH_FOO_FIELD_I.getAndSet(f, 2);
```

Access to array elements will require an additional argument, of type `int`, occurring between the receiver and value arguments (if any) that corresponds to the array index of the element to be operated on.

It is important to optimize the `VarHandle` method invocation call-sites to avoid the overheads of boxing primitive arguments and packing all arguments into an array. Additionally, it is also important to guarantee that the hotspot runtime (C2) compiler will inline those invocations and produce efficient generated machine code when appropriate conditions are met. (One such condition is the `VarHandle` instance should be a constant, as when held in a static final field, so constant folding will occur, such as folding away method signature checks and/or receiver/value argument cast checks.)

Two possible solutions to avoid boxing and packing are:

1. The `VarHandle` methods are declared to be signature polymorphic with minor enhancements on signature polymorphism to support non-polymorphic return types; or
2. The java compiler compiles `VarHandle` methods invocations into `invokedynamic` instructions, producing the same signature polymorphic method signatures as in 1), but crucially having equivalent semantics as ordinarily generated `invokevirtual` instructions.

(Note: the former is essentially a form of hard-coded `invokedynamic` which is currently specific to `MethodHandles` and is leveraged by `invokedynamic`. The properties of either should be roughly equivalent in terms of optimization.)

Updates to the Java Virtual Machine Specification and the Java Language Specification (JLS) may be required to specify refinements to signature polymorphism and the addition of new signature polymorphic methods.

A disadvantage of the explicit lookup of a `VarHandle` and leveraging "varargs" signature polymorphic methods is the lack of static type checking to catch errors at compile time rather than such type checking errors produced at runtime. Two possible improvements are further described.

The first improvement is to introduce, perhaps in conjunction with `VarHandle`, two generic classes, `FieldHandle<R, V>` and `ArrayHandle<A, I, V>`:

```
public abstract class FieldHandle<R, V> {
    ...
    public abstract V getAndSet(R r, V v);
    public abstract boolean compareAndSet(R r, V e, V a);
    ...
}

class ArrayHandle<R, I, V> {
    ...
    public abstract V getAndSet(R r, I i, V v);
    public abstract boolean compareAndSet(R r, I i, V e, V a);
    ...
}
```

Constrained method signature polymorphism is applied to arguments associated with type variables that can be resolved to boxed types, such as `Integer`. This approach requires further careful analysis to ascertain if it is applicable in the short term while not causing issues in the long term if/when generics-over-primitives JEP218 and value types are introduced. Furthermore, although out of scope for this JEP, usages of `VarHandle`, `FieldHandle` or `ArrayHandle` in polymorphic or generic code should be anticipated.

A `FieldHandle<R, V>` instance can cover both instance and static fields. If the latter then the receiver argument is ignored and may be `null` (but would still be verified).

An `ArrayHandle<A, I, V>` may cover various array representations since array type and the index type is generic (the latter declared as type variable `I`). For existing Java arrays `I` will correspond to `Integer` and `int` indexes can be used. For alternative array representations (perhaps an off-heap pointer or an enhancement to existing Java arrays) `I` may correspond to `Long` and `long` index values. Further investigation will be required for multi-index configurations and potential integration with Arrays 2.0 (Note: `I` could correspond to an `int[]` array or, more preferably when value types are available, a kind of tuple type.)

The second improvement is a language syntax enhancement, leveraging the syntax for method references to look up a `VarHandle` (and a `FieldHandle` and `ArrayHandle`). For example, the previous lookup of a field 'i' on receiver class `Foo` can be expressed as follows:

```
static final VarHandle VH_FOO_FIELD_I = Foo::i;
```

(Note: this same syntax may also be used to lookup a `java.lang.Field` or a `MethodHandle` since the target type defines what field representation to look up.)

Further investigation is required to determine the compilation strategy. The Java compiler might compile this to the explicit lookup method calls (and manage possible exceptions). Alternatively this could be compiled to an `invokedynamic` instruction where it may be possible to better guarantee the return of a constant instance (in combination with a `final` local variable).

Alternatives

We considered instead introducing new forms of "value type" that support volatile operations. However, this would be inconsistent with properties of other types, and would also require more effort for programmers to use. We also considered expanding reliance on `java.util.concurrent.atomic` `FieldUpdaters`, but their dynamic overhead and usage limitations make them unsuitable.

Syntax enhancements were considered in a previous version of this JEP but were deemed too "magical" with the overloaded use of the `volatile` keyword scoping to floating interfaces, one for references and one for each supported primitive type.

Several other alternatives (including those based on field references) have been raised and dismissed as unworkable on syntactic, efficiency, and/or usability grounds over the many years that these issues have been discussed.

Risks and Assumptions

A prototype implementation of `VarHandle` has been performance tested with nano-benchmarks and fork join benchmarks, where the fork join library's use of `sun.misc.Unsafe` was replaced with `VarHandle`. So far no major performance issues have been observed and the identified hotspot compiler issues do not seem onerous (folding cast checks and strength reducing array bounds checks). So we are confident of the feasibility. However, we expect that it will require more experimentation to ensure the compilation techniques are reliable in the performance-critical contexts where these constructs are most often needed.

Impact

The classes in `java.util.concurrent` (and other areas identified in the JDK) will be migrated from `sun.misc.Unsafe` to `VarHandle`.