

Safety Not Guaranteed:

`sun.misc.Unsafe` and the
quest for safe alternatives

Paul Sandoz
Oracle
[@PaulSandoz](#)



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

WANTED: Somebody to go back in time with me. This is not a joke. P.O. Box 322, Oakview, CA 93022. You'll get paid after we get back. Must bring your own weapons. Safety not guaranteed. I have only done this once before.

http://en.wikipedia.org/wiki/Backwoods_Home_Magazine

<http://www.backwoodshome.com/articles2/silveira125.html>

Unsafe vs. Java culture

- Java developers rely on safety
 - https://blogs.oracle.com/jrose/entry/the_isthmus_in_the_vm#culture
- Understatement that this is important
 - You are not supposed to worry about SEGV or memory corruption!
- Safety enables an ecosystem of libraries



`sun.misc.Unsafe`

- Internal class within the JDK
- Used inside and unfortunately outside of the JDK
- Never intended for outside use!
- Used by many popular libraries/applications



Functionality

- Peek and poke at memory both on and off heap
- Define, instantiate and check initialization of classes
- Park/unpark threads
- Memory fences
- Some other minor stuff...



Unsafe is a sharp tool

- Even advanced developers can get things wrong or assumptions change
 - Perhaps some developers don't care
- Many other pitfalls
 - Especially security and serialization related



Is this code safe?

src/java/org/apache/cassandra/utils/FastByteComparisons.java

```
182    int minLength = Math.min(length1, length2);  
183    int minWords = minLength / Longs.BYTES;  
184    int offset1Adj = offset1 + BYTE_ARRAY_BASE_OFFSET;  
185    int offset2Adj = offset2 + BYTE_ARRAY_BASE_OFFSET;  
186  
187    /*  
188     * Compare 8 bytes at a time. Benchmarking shows comparing 8 bytes at a  
189     * time is no slower than comparing 4 bytes at a time even on 32-bit.  
190     * On the other hand, it is substantially faster on 64-bit.  
191     */  
192    for (int i = 0; i < minWords * Longs.BYTES; i += Longs.BYTES) {  
193        long lw = theUnsafe.getLong(buffer1, offset1Adj + (long) i);  
194        long rw = theUnsafe.getLong(buffer2, offset2Adj + (long) i);  
195        long diff = lw ^ rw;
```



Answer

- It is now but was not before Feb 2014
 - Can cause JVM crash on Solaris/Sparc
 - Bug has been present for over 2 years
 - Introduced in 1.1.3 fixed in 2.0.5
- Bug: CASSANDRA-6628
- Code diff



What about this?

<https://github.com/airlift/slice/blob/master/src/main/java/io/airlift/slice/Slice.java>

```
public long getLong(int index)
{
    checkIndexLength(index, SIZE_OF_LONG);
    return unsafe.getLong(base, address + index);
}
```



Answer

- No, does not account for alignment and native byte order
- Pull request created Oct 2013

-  **electrum** added a note on 4 Dec 2013 Owner
I think we should just say "Slice only supports little endian machines"

What about this?

<http://hg.openjdk.java.net/jdk9/dev/jdk/file/tip/src/share/classes/java/util/concurrent/atomic/AtomicLong.java>

```
public class AtomicLong extends Number implements java.io.Serializable {
    ...
    private volatile long value;
    ...
    public final void set(long newValue) {
        value = newValue;
    }
    ...
    public final boolean compareAndSet(long expect, long update) {
        return unsafe.compareAndSwapLong(this, valueOffset, expect, update);
    }
    ...
}
```



Answer

- Yes, on all mainline OpenJDK platforms
- No, on a platform not supporting compare-and-set of 64 bit values
- Issue JDK-8044616
- Compare-and-set requires a lock
- Cannot enforce atomicity w.r.t direct stores



The situation

- We would like to stop outside use of **Unsafe**
- Undermines the safety guarantees offered by Java
- Project Jigsaw will enforce this



Why is Unsafe in the JDK?

- For low-level VM-based features
 - Often intrinsified by HotSpot
 - JNI is PITA! (Project Panama to the rescue)
- For stuff not directly expressible in byte-code
- The glue between the the J and the VM



Future Between J and the VM?

- User-level profiling experiments for Lambda Forms

<http://cr.openjdk.java.net/~vlivanov/profiling/>

- Shrinking an array without copying

```
T[] t = ...  
T[] ct = Unsafe.chopArray(t, t.length / 2);  
t = null; // loose reference to t
```



Developer distinctions

- 99% of Java developers don't use, and have probably never heard of, and have no need to use, **Unsafe**
- 1% of Java developers use **Unsafe**
 - Often write widely-used libraries
 - Therefore much of that 99% transitively use **Unsafe** too without knowing it

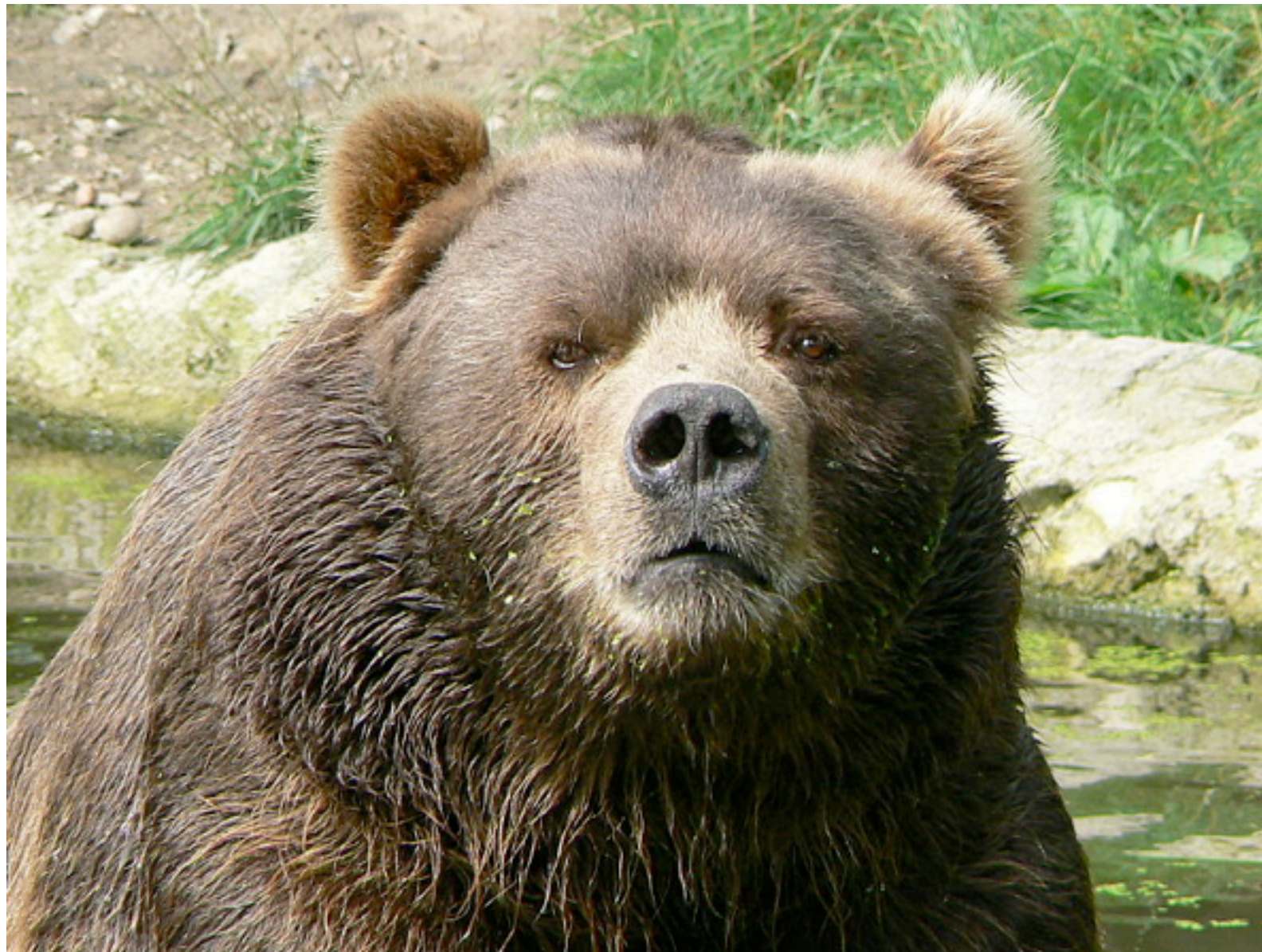


The 1% mix

- Developers who think they need to use it but probably don't really need to
- Advanced developers with a genuine need and don't care about cross platform safety
- Advanced developers with a genuine need and align with Java's culture
- Developers working on the JDK itself



Poke the bear



http://en.wikipedia.org/wiki/Wikipedia:Don't_poke_the_bear#mediaviewer/File:Male_kodiak_bear_face.JPG



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Poke!

Focus on

- ~~Developers who think they need to use it but probably don't~~
- ~~Advanced developers with a genuine need and don't care about cross platform safety~~
- Advanced developers with a genuine need and align with Java's culture
- Developers working on the JDK itself



Advanced Unsafe Developers

- Pushing the boundaries of where and how the JVM is used
- Often vocal & innovative
- Develop popular libraries
- Good for the Java community



The Unsafe survey

6. Do you have an optional dependency on Unsafe to ensure code is portable across multiple Java platforms?

Yes



**Response
Percent**

**Response
Count**

30.4%

96

No





69.6%

220



The Unsafe survey

7. If there was a "safe unsafe" standard (cross-platform) alternative for your use-cases (perhaps a new API, perhaps language changes, or both) would you be prepared to replace Unsafe with that alternative? If so under what conditions?

		Response Percent	Response Count
Yes (perhaps with conditions, if so please state those conditions)		88.6%	280
No		11.4%	36



Use-cases

- When you can peek and poke at memory the use-cases are many and varied
- Use-cases come down to two things
 - Performance; and/or
 - Work around JDK restrictions/limitations



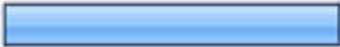






Characterizing use-cases

- A number of vertical use-cases
- Each of which can be tackled independently
 - Some sooner than others
- Wean developers off **Unsafe** step-by-step

The Unsafe survey

4. What reasons did you use Unsafe for?

		Response Percent	Response Count
Atomic access to fields and array elements (such as compare-and-swap)		44.1%	149
Off-heap memory operations (such as to emulate structures or packed objects)		63.6%	215
Deserialization hacks		36.4%	123
Fencing (to constrain re-ordering of memory operations)		22.5%	76
Access to private fields of another class		25.1%	85
Array access without bounds checks		32.5%	110
Other (please specify)		22.2%	75

Three general cases

- Off-heap
- Enhanced atomic access & fences
- De/Serialization



Four off-heap use-cases

- Reduce GC
- Efficient memory layout
- Very large collections
- Communicate across JVM boundary



Use-case to Feature

Use-case	Feature
Enhanced atomic access	<u>JEP 193 Enhanced Volatiles</u>
De/Serialization	<u>JEP 187 Serialization 2.0</u>
Reduce GC	<u>Value types</u> <u>JEP 189 Shenandoah: Low Pause GC</u>
Efficient memory layout	<u>Value types</u> , <u>Arrays 2.0</u> & <u>Layouts</u>
Very Large Collections	<u>Value types</u> , <u>Arrays 2.0</u> & <u>Layouts</u>
Communicate across JVM boundary	<u>Project Panama</u> & <u>JEP 191 FFI</u>



A Strategy

- Put features in **Unsafe**; use them in JDK
- Identify features that really belong in the Java programming model
- Add support for those features
- (Theoretically) garbage-collect them from **Unsafe**



Carrot and Stick

- Removing key functionality can hurt
- We need to make it easier for library/framework developers to migrate gradually
- From release N with **Unsafe** to release N+1 without **Unsafe**



JEP 193

Enhanced Volatiles

- **Safe, performant**, enhanced atomic access to field and array elements
- Without the dynamic overhead of **`Atomic* {Updater|Array}`** classes
- Depends on JEP 188 Java Memory Model Update



JEP 193

Success Metrics

- API should be at least as good as **Unsafe**
- Performance results should be close to **Unsafe**, and faster than **Atomic*** classes
- Good enough to replace **Unsafe** usages in **java.util.concurrent** classes



Strawman Proposal

- Enable access to corresponding methods for fields using the `.volatile` prefix

```
class Usage {  
    volatile int count;  
  
    int incrementCount() {  
        return count.volatile.  
            incrementAndGet();  
    }  
}
```

Strawman Proposal

- The `.volatile` prefix introduces scope for atomic ops on “L-values”
- Scoped to interfaces with methods, `VolatileRef`, `VolatileInt` etc.
- Requires language changes



Strawman Proposal

- What should be the implementation?
- Would it be sufficient on its own?
- Is there a way do this without such language changes?



Alternative Proposals

1. MethodHandles

2. VarHandles



MethodHandle

- A **MethodHandle** is a reference to an underlying method, constructor, or field
- Supports volatile access to a field
- Invocations inline surprisingly well
- With some tweaks can support other forms of access



Code in ForkJoinPool

```
final ForkJoinTask<?> poll() {
    ForkJoinTask<?>[] a; int b; ForkJoinTask<?> t;
    while ((b = base) - top < 0 && (a = array) != null) {
        int j = (((a.length - 1) & b) << ASHIFT) + ABASE;
        t = (ForkJoinTask<?>)U.getObjectVolatile(a, j);
        if (t != null) {
            if (U.compareAndSwapObject(a, j, t, null)) {
                U.putOrderedInt(this, QBASE, b + 1);
                return t;
            }
        }
        else if (base == b) {
            if (b + 1 == top)
                break;
            Thread.yield(); // wait for lagging update (very rare)
        }
    }
    return null;
}
```





Code in ForkJoinPool

```
final ForkJoinTask<?> poll() throws Throwable {
    ForkJoinTask<?>[] a; int b; ForkJoinTask<?> t;
    while ((b = base) - top < 0 && (a = array) != null) {
        int j = (a.length - 1) & b;
        t = (ForkJoinTask<?>)ABASE_getVolatile.invokeExact(a, j);
        if (t != null) {
            if ((boolean) ABASE_compareAndSet.invokeExact(
                a, j, t, (ForkJoinTask) null)) {
                QBASE_setRelease.invokeExact(this, b + 1);
                return t;
            }
        }
        else if (base == b) {
            if (b + 1 == top)
                break;
            Thread.yield(); // wait for lagging update (very rare)
        }
    }
    return null;
}
```



MethodHandle: Success Metrics

-  API should be at least as good as **Unsafe**, preferably better
-  Performance results should be close to **Unsafe**, and faster than **Atomic*** classes

VarHandle

- Abstraction of **safe** access to a memory location
- Leveraging **MethodHandle** invoke intrinsics and **sun.misc.Unsafe**
 - “MethodHandles for data”
- Video and slides from JVMLS talk



VarHandle:

One abstract class

- Many *access-modes*, one per method
 - Fenced, atomic (CAS)
- Many *access-kinds*, one per instance
 - Static/instance field, array, off-heap
- Many *value-kinds*, one per instance
 - Object ref, primitive, “composite” value



Code in ForkJoinPool

```
final ForkJoinTask<?> poll() {
    ForkJoinTask<?>[] a; int b; ForkJoinTask<?> t;
    while ((b = base) - top < 0 && (a = array) != null) {
        int j = (a.length - 1) & b;
        t = (ForkJoinTask<?>)ABASE.getVolatile(a, j);
        if (t != null) {
            if (ABASE.compareAndSet(a, j, t, (ForkJoinTask) null)) {
                QBASE.setRelease(this, b + 1);
                return t;
            }
        }
        else if (base == b) {
            if (b + 1 == top)
                break;
            Thread.yield(); // wait for lagging update (very rare)
        }
    }
    return null;
}
```



VarHandles: Success Metrics

- ✓ API should be at least as good as **Unsafe**, preferably better
- ✓ Performance results should be close to **Unsafe**, and faster than **Atomic*** classes

Low Hanging Fruit

- The methods `monitorEnter/monitorExit/tryMonitorEnter` can be removed
 - No one is using them
- Add a lexicographical `byte[]` comparator to `Arrays` or `ByteBuffer`
- Cassandra, Guava, ...



In Safety...

- `sun.misc.Unsafe` is going away
 - Accessible only from within the JDK
- Safe supported features planned that replace common unsafe use
- Preserving Java's culture



In Legal Safety...

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.



