What to do about `sun.misc.Unsafe` and Pals?

27 October 2015
Edited by Martijn Verburg - 28 October 2018

Contributors:
Greg Luck, Hazelcast, EC Representative
Chris Engelbert, Hazelcast, EC Representative
Martijn Verburg, LJC, jClarity, Java Champion, EC Representative
Ben Evans, LJC, jClarity, Java Champion, EC Representative
Gil Tene, Azul Systems EC Representative
Peter Lawrey, Higher Frequency Trading, Java Champion
Rafael Winterhalter, Bouvet ASA
Richard Warburton, Monotonic Ltd.
Henri Tremblay, Java Champion, as Himself

Status: Final

Status

27th of October 2015 - Final document shared with the community at that time.
28th of October 2018 - Minor edits made before continued discussion within the Java Champions group.

Disclaimer

This document is **NOT** an official Oracle document. It neither was started nor influenced by Oracle or its employees. It was started as a community effort to raise awareness of the possible problems that arose from an early proposal (JEP 260 now supersedes that earlier plan) for Java 9 to remove or hide access to `sun.misc.Unsafe` and other internal unsupported APIs.

The purpose of this document is to feature an overview of the current situation and the possible, publicly available solutions that are proposed and matches them against the `sun.misc.Unsafe` (and friends) feature set.

Table of Contents

Status

Disclaimer

Table of Contents

Summary

Current Challenges
- Widespread Community use of sun.misc.Unsafe - a proprietary API
- Community use of other (sun.*) internal proprietary APIs
- Why JEP 260 is needed
  - JNI is not a viable replacement
- Missing Cross-Vendor Specification for 'safe' Unsafe features

Uses of sun.misc.Unsafe
- Examples of projects/products using Unsafe

Uses of sun.nio.ch.FileChannelImpl.*
- Examples of projects/products using sun.nio.ch.FileChannelImpl

Uses of sun.nio.ch.DirectBuffer.*
- Examples of projects/products using sun.nio.ch.DirectBuffer

Uses of sun.misc.Cleaner
- Examples of projects/products using sun.misc.Cleaner

GAP Analysis of Features <-> JEPs
- JEPs - Possible Replacements for some aspects of Unsafe
- A mapping of Unsafe Features to JEPs / Features
  - Fields in sun.misc.Unsafe

sun.misc.Unsafe Features in detail
- allocateInstance
  - Usage
  - Possible replacements

Working Group
- Working Group Members

The proposal from Mark Reinhold, Chief Architect
- Status of Implementation of this Solution (Updated 27 October 2015)
  - -XX Flag to allow Unsafe Access
  - Current sun.misc.Unsafe in Java 9 Trunk

Summary

`sun.misc.Unsafe` is an unsupported API which has popular usage in the industry but it is not an official standard for Java SE. There are plans for its gradual encapsulation and removal/deprecation under the modularization efforts for Java SE 9 (See JEP 260). This is deemed to be *"a good thing"* in general.

Many organizations and framework, library, product owners have stated they will be unable to move to Java SE 9 without some sort of replacement for some of the 'safe' `sun.misc.Unsafe` features.

While normally the community at large should work on and propose JEPs through OpenJDK to provide the same functionality as the 'safe' parts of `sun.misc.Unsafe`, it is generally acknowledged that there will not be enough time for this before Java SE 9 is feature complete (~Nov 2015 - hence the compromise that is proposed through JEP 260). One of the main purposes of this document is to map out features currently used versus their proposed replacements (if any exist).

Current Challenges

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 become 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.

Community use of other (sun.*) internal proprietary APIs

Several products/projects use `sun.nio.*` APIs for fast low-level data transfer, especially in combination with `sun.misc.Unsafe` calls.

TODO Flesh out this section

Why JEP 260 is needed

Access to `sun.misc.Unsafe` was slated to be restricted in the upcoming Java 9 release as part of Project Jigsaw. This would have 'out of the box' broken many products/projects.

JNI is not a viable replacement

While additional JNI libraries could provide the same functionality as `Unsafe`, such libraries would need to provide 32-bit and 64-bit implementations as well as Windows and Linux variations. This would have been less safe than the Unsafe class in Java 8, or a potential replacement in Java 9, as each framework would have to offer its 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 intrinsic available in Unsafe, such an operation would have to be written in JNI to avoid crossing the JNI barrier too many times.

Missing Cross-Vendor Specification for 'safe' Unsafe features

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 sun.misc.Unsafe

Broadly speaking `sun.misc.Unsafe` is used for:

- Mocking Classes
- Low, very predictable latencies (low GC overhead)
- Fast de-/serialization
- Thread safe 64-bit sized native memory access (for example off-heap)
- Atomic memory operations
- Efficient object/memory layouts
- Fast   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.
- Access to array items with volatile semantic
- Uniform representation of memory chunks in byte arrays and direct buffers

Examples of projects/products using Unsafe

- MapDB
- Netty
- Hazelcast
- Cassandra
- Mockito / EasyMock / JMock / JMock
- Scala Specs
- Spock

- Robolectric
- Grails
- Gson
- Neo4j
- Apache Hadoop
- Apache Ignite
- Apache Spark
- Apache Kafka
- Apache Wink
- Apache Storm
- Apache Flink
- Apache Continuum
- Zookeeper
- Dropwizard
- Metrics (AOP)
- Kryo
- ByteBuddy
- Hibernate
- Liquibase
- Spring Framework (via Objenesis, with a fallback)
- Ehcache (sizeof)
- OrientDB
- Chronicle (OpenHFT)
- Apache Hadoop, Apache HBase (hadoop based database)
- GWT
- Disruptor
- JRuby
- Scala
- Akka
- Real Logic Agrona
- Aeron
- Simple Binary Encoding
- XRebel
- Presto (Slice, jol)
- Quartet FS ActivePivot (Off-Heap memory management)
- LWJGL (graphics library used by Minecraft)
- XAP
- XStream
- CapLogic
- WildFly
- Infinispan

See also Use at Your Own Risk: The Java Unsafe API in the Wild.

Uses of sun.nio.ch.FileChannelImpl.*

The following internal APIs are used in combination with Unsafe but are in addition to the Unsafe class. Currently, the only alternative is to use JNI (as these methods do)

`sun.nio.ch.FileChannelImpl.map0` - to map in 64 bit regions
`sun.nio.ch.FileChannelImpl.unmap0` - to unmap 64 bit regions.

If there was a 64-bit version of MappedByteBuffer, these wouldn't be needed.

Examples of projects/products using sun.nio.ch.FileChannelImpl

- Chronicle Core - for mapping > 31-bit sizes.
- one-nio
- CapLogic
- LevelDB
- Neo4J

Uses of sun.nio.ch.DirectBuffer.*

The following methods can be replaced via reflection, however, using an API (even an internal one) is nicer.

`sun.nio.ch.DirectBuffer.address()` to get the address of a ByteBuffer
`sun.nio.ch.DirectBuffer.cleaner()` to release memory deterministically.

In low GC systems, by design, you cannot assume that the GC will run often or regularly. Ideally, the system should run for 24 hours to a week between *minor* GCs. If you are producing < 700 KB/s of garbage this is possible. In such an environment, with the bulk of your data off-heap, you need to be able to clean up these resources directly.

Examples of projects/products using sun.nio.ch.DirectBuffer

- Chronicle Bytes
- Apache Spark
- Kryo
- Cassandra

Uses of sun.misc.Cleaner

`sun.misc.Cleaner` is used to handle cleanup of memory allocated via Unsafe

There is no alternative using reflection or JNI. In some cases reflection is used to get the Cleaner from ByteBuffer, in some cases, a cast to `sun.nio.ch.DirectBuffer` is used and in some cases, Cleaners are created directly .

Examples of projects/products using sun.misc.Cleaner

- Chronicle Bytes - NativeBytesStore
- Netbeans
- Quartet FS (Probably)
- Netty and IntelliJ
- JDBM
- Apache Lucene
- Kryo
- VoltDB
- cleakka
- SAP (Probably)
- jMonkeyEngine
- fqueue
- imageio-ext
- exs-aion-emu
- Aeron
- Agrona

GAP Analysis of Features <-> JEPs

Here's an attempt to look at what JEPs would need to be raised to provide safe, support versions of existing 'safe' `sun.misc.Unsafe` features.

JEPs - Possible Replacements for some aspects of Unsafe

Green = Already available / will be available in Java 9 / 10
Orange = May be available in Java 9 / 10
Red = Unknown

Proposal	**Expected in Java 9**	**Expected in Java 10**
VarHandle (with uses suggested in JEP 193)	🟩 Yes, although the MemoryModel update is still draft and VarHandle depends on it.	🟩 -
Project Panama (JFFI, JEP 191)	🟩 No	🟩 Yes
Serialization 2.0 (JEP 187)	🟥 No (JEP disappeared (wayback machine)	🟥 No
ValueTypes (no JEP)	🟧 No	🟧 Maybe
Arrays 2.0 (no JEP)	🟧 No	🟧 Maybe
Variable Object Layout (no JEP)	🟥 No	🟥 No
Extending Field / Array reflection access	🟥 Not yet discussed	🟥 Not yet discussed

A mapping of Unsafe Features to JEPs / Features

TODO Need to look at the OpenJDK 8 implementation

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

For the current status of Java 9 Unsafe changes, see the trunk source:

http://hg.openjdk.java.net/jdk9/dev/jdk/file/8271f42bae4a/src/java.base/share/classes/sun/misc/Unsafe.java

Feature	**`sun.misc.Unsafe`**	**Usage** Google Search Results$	**Java 9 / 10 replacement?**
Memory Information	`addressSize` `pageSize`	17,700 65,800	Unknown
Objects	`allocateInstance` `objectFieldOffset`	5,290 2,820	Reflection (Field), JEP 193?

Classes	`staticFieldOffset`
	`defineClass`
	`defineAnonymousClass`
	`ensureClassInitialized`
		2,820
		11,400
		2,350
		2,760
			Unknown

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`
	
	`copyMemory(src, .., 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)
		
		19.400
		
		19.900
			Unknown

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, …)`
		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)
			￼Unknown

Fences	`storeFence`
	`readFence`
	`fullFence`
		1.820
		202
		1.900
			￼Unknown

$ 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()}

sun.misc.Unsafe Features in detail

allocateInstance

We will detail here the case of `allocateInstance`. This method is used to create a Java class without calling any constructor. Calling the default constructor of the `Object` class to create another class brings the same result.

Note: The way the serialization mechanism creates an object is close to this. However, it calls the constructor of the first non-serializable class -which is frequently `Object`.

Usage

Not calling a constructor is useful for 3 use-cases

- **Mocking:** Mocking frameworks want to be able to create fake objects with a recorder instead of a real behavior. They are usually found during unit testing but could also be used for stubbing in later test phases. They should not be needed in a production environment
- **Serialization:** Serialization frameworks want to be able to bypass constructors instead of forcing to have a default constructor. This is a legitimate request since the Java serialization is allowed to do so
- **Proxying:** A proxy wants to add some behavior to a class dynamically. They should extend the original class and delegate to it. However, calling a constructor when creating a proxy is useless and possibly not working

Possible replacements

There are currently many ways in the OpenJDK to do the same thing as `allocateInstance`.

- `sun.reflect.ReflectionFactory.newConstructorForSerialization` provides a Constructor that will instantiate any class by calling `Object` default constructor
- Writing the actual bytecode of a class but do not call any super constructor in its constructor. This will only work when `-Xverify:none` is specified
- Writing the bytecode of a class extending `sun.reflect.MagicAccessorImpl`. This class will be allowed to instantiate another class by calling `Object` default constructor

Currently, `ReflectionFactory` is a good candidate as a replacement. An unverified benchmark is showing that it is much faster. The class could be moved to a `java.*` package and the feature become official. Accessing the single instance of the class is protected by the security manager using the `reflectionFactoryAccess` right.

Working Group

In order to propose sane replacements in OpenJDK via the JEP process, there needs to be some serious analysis from the industry at large. Here is an (incomplete) list of people/organizations who will assist in the GAP analysis.

Working Group Members

Geir Magnusson
London JUG - Martijn Verburg & Ben Evans
Hazelcast Inc. - Greg Luck and Chris Engelbert
Credit Suisse
Microdoc - Hendrik
Azul - Gil Tene
Goldman Sachs - John Weir
Fujitsu - Maike De Nicol
SouJava
Eclipse Software Foundation
DataStax - Jonathan Ellis and Ariel Weisberg
Chronicle Software - Peter Lawrey

The community at large is also welcome to join in.

The mailing list is java9unsafe@googlegroups.com.

The proposal from Mark Reinhold, Chief Architect

Mark Reinhold's proposal is detailed in
http://mail.openjdk.java.net/pipermail/jigsaw-dev/2015-August/004433.html

"It's well-known that some popular libraries make use of a few of these internal APIs, such as sun.misc.Unsafe, to invoke methods that would be difficult, if not impossible, to implement outside of the JDK. To ensure the broad testing and adoption of the release we propose to treat these critical APIs as follows:

- If it has a supported replacement in JDK 8 then we will encapsulate it in JDK 9;

- If it does not have a supported replacement in JDK 8 then we will not encapsulate it in JDK 9, so that it remains accessible to outside code; and, further,

- If it has a supported replacement in JDK 9 then we will deprecate it in JDK 9 and encapsulate it, or possibly even remove it, in JDK 10.

The critical internal APIs proposed to remain accessible in JDK 9 are listed in JEP 260 [2]. Suggested additions to the list, justified by real-world use cases and estimates of developer and end-user impact, are welcome.

Please also see an updated http://openjdk.java.net/jeps/260 which states:

The critical internal APIs proposed to remain accessible in JDK 9 are:

- *sun.misc.Cleaner*
- *sun.misc.{Signal, SignalHandler}*
- *sun.misc.Unsafe (The functionality of many of the methods in this class is now available via variable handles (JEP 193).)*
- *sun.reflect.Reflection::getCallerClass (The functionality of this method may be provided in a standard form via JEP 259.)*
- *sun.reflect.ReflectionFactory"*

Status of Implementation of this Solution (Updated 27 October 2015)

-XX Flag to allow Unsafe Access

Nothing yet.

Current sun.misc.Unsafe in Java 9 Trunk

http://hg.openjdk.java.net/jdk9/dev/jdk/file/8271f42bae4a/src/java.base/share/classes/sun/misc/Unsafe.java