What to do about sun.misc.Unsafe

15 June 2015
Greg Luck, Hazelcast JCP EC representative
Chris Engelbert, Hazelcast JCP EC representative
Martijn Verburg, LJC JCP EC representative
Ben Evans, LJC JCP EC representative
Gil Tene, Azul Systems EC representative
Peter Lawrey, Higher Frequency Trading, Java Champion
Rafael Winterhalter, Bouvet ASA

Contents

Summary

- Unsafe has become a de facto but not an official standard

- Oracle's move to clarify this is welcome if it is constructive and not destructive
- The community feels very strongly that there is an upgrade path missing
- We would like to open the debate about how that path should be defined

Current Problems

Widespread Community use of sun.misc.Unsafe - a proprietary API

sun.misc.Unsafe has wide traction in common Java frameworks and applications. Most applications, at least indirectly, depend on some library that uses Unsafe to speed up one thing or another.

In fact, even standard libraries such as java.util.concurrent depend upon pieces of Unsafe (such as park and CAS operations) for which there is no realistic alternative.

Over time, Unsafe has becoming a "dumping ground" for non-standard, yet necessary, methods for the platform, with useful methods that are relatively safe in experienced hands (such as the CAS operations) being lumped in with low-level methods that are of no real use to library developers.

Pending removal in Java 9 with Modularisation

Access to sun.misc.Unsafe is currently slated to be prevented in the upcoming Java 9 release as part of Project Jigsaw. This will break frameworks that do not (or cannot) offer a sufficient fallback and will still harm frameworks that do provide a fallback implementation, as the primary reason for adopting Unsafe in the first place is usually performance.

While additional JNI libraries could provide the same functionality as Unsafe, such libraries would need to provide 32-bit and 64-bit as well as Windows and Linux variations. This is less safe than the Unsafe class in Java 8, or a potential replacement in Java 9, as each framework would have to offer it's own implementation. To achieve comparable performance, more functionality would need to be migrated into C. e.g. an operation to read or write a String in UTF-8 format to/from native memory can be written in Java currently and achieve near C speeds, but without the intrinsics available in Unsafe, such an operation would have to be written in JNI to avoid crossing the JNI barrier too many times.

Lack of Maintenance of Bugs in Unsafe

Mark Reinhold advised Oracle will not fix bugs reported against Unsafe.

Hazelcast reported a JIT bug that was manifesting as an Unsafe bug starting with Java 8 build 40.

“Sorry, but as you know sun.misc.Unsafe is not just unsafe but it's not supported for use outside of the JDK.“

--- Mark Reinhold, 10 June 2015

Missing Cross-Vendor Specification

The current sun.misc.Unsafe class is not specified. Content changes from version to version and vendor to vendor. Cross-JVM implementations need to check for a lot of circumstances to make sure the Unsafe based implementation works on most JVMs.

Uses of Unsafe

- Low, very predictable latencies (low GC overhead)
- Fast de-/serialization
- Thread safe 64-bit sized native memory access (for example offheap)
- Atomic memory operations
- Efficient object / memory layouts
- Fast field / memory access
- Custom memory fences
- Fast interaction with native code
- Multi-operating system replacement for JNI.
- “Type hijacking” of classes for type-safe APIs without calling a constructor.

Examples of projects/products using Unsafe

- MapDB
- Netty
- Hazelcast
- Cassandra
- Mockito / EasyMock / JMock / PowerMock
- Scala Specs
- Spock
- Robolectric
- Grails
- Neo4j
- Apache Kafka
- Apache Wink
- Apache Storm
- Apache Continuum
- Zookeeper

- Dropwizard
- Metrics (AOP)
- Kryo
- Byte Buddy
- Hibernate
- Liquibase
- Spring Framework
- Ehcache (sizeof)
- OrientDB
- Chronicles (OpenHFT)
- Apache hadoop, Apache HBase (hadoop based database)
- GWT
- Disruptor
- jRuby
- Real Logic Argona
- ...

Withdrawn JEPs

There is a lack of transparency over what has happened here. Much of the community think wrongly there are JEPS to deal with these issues, but in fact some of the JEPs have been cancelled.

Possible Replacements for some aspects (mentioned in the past)

Green = Already available / Will be available in Java 9
Orange = May be available in Java 9
Red = Unlikely to be available in Java 9

Proposal	Expected in Java 9
VarHandle (no JEP)	no
Project Panama (JFFI, JEP 191)	maybe
Serialization 2.0 (JEP 187)	no (JEP disappeared - wayback machine)
ValueTypes (no JEP)	no
Enhanced Volatiles (JEP 193)	maybe
Arrays 2.0 (no JEP)	maybe

Variable Object Layout (no JEP)	no
Byte Buffers	available today (but missing 64 bit and atomic operations)
Extending Field / Array reflection access	not yet discussed

What is Needed

An open, transparent process for planning the retirement of Unsafe

Ideally, the retirement of Unsafe should be governed by a JEP within OpenJDK, with a JSR to cover the standardisation of the “good / safe pieces of Unsafe”.

A mapping of Unsafe Features to safe, stable alternatives in Java 9

In OpenJDK 7 sun.misc.Unsafe consisted of 105 methods. These subdivide into a few groups of important methods for manipulating various entities. Here are some of the main groupings:

Off-heap memory access is the number one used feature followed by Memory Information.

Green = Full Replacement
Orange = Possible Replacement (partly replacing the functionality)
Red = None

Feature	**sun.misc.Unsafe**	**Usage** Google Search Results[$]	**Java 9 replacement, if any**
Memory Information	addressSize pageSize	17,700 65,800	
Objects	allocateInstance objectFieldOffset	5,290 2,820	Reflection (Field), JEP 193?
Classes	staticFieldOffset defineClass defineAnonymousClass ensureClassInitialized	2,820 11,400 2,350 2,760	
Arrays	arrayBaseOffset arrayIndexScale	1,560 4,960	Reflection (Array), Enhanced Volatiles -

			JEP 193?
Synchronization	monitorEnter tryMonitorEnter monitorExit park unpark	4,680 2,360 14,700 N/A 13,200	Existing Java syntax and libraries. park / unpark by using LockSupport

"Safe Unsafe" On-heap Object access
 	
Note: All other access operations (e.g. getX/putX with address argument) are currently invalid (as in "will cause random heap corruption") for on-heap object access
	Unordered field access:
	  getX(Object o, ...)
	  putX(Object o, ...)
	Volatile/ordered field access:
	  getXVolatile
	  putXVolatile
	  putOrderedX
	Atomics:
	  compareAndSwapX
	  getAndAddX
	  getAndSetX
	Fences:
	  storeFence
	  readFence
	  fullFence
	copyMemory(Object src, …, Object dst, ...)
	setMemory(Object o, …)
		
		26.300 (object)
		5.420 (object)
		
		3.350 (object)
		3.110 (object)
		4.030 (int)
		
		3.800 (int)
		1.010 (int)
		290 (int)
		
		1.820
		202
		1.900
		19.400
		
		19.900
			Reflection (Field, Array).
			Enhanced Volatiles - JEP 193?
			VarHandle?, Arrays 2.0?, Variable Object Layout,
			A fences API JEP?

Off-heap Memory access	allocateMemory
	freeMemory
	copyMemory
	setMemory
	getAddress
	Unordered field access:
	  getX(long address, …)
	  putX(long address, …)
	Volatile/ordered field access:
	  getXVolatile(0, …)
	  putXVolatile(0, …)
	  putOrderedX(0, …)
	Atomics:
	  compareAndSwapX(0, ...)
	  getAndAddX(0, …)
	  getAndSetX(0, …)
	Fences:
	  storeFence
	  readFence
	  fullFence
		39,200
		122,000
		19.400
		19.900
		10.600
		
		26.300 (object)
		5.420 (object)
		
		3.350 (object)
		3.110 (object)
		4.030 (int)
		
		3.800 (int)
		1.010 (int)
		290 (int)
		
		1.820
		202
		1.900
			Mapped Byte Buffers (Speed-enhanced mapped buffer? e.g. ones with a constant limit that can allow compilers to avoid range checks in loops)
			Variable Object Layout,
			A fences API JEP?

[$] An indicator of popularity

Fields in sun.misc.Unsafe

	INVALID_FIELD_OFFSET	This constant differs from all results that will ever be returned from #staticFieldOffset , #objectFieldOffset , or #arrayBaseOffset .
	ARRAY_BOOLEAN_BASE_OFFSET	The value of {@code arrayBaseOffset(boolean[].class)}
	ARRAY_BYTE_BASE_OFFSET	The value of {@code arrayBaseOffset(byte[].class)}
	ARRAY_SHORT_BASE_OFFSET	The value of {@code arrayBaseOffset(short[].class)}
	ARRAY_CHAR_BASE_OFFSET	The value of {@code arrayBaseOffset(char[].class)}
	ARRAY_INT_BASE_OFFSET	The value of {@code arrayBaseOffset(int[].class)}
	ARRAY_LONG_BASE_OFFSET	The value of {@code arrayBaseOffset(long[].class)}
	ARRAY_FLOAT_BASE_OFFSET	The value of {@code arrayBaseOffset(float[].class)}
	ARRAY_DOUBLE_BASE_OFFSET	The value of {@code arrayBaseOffset(double[].class)}
	ARRAY_OBJECT_BASE_OFFSET	The value of {@code arrayBaseOffset(Object[].class)}
	ARRAY_BOOLEAN_INDEX_SCALE	The value of {@code arrayIndexScale(boolean[].class)}
	ARRAY_BYTE_INDEX_SCALE	The value of {@code arrayIndexScale(byte[].class)}
	ARRAY_SHORT_INDEX_SCALE	The value of {@code arrayIndexScale(short[].class)}
	ARRAY_CHAR_INDEX_SCALE	The value of {@code arrayIndexScale(char[].class)}
	ARRAY_INT_INDEX_SCALE	The value of {@code arrayIndexScale(int[].class)}
	ARRAY_LONG_INDEX_SCALE	The value of {@code arrayIndexScale(long[].class)}

	ARRAY_FLOAT_INDEX_SCALE	The value of {@code arrayIndexScale(float[].class)}
	ARRAY_DOUBLE_INDEX_SCALE	The value of {@code arrayIndexScale(double[].class)}
	ARRAY_OBJECT_INDEX_SCALE	The value of {@code arrayIndexScale(Object[].class)}
	ADDRESS_SIZE	The value of {@code addressSize()}

JSRs and JEPS

JEP1 states:

This process does not in any way supplant the Java Community Process. The JCP remains the governing body for all standard Java SE APIs and related interfaces. If a proposal accepted into this process ***intends to revise existing standard interfaces, or to define new ones, then a parallel effort to design, review, and approve those changes must be undertaken in the JCP****, either as part of a Maintenance Review of an existing JSR or in the context of a new JSR.*

Because Unsafe is not part of a standard interface, it can be removed without a JSR. A JSR is required for Java 9, but the only way to reject the changes to Unsafe is to vote down the forthcoming Java 9 JSR.

The JEP Process is not at all representative. From JEP1:

The **OpenJDK Lead ultimately decides** which JEPs to accept for inclusion into the Roadmap.

The **OpenJDK Lead can move a proposal forward from Submitted to Candidate.**

Mark Reinhold is the OpenJDK Lead. So one person decides across SE what can move forward.

The JCP can only supervise and vote on the bundle of JEPs that are put into a JSR. When that is a large bundle, there is no real supervision or voting at all.

Proposal - Working Group

This issue will not be solved in a short meeting. We propose a JCP EC Working Group to further consider what needs to be done here.

Working Group Members

JCP EC Members

Geir Magnusson
London JUG - Martijn Verburg & Ben Evans
Hazelcast Inc. - Greg Luck and Chris Engelbert

Third Party Experts

Peter Lawrey