Usage and Impact of the Unsafe Java API

Authors Omitted for Submission

Abstract

Java is a safe language. Its runtime environment provides strong safety guarantees that any Java application can rely on. Or so we think. We show that the runtime actually does not provide these guarantees—for a large fraction of today's Java code. Unbeknownst to many application developers, the Java runtime includes a "backdoor" that allows expert library and framework developers to circumvent Java's safety guarantees. This backdoor is there by design, and is well known to experts, as it enables them to write high-performance "systems-level" code in Java.

For much the same reasons that safe languages are preferred over unsafe languages, these powerful—but unsafe—capabilities in Java should be restricted. They should be made safe by changing the language, the runtime system, or the libraries. At the very least, their use should be restricted. This paper is a step in that direction.

We analyzed 70 GB of compiled Java code, spread over 75,405 Java archives, to determine how Java's unsafe capabilities are used in real-world libraries and applications. We found that 25% of Java bytecode archives depend on unsafe third-party Java code, and thus Java's safety guarantees cannot be trusted. We identify 14 different usage patterns of Java's unsafe capabilities, and we provide supporting evidence for why real-world code needs these capabilities. Our long-term goal is to provide a foundation for the design of new language features to regain safety in Java.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms Languages, Safety, Patterns, Maven Central, Stack Overflow

Keywords Unsafe, Bytecode Analysis, Java, Mining

1. Introduction

The Java Virtual Machine (JVM) executes Java bytecode and provides other services for programs written in many programming languages, including Java, Scala, and Clojure. The JVM was designed to provide strong safety guarantees. However, many widely used JVM implementations expose an API that allows the developer to access low-level, unsafe features of the JVM and underlying hardware, features that are unavailable in safe Java bytecode. This API is provided through an undocumented[1] class, *sun.misc.Unsafe*, in the Java reference implementation produced by Oracle.

Other virtual machines provide similar functionality. For example, the C# language provides an `unsafe` construct on the .NET platform[2], and Racket provides unsafe operations[3].

The operations *sun.misc.Unsafe* provides can be dangerous, as they allow developers to circumvent the safety guarantees provided by the Java language and the JVM. If misused, the consequences can be resource leaks, deadlocks, data corruption, and even JVM crashes.

We believe that *sun.misc.Unsafe* was introduced to provide better performance and more capabilities to the writers of the Java runtime library. However, *sun.misc.Unsafe* is increasingly being used in third-party frameworks and libraries. Application developers who rely on Java's safety guarantees have to trust the implementers of the language runtime environment (including the core runtime libraries). Thus the use of *sun.misc.Unsafe* in the runtime libraries is no more risky than the use of an unsafe language to implement the JVM. However, the fact that more and more "normal" libraries are using *sun.misc.Unsafe* means that application developers have to trust a growing community of third-party Java library developers to not inadvertently tamper with the fragile internal state of the JVM.

Given that the benefits of safe languages are well known, and the risks of unsafe languages are obvious, why exactly does one need unsafe features in third-party libraries? Are those features used in real-world code? If yes, how are they used, and what are they used for?

[1] `http://www.oracle.com/technetwork/java/faq-sun-packages-142232.html`

[2] `https://msdn.microsoft.com/en-us/en-en/library/chfa2zb8(v=vs.90).aspx`

[3] `http://docs.racket-lang.org/reference/unsafe.html`

We studied a large repository of Java code, Maven Central, to answer these questions. We analyzed 70 GB of compiled Java code, spread over 75,405 Java libraries, to determine the usage and impact of *sun.misc.Unsafe*. Our goal is to provide a strong foundation for informed decisions in the future evolution of the Java language and virtual machine.

The rest of this paper is organized as follows: Section 2 presents the concrete risks of using *sun.misc.Unsafe*. Section 3 describes the process we used to find usage patterns while Sections 4 and 5 present the methodology and results of our study. Section 6 introduces and discusses the patterns we found. Section 7 presents related work. Section 8 concludes the paper.

2. The Risks of Compromising Safety

We outline the risks of Unsafe by illustrating how the improper use of Unsafe violates Java's safety guarantees.

In Java, the unsafe capabilities are provided as instance methods of class *sun.misc.Unsafe*. Given that being unsafe is a risk, access to unsafe capabilities has been made less than straightforward. Class *sun.misc.Unsafe* is final, and its constructor is not public. Thus, creating an instance requires some tricks. For example, one can invoke the private constructor via reflection. This is not the only way to get hold of an unsafe object, but it is the most portable.

```
Constructor<Unsafe> c = Unsafe.class.
    getDeclaredConstructor();
c.setAccessible(true);
Unsafe unsafe = c.newInstance();
```

Listing 1. Instantiating an Unsafe object

Given the unsafe object, one can now simply invoke any of its methods to directly perform unsafe operations.

2.1 Violating Type Safety

In Java, variables are strongly typed. For example, it is impossible to store an int value inside a variable of a reference type. Unsafe can violate that guarantee: it can be used to store a value of any type in a field or array element.

```
class C {
  private Object f = new Object();
}
long fieldOffset = unsafe.objectFieldOffset(C.
    class.getDeclaredField("f"));
C o = new C();
unsafe.putInt(o, fieldOffset, 1234567890); //
    result undefined; f may now point to nirvana
```

Listing 2. *sun.misc.Unsafe* can violate type safety

2.2 Crashing the Virtual Machine

A quick way to crash the VM is to free memory that is in a protected address range, for example by calling `freeMemory` as follows.

```
unsafe.freeMemory(1);
```

Listing 3. *sun.misc.Unsafe* can crash the VM

In Java, the normal behavior of a method to deal with such situations is to throw an exception. Being unsafe, instead of throwing an exception, this invocation of `freeMemory` crashes the VM.

2.3 Violating Method Contracts

In Java, a method that does not declare an exception cannot throw any checked exceptions. Unsafe can violate that contract: it can be used to throw a checked exception that the surrounding method does not declare or catch.

```
void m() {
  unsafe.throwException(new Exception());
}
```

Listing 4. *sun.misc.Unsafe* can violate a method contract

2.4 Uninitialized Objects

Java guarantees that an object allocation also initializes the object by running its constructor. Unsafe can violate that guarantee: it can be used to allocate an object without ever running its constructor. This can lead to objects in states that the objects' classes would not seem to admit.

```
class C {
  private int f;
  public C() { f = 5; }
  public int getF() { return f; }
}

C c = (C)unsafe.allocateInstance(C.class);
assert c.getF()==5; // violated
```

Listing 5. *sun.misc.Unsafe* can lead to uninitialized objects

2.5 Monitor Deadlock

Java provides synchronized methods and synchronized blocks. These constructs guarantee that monitors entered at the beginning of a section of code are exited at the end. Unsafe can violate that contract: it can be used to asymmetrically enter or exit a monitor, and that asymmetry might be not immediately obvious.

```
void m() {
  unsafe.monitorEnter(o);
  if (c) return;
  unsafe.monitorExit(o);
}
```

Listing 6. *sun.misc.Unsafe* can lead to monitor deadlocks

The above examples are just the most straightforward violations of Java's safety guarantees. The *sun.misc.Unsafe* class provides a multitude of methods that can be used to violate most guarantees Java provides.

To sum it up: Unsafe is dangerous. But should anybody care? In the next sections we present a study to determine whether and how Unsafe is used in real-world third-party Java libraries, and to what degree real-world applications directly and indirectly depend on it.

3. Overview of Our Study

We believe we should care about the dangers of Unsafe if the third-party usage of Unsafe could impact common application code. We want to answer the following questions:

*Q*1 : **Does Unsafe impact common application code?** We want to understand to what extent third-party code actually uses Unsafe.

*Q*2 : **Which features of Unsafe are used?** As Unsafe provides many features, we want to understand which ones are actually used, and which ones can be ignored.

*Q*3 : **Why are Unsafe features used?** We want to investigate what functionality third-party libraries require from Unsafe. This could point out ways in which the Java language and/or the JVM need to be evolved to provide the same functionality, but in a safer way.

*Q*4 : **What problems do developers who use Unsafe encounter?** If Unsafe is not just dangerous, but also confusing or difficult to use, then its use by third-party developers is particularly problematic. If there are specific Unsafe features or usage patterns that developers worry about, it would make sense to evolve Java or the JVM to provide safer alternatives in that direction.

To answer the above questions, we need to determine whether and how unsafe is actually used in real-world third-party Java libraries, and to what degree real-world applications directly and indirectly depend on such unsafe libraries. To achieve our goal, several elements are needed.

Code Repository. As a code base representative of the "real world", we have chosen the Maven Central[4] software repository. The rationale behind this decision is that a large number of well-known Java projects deploy to Maven Central using Apache Maven[5]. Besides code written in Java, projects written in Scala are also deployed to Maven Central using the Scala Build Tool (sbt)[6]. Moreover, Maven Central is the largest Java repository[7], and it contains projects from the most popular source code management repositories, like GitHub[8] and SourceForge[9].

Artifacts. In Maven terminology, an artifact is the output of the build procedure of a project. An artifact can be any type of file, ranging from a `.pdf` to a `.zip` file. However, artifacts are usually `.jar` files, which archive compiled Java bytecode stored in `.class` files.

Bytecode Analysis. We examine these kinds of artifacts to analyze how they use *sun.misc.Unsafe*. We use a bytecode analysis library to search for method call sites and field accesses of the *sun.misc.Unsafe* class.

Dependency Analysis. We define the impact of an artifact as how many artifacts depend on it, either directly or indirectly. This helps us to define the impact of artifacts that use *sun.misc.Unsafe*, and thus the impact *sun.misc.Unsafe* has on real-world code overall.

Usage Pattern Detection. After all call sites and field accesses are found, we analyze this information to discover usage patterns. It is common that an artifact exhibits more than one pattern. Our list of patterns is not exhaustive. We have manually investigated the source code of the 100 highest-impact artifacts using *sun.misc.Unsafe* to understand why and how they are using it.

Stack Overflow Analysis. We studied problems encountered using *sun.misc.Unsafe* by analyzing the Stack Overflow question/answer database. After discovering usage patterns in the Maven archive, we use Stack Overflow to correlate them to discussions. Our goal is to understand the difficulties in implementing certain Unsafe usage patterns.

4. Is Unsafe Used?

In this section we answer our first two research questions: whether Unsafe impacts common application code, and which features of Unsafe are actually used.

We do this by mining Maven Central. The complete scripts and results used for this study are available online[10].

4.1 Gathering Artifacts

The complete Maven Central repository contains 900,901 artifacts, 100,885 unique artifacts—artifacts are versioned—and consists of ca. 1.7 TB of data[11]. For our analysis we only look at the last version of each artifact to search for *sun.misc.Unsafe* uses. Moreover we are only interested in a subset of this data: artifacts that archive compiled bytecode (`.class` files) *e.g.,* `.jar`, `.war`, `.ear`, and `.ejb` files.

We downloaded all artifacts subject to analysis from a mirror of Maven Central provided by the ibiblio Digital Archive[12]. We downloaded the archive between February 16th and 20th, 2015. The downloaded repository consists of ca. 70 GB of data from 75,405 unique artifacts.

4.2 Determining Usage

To search for *sun.misc.Unsafe* static use, we mined bytecode using the following facts about the *sun.misc.Unsafe* class: *a*) it is declared as `final`; *b*) it inherits directly from `java.lang.Object`; *c*) its public methods (except for `getUnsafe`) are instance methods; and *d*) its public fields are declared as `static final`.

We implement our analysis on top of ASM [14]. Our analysis finds all virtual method invocation sites where the call target is of type *sun.misc.Unsafe*, and all static reads of fields of class *sun.misc.Unsafe*.

[4] `http://central.sonatype.org/`

[5] `http://maven.apache.org/`

[6] `http://www.scala-sbt.org/`

[7] `http://www.modulecounts.com/`

[8] `https://github.com/`

[9] `http://sourceforge.net/`

[10] Repository Omitted for Submission

[11] `http://search.maven.org/#stats`

[12] `http://mirrors.ibiblio.org/maven2/`

Sometimes *sun.misc.Unsafe* is used through reflection to avoid compilation dependencies (given that *sun.misc.Unsafe* is not part of the public API). Our study is restricted to static uses of *sun.misc.Unsafe*, which is a limitation of our work.

Our analysis found 40,343 uses of *sun.misc.Unsafe*—40,040 call sites and 303 field accesses—distributed over 720 different artifacts. This initial result shows that Unsafe is indeed used in third-party code.

4.3 Determining Impact

Unsafe does not only impact the artifacts that use it, but it transitively impacts all artifacts depending on those artifacts. We thus need to determine the transitive closure of unsafety.

Maven projects are described using POM files, which may contain dependency information. 40,622 of the artifacts we downloaded include such dependency information.

In Maven, dependencies have a scope. For example, a library artifact may depend on a testing framework artifact only for the purpose of testing. It will not depend on the testing framework for production runs.

We use the dependency information to determine the impact of the artifacts that use *sun.misc.Unsafe*. We rank all artifacts according to their impact (the number of artifacts that directly or indirectly depend on them). High-impact artifacts are important; a safety violation in them can affect any artifact that directly or indirectly depends on them. We find that while overall about 1% of artifacts directly use Unsafe, for the top-ranked 1000 artifacts, 3% directly use Unsafe. Thus, Unsafe usage is particularly prevalent in high-impact artifacts, artifacts that can affect many other artifacts.

Moreover, we found that 19,333 artifacts (47% of the 40,622 artifacts with dependency information, or 25% of the 75,405 artifacts we downloaded) directly or indirectly depend on *sun.misc.Unsafe*. Thus, *sun.misc.Unsafe* usage in third-party code indeed impacts a large fraction of projects.

4.4 Which Features of Unsafe Are Actually Used?

Figures 1 and 2 show all instance methods and static fields of *sun.misc.Unsafe*. For each member we show how many call sites or field accesses we found across the artifacts. The class provides 120 public instance methods and 20 public fields (version 1.8 update 40). The figure only shows 93 methods because the 18 methods in the *Heap Get* and *Heap Put* groups, and *staticFieldBase* are overloaded, and we combine overloaded methods into one bar.

We categorized the members into groups, based on the functionality they provide:

- The *Alloc* group contains only the *allocateInstance* method, which allows the developer to allocate a Java object without executing a constructor. This method is used 125 times.
- The *Array* group contains methods and fields for computing relative addresses of array elements. The fields were added as a simpler and potentially faster alternative in a

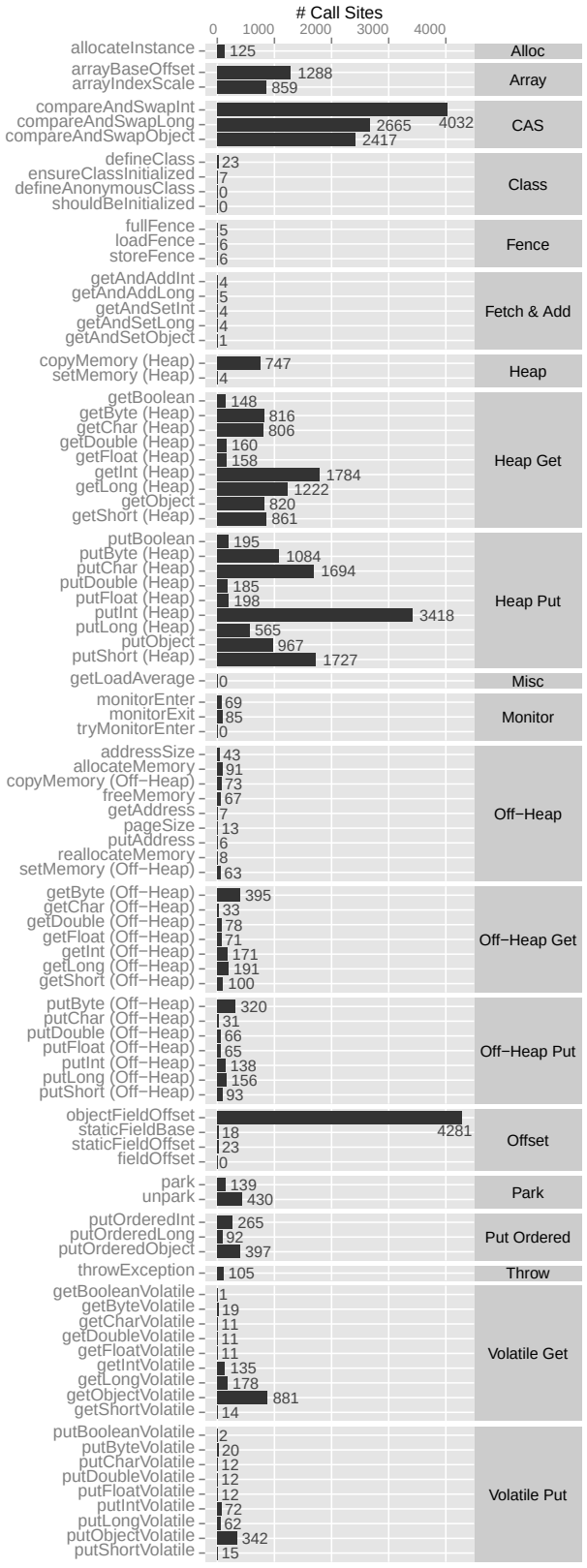

Figure 1. *sun.misc.Unsafe* method usage on Maven Central

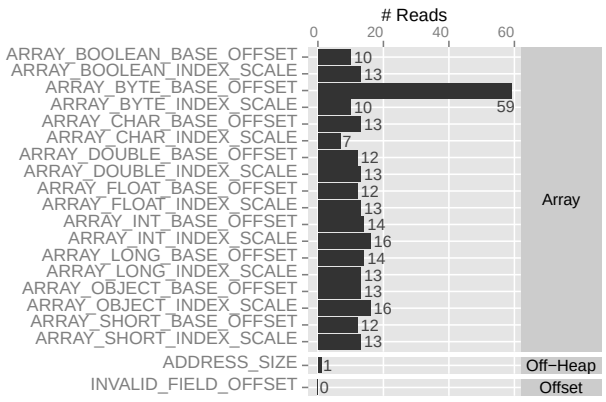

Figure 2. *sun.misc.Unsafe* field usage on Maven Central

more recent version of Unsafe. The value of all fields in this group are constants initialized with the result of a call to either *arrayBaseOffset* or *arrayIndexScale* in the *Array* group. The figures show that the majority of sites still invoke the methods instead of accessing the corresponding constant fields.

- The *CAS* group contains methods to atomically compare-and-swap a Java variable. These operations are implemented using processor-specific atomic instructions. For instance, on *x86* architectures, *compareAndSwapInt* is implemented using the `CMPXCHG` machine instruction. Figure 1 shows that these methods represent the most heavily used feature of Unsafe.
- Methods of the *Class* group are used to dynamically load and check Java classes. They are rarely used, with *defineClass* being used the most.
- The methods of the *Fence* group provide memory fences to ensure loads and stores are visible to other threads. These methods are implemented using processor-specific instructions. These methods were introduced only recently in Java 8, which explains their limited use in our data set. We expect that their use will increase over time and that other operations, such as those in the *Put Ordered*, or *Volatile Put* groups will decrease as programmers use the lower-level fence operations.
- The *Fetch & Add* group, like the *CAS* group, allows the programmer to atomically update a Java variable. This group of methods was also added recently in Java 8. We expect their use to increase as programmers replace some calls to methods in the *CAS* group with the new functionality.
- The *Heap* group methods are used to directly access memory in the Java heap. The *Heap Get* and *Heap Put* groups allow the developer to load and store a Java variable. These groups are among the most frequently used ones in Unsafe.
- The *Misc* group contains the method *getLoadAverage*, to get the load average in the operating system run queue assigned to the available processors. It is not used.
- The *Off-Heap* group provides access to unmanaged memory, enabling explicit memory management. Similarly to the *Heap Get* and *Heap Put* groups, the *Off-Heap Get* and *Off-Heap Put* groups allow the developer to load and store values in Off-Heap memory. The usage of these methods is non-negligible, with *getByte* and *putByte* dominating the rest. The value of the *ADDRESS_SIZE* field is the result of the method *addressSize()*.
- Methods of the *Offset* group are used to compute the location of fields within Java objects. The offsets are used in calls to many other *sun.misc.Unsafe* methods, for instance those in the *Heap Get*, *Heap Put*, and the *CAS* groups. The method *objectFieldOffset* is the most called method in *sun.misc.Unsafe* due to its result being used by many other *sun.misc.Unsafe* methods. The *fieldOffset* method is deprecated, and indeed, we found no uses. The *INVALID_FIELD_OFFSET* field indicates an invalid field offset; it is never used.
- The *park* and *unpark* methods are contained in the *Park* group. With them, it is possible to block and unblock a thread's execution.
- The *Put Ordered* group has methods to store to a Java variable without emitting any memory barrier but guaranteeing no reordering across the store.
- The *throwException* method is contained in the *Throw* group, and allows one to throw checked exceptions without declaring them in the `throws` clause.
- Finally, the *Volatile Get* and *Volatile Put* groups allow the developer to store a value to a Java variable with `volatile` semantics.

It is interesting to note that despite our large corpus of code, there are several Unsafe methods that are never actually called. If Unsafe is to be used in third-party code, then it might make sense to extract those methods into a separate class to be only used from within the runtime library.

5. Question/Answer Database Analysis

To complement our analysis of *sun.misc.Unsafe* usage in practice, and in particular to answer the questions relative to which features are commonly used ($Q2$), why they are used ($Q3$), and if they generate issues or problems ($Q4$), we decided to search for further evidence in Stack Overflow.

A prerequisite for performing the analysis is to identify discussions concerning the usage of *sun.misc.Unsafe*. Such identification cannot rely only on the tagging system provided by Stack Overflow. In general, the topic is rarely discussed, and the only tag called *unsafe* is rather used to identify general issues about unsafety at different

abstraction levels than the typical uses of *sun.misc.Unsafe*. A more precise analysis of the contents of discussions is required to understand if a discussion actually involves, or mentions, *sun.misc.Unsafe*. We use *island parsing* [17] of structured fragments in natural language artifacts [2, 20] to mine discussions belonging to the Stack Overflow data dump of September 2014[13] to discover discussions that involve *sun.misc.Unsafe*.

5.1 (Island) Parsing Stack Overflow Discussions

Since we cannot rely solely on discussion tagging, we need to discover and analyze specific constructs that reveal the usage of *sun.misc.Unsafe*. For example, a discussion could report a code sample using *sun.misc.Unsafe*, or a user could mention the class (or one of its fields/methods) in an answer to a question concerning some specific problem that the usage of *sun.misc.Unsafe* can tackle. To identify such elements, we devised an island grammar [17] capable of identifying constructs of interests immersed in natural language [2]. Our island grammar allows to identify constructs like stack traces, stack trace lines, and incomplete and complete Java code fragments, like method invocations or mentions inside the natural language narrative. The constructs we devised in the parser are not used for the simple identification and extraction, but we use them to model the contents with a Heterogeneous Abstract Syntax Tree (H-AST) [20] that can be traversed to analyze the information afterwards.

5.2 Identifying Relevant Discussions

To identify Stack Overflow discussions concerning the *sun.misc.Unsafe* class, we start by analyzing all the discussions whose tags contain one among *java*, *scala*, *android*, and *jvm*. To understand if a discussion concerns *sun.misc.Unsafe*, we (i) search for uses of one of the fields or methods exposed by *Unsafe* or (ii) we identify any mention of the class itself. We focus on the following H-AST nodes to ensure a discussion matches one of the two criteria:

Qualified Identifiers: Qualified identifiers appear in constructs like import declarations and stack trace lines. We check that the qualified identifier matches values such as *Unsafe*, *unsafe*, *UNSAFE*, or the fully-qualified type *sun.misc.Unsafe*. In case of match, the post is marked as mentioning the type *Unsafe*. Moreover, if the last part of the qualified identifier matches the name of one of the *sun.misc.Unsafe* fields, we mark the post as mentioning a field of *Unsafe*.

Method Invocations: Each node matching a method invocation is analyzed to understand if the method name belongs to *sun.misc.Unsafe*.

Strict Identifiers: We extract identifiers respecting the Java naming conventions for methods and that are present in the narrative. Identifiers beginning with a lowercase letter and containing a case change (*e.g.,* fieldOffset) are considered as method names. The method name must then match one of the methods declared in *Unsafe*. We also check qualified identifiers appearing in the narrative and respecting naming convention for classes. We look for all the occurrences of qualified identifiers composed by three identifiers at least (*e.g., sun.misc.Unsafe*). Whatever matches this construct is treated as a normal qualified identifier.

5.3 Refining Parsing Results

The *park* method of *sun.misc.Unsafe* appears at the top of stack traces of threads that are disabled for scheduling. At any time during execution, one or more threads may be parked. Thus, if the JVM dumps stack traces for all threads, the *park* method usually appears in at least one of the stack traces. In this case, the presence of the method *park* does not represent a relevant usage of *Unsafe* and makes the *park* method the most mentioned in Stack Overflow. For this reason, we ignore occurrences of *park* inside stack traces.

We collected 20,915 discussions matching at least one of the two criteria, out of which 560 discussions report only the type *Unsafe*, 20,426 contain a method with a name matching the ones of *Unsafe*, and 5 discussions mention a field of *Unsafe*. However, if the presence of the type *Unsafe* guarantees that the discussion is likely about *sun.misc.Unsafe*, the lone presence of the method name does not guarantee that. For example, methods like *getInt* and *getFloat* can be found in other classes like *java.nio.ByteBuffer*[14]. Neither does the absence of the type in our parsing results guarantee that the discussion does not concern *sun.misc.Unsafe*. Indeed, to avoid false positives, we do not check at parsing time if the lone term "unsafe" is mentioned in the natural language parts of a discussion.

After parsing, we consider all discussions with a method name of the class *Unsafe*, and we perform a pure text search for the term "unsafe". Out of 20,426 discussions that mention an *Unsafe* method, 49 discussions also contain the term "unsafe". We manually inspected and verified each discussion, resorting to 18 discussions effectively reporting a usage of *sun.misc.Unsafe*. Thus, our final dataset contains a 560 discussions explicitly using the type *Unsafe*, and 18 discussions reporting the method name only and the term "unsafe", for a total of 578 discussions that effectively concern *sun.misc.Unsafe*. Of these ones, 163 questions mention both the type (or the term *unsafe*) and at least one of its methods.

5.4 Findings on Stack Overflow Discussions

Figure 3 presents an overview of *sun.misc.Unsafe* method mentions in Stack Overflow. The mentions are presented by distinguishing whether they appear only in the question, only in the answer, or in both. The list of methods does not distinguish between overloaded variants. In fact, people often mention method names without formal or actual parameters.

[13] `https://archive.org/details/stackexchange`

[14] `http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html`

Thus, in many cases, to understand which is the overloaded alternative one would have to do a manual inspection.

Most method mentions appear in answers only, indicating a prevalence of the use of *sun.misc.Unsafe* as a proposed solution for some task. In other frequent cases, they are present in both questions and answers. More rarely, they are present in questions only, a case that captures a clarification from the user that mostly requires narrative only (*e.g.,* for documentation). The most mentioned method in Stack Overflow discussions is *allocateInstance*, followed by *getInt*. The former is often proposed in the replies, suggesting its use for some particular task, or it is present in both questions and answer, discussing a particular task for which the developer is already aware of its possible use. The most frequently occurring groups of methods are *Get* and *Off-Heap*. Moreover, the methods mentioned most in questions are the ones in the *Put* group.

We further investigate the extracted Stack Overflow discussions by analyzing the popularity of the users engaging in discussions and by characterizing the most common classes of discussions through a manual inspection.

Popularity of Repliers. To understand the difficulty of the topics, we collected all the repliers of the questions in our final dataset. Focusing on repliers allows us to understand which users are capable of discussing topics related to *sun.misc.Unsafe*. The average reputation of all the selected repliers is 18,000. This is just below the level of *trusted user*[15], the highest reputation rank to get special privileges on Stack Overflow. If we refine our selection to include only the answers mentioning *sun.misc.Unsafe* or one of its methods, the average reputation of the corresponding repliers reaches around 21,770, which is above the maximum privilege threshold. This is evidence that the topics related to *sun.misc.Unsafe* and its use require a significant development experience.

To get further evidence that topics related to *Unsafe* attract expert users, we computed the distribution of replier's reputation among the ranks defined in the Stack Overflow reputation league[16]. For each reputation range, we computed the total number of users in the Stack Overflow dump, the amount of distinct users that replied to questions in our dataset, and among them, the ones who mentioned *sun.misc.Unsafe*. Table 1 shows the distribution of users per reputation rank.

One interesting finding is that the repliers of the questions in our dataset cover around 17% of the top-ranked developers, and around 6% of top-ranked developers mention one of *sun.misc.Unsafe* methods. We interpret this as further evidence that the topics involving *sun.misc.Unsafe* attract users with high-reputation, who are also skilled developers, thus

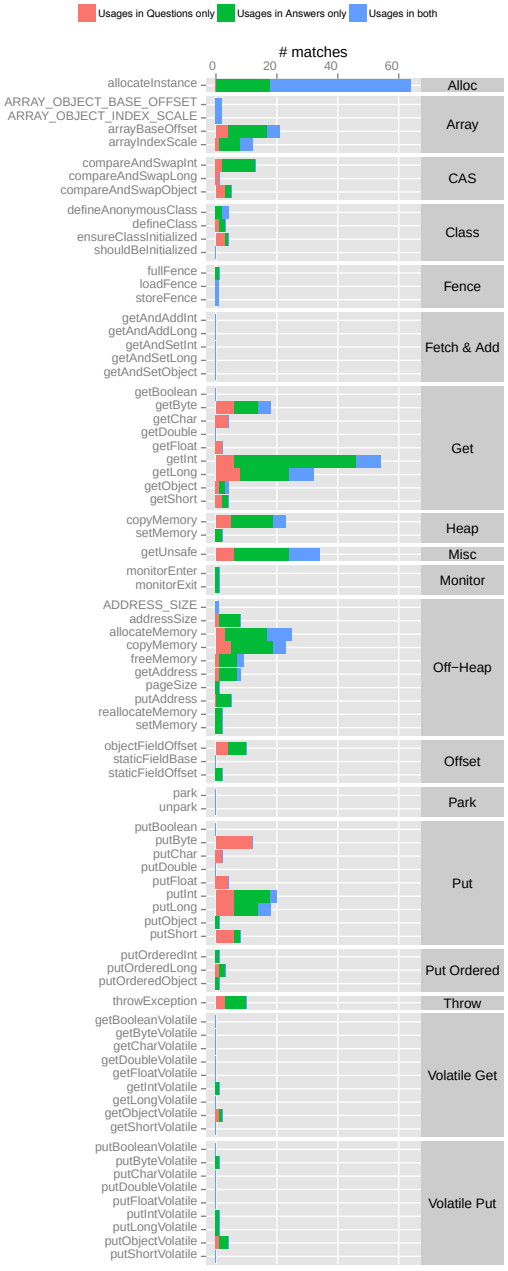

Figure 3. *Unsafe* method mentions on Stack Overflow

suggesting that these topics are relatively difficult and not for average developers.

Discussions Archetypes. We manually inspected the discussions concerning *Unsafe* to understand how it is discussed and used, and we devised a set of common discussion archetypes:

Lack of documentation. Unsafe is an undocumented API, and a primary archetype concerns the usage of this library. The lack of documentation and the volatility of the API makes it hard for the average developer to get a precise

[15] `http://stackoverflow.com/help/privileges`

[16] `http://stackexchange.com/leagues/1/alltime/stackoverflow`

Table 1. Distribution of Repliers

Reputation Range	All Users	Repliers	
		all	w/ *Unsafe*
1–199	3,276,655	120 (0.0%)	31 (0.0%)
200–499	80,105	51 (0.1%)	10 (0.0%)
500–999	49,825	62 (0.1%)	18 (0.0%)
1,000–1,999	30,833	103 (0.3%)	31 (0.1%)
2,000–2,999	11,847	65 (0.5%)	17 (0.1%)
3,000–4,999	10,151	93 (0.9%)	35 (0.3%)
5,000–9,999	7,462	133 (1.8%)	34 (0.5%)
10,000–24,999	4,278	140 (3.3%)	42 (1.0%)
25,000–49,999	1,271	72 (5.7%)	19 (1.5%)
50,000–99,999	444	41 (9.2%)	17 (3.8%)
>100,000	234	39 (16.7%)	14 (6.0%)

understanding. Therefore, developers ask the crowd to obtain clarification. For example, a user asked if an instance is allocated on-heap or off-heap when *allocateInstance* is used.[17] And a relatively popular question, entitled "Using sun.misc.Unsafe in real world", asked for typical use cases of *sun.misc.Unsafe*.[18]

Performance. Users coming from unmanaged languages like C and C++ discuss how to avoid the cost of Java's various safety checks. For instance, a user asked for an equivalent method call for *memcpy*, and one of the answers mentions *sun.misc.Unsafe*.[19] Another post discusses how to speed up array lookup[20].

Misdirected uses. We identified several discussions that show that developers consider or propose *sun.misc.Unsafe* for inappropriate purposes. For example, they discuss solutions for how to retrieve the address of an object. There is little reason for ever getting an objects' address. Depending on the GC, an object might be moved at any time. For this reason, none of the Unsafe methods require an object's address (they all require an object reference and an offset). Another post even discusses the use of the address to free an object on the Java heap[21]. This indicates a misunderstanding of basic memory management concepts. Yet another post asks for how to create memory leaks in Java[22], based on a job interview question. That interview question probably was testing the applicants' understanding of the benefits of GC, or of how to prevent the GC from collecting an object by holding on to that object. It is unlikely that the question was about Unsafe and its explicit off-heap memory management features. Another post asks why one cannot throw an exception from `Runnable.run()`[23]. One of the answers describes how to do so using Unsafe. It is unlikely that using Unsafe in that context is the best solution.

Overall, the availability of Unsafe to third-party developers who do not have a deep understanding of the JVM comes with the risk of uninformed and misdirected uses of Unsafe features. This risk is not unlike the risk of inappropriately using eval [23] in JavaScript.

6. Usage Patterns of *sun.misc.Unsafe*

This section presents the patterns we have found during our study. We present them sorted by how many artifacts depend on them, as computed from the Maven dependency graph described in Section 4. We present each pattern using the following template.

Description. What is the purpose of the pattern? What does it do?
Rationale. What problem is the pattern trying to solve? In what contexts is it used?
Unsafe members. Methods and fields of *sun.misc.Unsafe* used in the pattern. In some cases, there are multiple alternatives to implementing the pattern. These are explained when necessary in turn for each pattern.
Implementation. How is the pattern implemented using *sun.misc.Unsafe*?
Found in. Number of artifacts in Maven Central that contain the pattern.
Used by. Number of artifacts that transitively depend on the artifacts with the pattern.
Most used artifacts. The three most used artifacts containing the pattern, that is, the artifact with the most other artifacts that transitively depend upon it. Artifacts are shown using their Maven identifier, *i.e.,* ⟨*groupId*⟩:⟨*artifactId*⟩.
Issues. Issues to consider when using the pattern and problems discussed in the Stack Overflow database.
Alternatives. Potential alternatives to using the *Unsafe* for solving the same problem.

6.1 Allocate an object without invoking a constructor

Description. With this pattern an object can be allocated on the heap without executing its constructor.
Rationale. This pattern is useful for creating mock objects for testing and in deserializing serialized objects.
Unsafe members. *allocateInstance*
Implementation. The *allocateInstance* method takes as parameter a *java.lang.Class* object, and returns a new instance of that class. Unlike allocating an object directly, or through the reflection API, the object's constructor is not invoked.
Found in: 88 artifacts ***Used by:*** 14794 artifacts
Most used artifacts. *org.springframework:spring-core*[24], *org.objenesis:objenesis*[25], *org.mockito:mockito-all*[26]

[17] `http://stackoverflow.com/questions/21589159`
[18] `http://stackoverflow.com/questions/5574241`
[19] `http://stackoverflow.com/questions/6060163`
[20] `http://stackoverflow.com/questions/12226123`
[21] `http://stackoverflow.com/questions/24429777`
[22] `http://stackoverflow.com/questions/6470651`
[23] `http://stackoverflow.com/questions/11410042`
[24] `http://projects.spring.io/spring-framework/`
[25] `http://objenesis.org/`
[26] `https://github.com/mockito/mockito`

Issues. If the constructor is not invoked, the object might be left uninitialized and its invariants might not hold. Users of *allocateInstance* must take care to properly initialize the object before it is used by other code. This is often done in conjunction with other methods of *Unsafe*, for instance those in the *Heap Put* group, or by using the Java reflection API.
Alternatives. The language could be extended to support unconstructed objects, possibly with a type system to prevent usage of the object before initialization [22].

6.2 Process byte arrays in block

Description. When processing the elements of a byte array, better performance can be achieved by processing the elements 8 bytes at a time, treating it as a long array, rather than one byte at a time.
Rationale. The pattern is used for fast byte array processing, for instance, when comparing two byte arrays lexicographically.
Unsafe members. *arrayBaseOffset*, *getLong*, and optionally *arrayIndexScale* (to assert that the size of byte is equal to 1)
Implementation. The *arrayBaseOffset* is invoked to get the base offset of the *byte[]* class. Then the *getLong* method is used fetch and process 8 bytes of the array at a time.
Found in: 44 artifacts ***Used by:*** 12274 artifacts
Most used artifacts. *com.google.guava*:*guava*[27], *com.google.gwt*:*gwt-dev*[28], *net.jpountz.lz4*:*lz4*[29]
Issues. The pattern assumes that bytes in an array are stored contiguously. This may not be true for some VMs, *e.g.,* those implementing large arrays using discontinuous arrays or arraylets [3, 29]. Users of the pattern should be aware of the endianness of the underlying hardware. In one Stack Overflow discussion, this pattern is discouraged since it is non-portable and, on many JVMs, results in slower code[30].
Alternatives. The JVM's runtime compiler can be extended with optimizations for vectorizing byte array accesses.

6.3 Atomic operations

Description. To implement non-blocking concurrent data structures and synchronization primitives, hardware-specific atomic operations provided by *sun.misc.Unsafe* are used.
Rationale. Non-blocking algorithms often scale better than algorithms that use locking.
Unsafe members. Either *objectFieldOffset* or *arrayBaseOffset* in conjunction with *arrayIndexScale*, plus methods of the *CAS* group or the *Fetch & Add* group.
Implementation. To get the offset of a Java variable either *objectFieldOffset* or *arrayBaseOffset*/*arrayIndexScale* can be used. With this offset, the methods from the *CAS* or *Fetch & Add* groups are used to perform atomic operations on the variable. Other methods of *Unsafe* are often used in the implementation of concurrent data structures, including *Volatile Get/Put*, *Put Ordered*, and *Fence* methods.
Found in: 84 artifacts ***Used by:*** 10259 artifacts
Most used artifacts. *org.scala-lang*:*scala-library*[31], *org.apache.hadoop*:*hadoop-hdfs*[32], *org.glassfish.grizzly*:*grizzly-framework*[33]
Issues. Non-blocking algorithms can be difficult to implement correctly. Programmers must understand the Java memory model and how the *Unsafe* methods interact with the memory model.
Alternatives. The Java standard library provides classes for some concurrent data structures. The library also provides classes (*AtomicFieldReferenceUpdater*, *AtomicIntegerArray*, etc.) for safely performing atomic operations on fields and array elements, as well as several synchronizer classes. These can be used instead of the *Unsafe* atomic operations.

6.4 Strongly consistent shared variables

Description. Because of Java's weak memory model, when implementing concurrent code, it is often necessary to ensure that writes to a shared variable by one thread become visible to other threads, or to prevent reordering of loads and stores. Volatile variables can be used for this purpose, but *sun.misc.Unsafe* can be used instead with better performance. Additionally, because Java does not allow array elements to be declared volatile, there is no possibility other than to use *Unsafe* to ensure visibility of array stores. The methods of the *Put Ordered* groups and the *Volatile Get/Put* groups can be used for these purposes. In addition, the *Fence* methods were introduced in Java 8 expressly to provide greater flexibility for this use case.
Rationale. This pattern is useful for implementing concurrent algorithms or shared variables in concurrent settings. JRuby uses a *fullFence* to ensure visibility of writes to object fields.
Unsafe members. Methods of the *Fence* group, or methods of the *Volatile Get/Put* groups or *Put Ordered* group
Implementation. To ensure a write is visible to another thread, *Volatile Put* methods or *Put Ordered* methods can be used, even on non-volatile variables. Alternatively, a *storeFence* or *fullFence* can be used. *Volatile Get* methods ensure other loads and stores are not reordered across the load. A *loadFence* could also be used before a read of a shared variable.
Found in: 198 artifacts ***Used by:*** 9795 artifacts
Most used artifacts. *org.scala-lang*:*scala-library*, *org.jruby*:*jruby-core*[34], *com.hazelcast*:*hazelcast-all*[35]
Issues. Fences can replace volatile variables in some situations, offering better performance. Most of the uses of the

[27] https://github.com/google/guava
[28] http://www.gwtproject.org/
[29] https://code.google.com/p/lz4/
[30] http://stackoverflow.com/questions/12226123
[31] http://scala-lang.org/
[32] https://hadoop.apache.org/
[33] https://grizzly.java.net/
[34] http://jruby.org/
[35] http://hazelcast.com/

pattern use the *Put Ordered* and *Volatile Put* methods. Since they were added to Java only recently, there are currently few instances of the pattern that use the *Fence* methods.

Alternatives. Memory fence operations can be added to the standard library. The language can be changed to make volatile variables more flexible.

6.5 Park/Unpark Threads

Description. The park and unpark methods are useful for implementing locks and other blocking synchronization constructs.

Rationale. The alternative to parking a thread is to busy-wait, which uses CPU resources and does not allow other threads to proceed.

Unsafe members. *park*, *unpark*

Implementation. The *park* method blocks the current thread while the *unpark* methods unblocks a thread given as an argument.

Found in: 62 artifacts ***Used by:*** 7330 artifacts

Most used artifacts. *org.scala-lang:scala-library*, *org.codehaus.jsr166-mirror:jsr166y*[36], *com.netflix.servo:servo-internal*[37]

Issues. Users of *park* must be careful to avoid deadlock.

Alternatives. The standard library class *java.util.concurrent.locks.LockSupport* provides park and unpark methods to be used for implementing locks.

6.6 Update final fields

Description. This pattern is used to update a final field.

Rationale. Although it is possible to use reflection to implement the same behavior, updating a final field is easier and more efficient using *sun.misc.Unsafe*. Some applications update final fields when cloning objects or when deserializing objects.

Unsafe members. *objectFieldOffset*; and, at least one method of the *Heap Put* or *Volatile Put* groups.

Implementation. The *objectFieldOffset* and one of the put methods work in conjunction to directly modify the memory where a final field resides.

Found in: 11 artifacts ***Used by:*** 7281 artifacts

Most used artifacts. *org.codehaus.groovy:groovy-all*[38], *org.jodd:jodd-core*[39], *com.lmax:disruptor*[40]

Issues. There are numerous security and safety issues with modifying final fields. The update should be done only on newly created objects (perhaps also using *allocateInstance* to avoid invoking the constructor) before the object becomes visible to other threads. The Java Language Specification (Section 17.5.3) [10] recommends that final fields not be read until all updates are complete. In addition, the language permits compiler optimizations with final fields that can prevent updates to the field from being observed. Since final fields can be cached by other threads, one instance of the pattern uses *putObjectVolatile* to update the field rather than simply *putObject*. Using this method ensures that any cached copy in other threads is invalidated.

Alternatives. The reflection API can be used to implement the same functionality.

6.7 Non-lexically-scoped monitors

Description. In this pattern, monitors are explicitly acquired and released without using `synchronized` blocks.

Rationale. The pattern is used in some situations to avoid deadlock, releasing a monitor temporarily, then reacquiring it.

Unsafe members. *monitorEnter*, *monitorExit*

Implementation. One usage of the pattern is to temporarily release monitor locks acquired in client code (e.g., through a synchronized block or method) and then to reenter the monitor before returning to the client. The *monitorExit* method is used to exit the synchronized block. Because monitors are reentrant, the pattern uses the method *Thread.holdsLock* implement a loop that repeatedly exits the monitor until the lock is no longer held. When reentering the monitor, *monitorEnter* is called the same number of types as *monitorExit* was called to release the lock.

Found in: 14 artifacts ***Used by:*** 7015 artifacts

Most used artifacts. *org.jboss.modules:jboss-modules*[41], *org.apache.cassandra:cassandra-all*[42], *org.gridgain:gridgain-core*[43]

Issues. Care must be taken to balance calls to *monitorEnter* and *monitorExit*, or else the lock might not be released or an `IllegalMonitorStateException` might be thrown.

Alternatives. One can extend the language to support non-lexically-scoped monitors.

6.8 Serialization/Deserialization

Description. In this pattern, *sun.misc.Unsafe* is used to persist and subsequently load objects to and from secondary memory dynamically. Serialization in Java is so important that it has a *Serializable* interface to automatically serialize objects that implement it. Although this kind of serialization is easy to use, it does not offer good performance and is inflexible. It is possible to implement serialization using the reflection API. This is also expensive in terms of performance. Therefore, fast serialization frameworks often use *Unsafe* to get and set fields of objects. Some of these projects use reflection to check if *sun.misc.Unsafe* is available, falling back on a slower implementation if not.

[36] `http://xircles.codehaus.org/projects/jsr166-mirror`

[37] `https://github.com/Netflix/servo`

[38] `http://groovy-lang.org/`

[39] `http://jodd.org/`

[40] `http://lmax-exchange.github.io/disruptor/`

[41] `http://www.jboss.org/`

[42] `http://cassandra.apache.org/`

[43] `http://www.gridgain.com/`

Rationale. De/serialization requires reading and writing fields to save and restore objects. Some of these fields may be final or private.
Unsafe members. *objectFieldOffset* and methods of the *Heap Get* and *Heap Put* groups
Implementation. Methods of *Heap Get* and *Heap Put* are used to read and write fields and array elements. Deserialization may use *allocateInstance* to create objects without invoking the constructor.
Found in: 32 artifacts ***Used by:*** 5689 artifacts
Most used artifacts. *com.hazelcast:hazelcast-all*, *com.esotericsoftware.kryo:kryo*[44], *com.thoughtworks.xstream:xstream*[45]
Issues. Using *Unsafe* for serialization and deserialization has many of the same issues as using *Unsafe* for updating final fields and for creating objects without invoking a constructor. Objects must not escape before being completely deserialized. Type safety can be violated by using methods of the *Heap Put* group. In addition, care must be taken when deserializing some data structures. For instance, hash tables that use *System.identityHashCode* or *Object.hashCode* may need to rehash objects on deserialization because the deserialized object might have a different hash code than the original serialized object.
Alternatives. Reflection can be used for accessing fields, more safely although less efficiently. Java's supports serialization of objects using *java.io.ObjectOutputStream* and related classes. These serialization features could be extended with support for user-defined serialization formats.

6.9 Foreign data access and object marshaling

Description. In this pattern *sun.misc.Unsafe* is used to share data between Java code and code written in another language, usually C or C++.
Rationale. This pattern is needed to efficiently pass data, especially structures and arrays, back and forth between Java and native code.
Unsafe members. Methods of the *Off-Heap* and *Off-Heap Get/Put* groups
Implementation. The methods of the *Off-Heap* group are used to access memory off the Java heap. Often a buffer is allocated using *allocateMemory*, which is then passed to the other language using JNI. Alternatively, the native code can allocate a buffer in a JNI method. The *Off-Heap Get* and *Off-Heap Put* methods are used to access the buffer.
Found in: 8 artifacts ***Used by:*** 3690 artifacts
Most used artifacts. *eu.stratosphere:stratosphere-core*[46], *com.github.jnr:jffi*[47], *org.python:jython*[48]
Issues. Use of *Unsafe* here is inherently not type-safe. Care must be taken especially with native pointers, which are represented as `long` values in Java code.
Alternatives. *java.nio.ByteBuffer* and related classes can be used for marshaling data instead of *Unsafe*.

6.10 Throw checked exceptions without being declared

Description. This pattern allows the programmer to throw checked exceptions without being declared in the method's `throws` clause.
Rationale. In testing and mocking frameworks, the pattern is used to circumvent declaring the exception to be thrown, which is often unknown. It is used in the Java Fork/Join framework to save the generic exception of a thread to be rethrown later.
Unsafe members. *throwException*
Implementation. The pattern is implemented using the *throwException* method.
Found in: 59 artifacts ***Used by:*** 3566 artifacts
Most used artifacts. *io.netty:netty-all*[49], *net.openhft:lang*[50], *ai.h2o:h2o-core*[51]
Issues. This method can violate the subtyping relation, because it is not expected for a method that does not declare an exception to actually throw it.
Alternatives. The issue can be avoided by not requiring `throws` declarations at all. Indeed, there is a long-running debate[52] about the software-engineering benefits of checked exceptions. C#, for instance, does not require that exceptions be declared in method signatures at all. One alternative proposed in a Stack Overflow discussion is to use Java generics instead.[53] Because of type erasure, a checked exception can be coerced unsafely into an unchecked exception and thrown.

6.11 Get the size of an object or an array

Description. This pattern uses *sun.misc.Unsafe* to estimate the size of an object or an array in memory.
Rationale. The object size can be useful for making manual memory management decisions. For instance, when implementing a cache, object size can be used to implement limit the cache size.
Unsafe members. *arrayBaseOffset*, *arrayIndexScale*, *objectFieldOffset*
Implementation. To compute the size of an array, add *arrayBaseOffset* and *arrayIndexScale* (for the given array base type) times the array length. For objects, use *objectFieldOffset* to compute the offset of the last instance field. In both cases, a VM-dependent fudge factor is added to account for the object header and for object alignment and padding.

[44] `https://github.com/EsotericSoftware/kryo`
[45] `http://xstream.codehaus.org/`
[46] `http://stratosphere.eu/`
[47] `https://github.com/jnr/jffi`
[48] `http://www.jython.org/`
[49] `http://netty.io/`
[50] `https://github.com/OpenHFT/Java-Lang`
[51] `https://github.com/h2oai/h2o-dev`
[52] `http://www.ibm.com/developerworks/library/j-jtp05254/`
[53] `http://stackoverflow.com/questions/11410042`

Found in: 4 artifacts ***Used by:*** 3003 artifacts
Most used artifacts. *net.sf.ehcache:ehcache*[54], *com.github.jbellis:jamm*[55], *org.openjdk.jol:jol-core*[56]
Issues. Object size is very implementation dependent. Accounting for the object header and alignment is requires adding VM-dependent constants for these parameters.
Alternatives. A sizeof feature could be introduced into the language or into the standard library to make the implementation portable.

6.12 Large arrays / off-heap data structures

Description. This pattern uses off-heap memory to create large arrays or data structures with manual memory management.
Rationale. Java's arrays are indexed by `int` and are thus limited to 2^{31} elements. Using *Unsafe*, larger buffers can be allocated outside the heap.
Unsafe members. *allocateMemory*, *freeMemory*, *setMemory*, *getInt*, *getLong*, *putInt*, *putLong*
Implementation. A block of memory is allocated with *allocateMemory* and then accessed using *Off-Heap Get* and *Off-Heap Put* methods. The block is freed with *freeMemory*.
Found in: 12 artifacts ***Used by:*** 487 artifacts
Most used artifacts. *org.neo4j:neo4j-primitive-collections*[57], *com.orientechnologies:orientdb-core*[58], *org.mapdb:mapdb*[59]
Issues. This pattern has all the issues of manual memory management: memory leaks, dangling pointers, double free, etc. One issue, mentioned on Stack Overflow is that the memory returned by *allocateMemory* is uninitialized and may be garbage.[60] Therefore, care must be taken to initialize allocated memory before use. The *Unsafe* method *setMemory* can be used for this purpose.
Alternatives. This functionality could be provided with a language feature or library.

6.13 Get memory page size

Description. *sun.misc.Unsafe* is used to determine the size of a page in memory.
Rationale. The page size is needed to allocate buffers or access memory by page. A common use case is to round up a buffer size, typically a *java.nio.ByteBuffer*, to the nearest page size. Hadoop uses the page size to track memory usage of cache files mapped directly into memory using *java.nio.MappedByteBuffer*. Another use is to process a buffer page-by-page. Some native libraries require or recommend allocating buffers on page-size boundaries.[61]
Unsafe members. *pageSize*
Implementation. Call *pageSize*.
Found in: 11 artifacts ***Used by:*** 359 artifacts
Most used artifacts. *org.apache.hadoop:hadoop-common*, *net.openhft:lang*, *org.xerial.larray:larray-mmap*[62]
Issues. Some platforms on which the JVM runs do not have virtual memory, so requesting the page size is non-portable.
Alternatives. This functionality could be added to the standard library, perhaps in the *java.nio* package.

6.14 Load class without security checks

Description. *sun.misc.Unsafe* is used to load a class from an array containing its bytecode. Unlike with the *ClassLoader* API, security checks are not performed.
Rationale. This pattern is useful for implementing lambdas, dynamic class generation, dynamic class rewriting. It is also useful in application frameworks that do not interact well with user-defined class loaders.
Unsafe members. *defineClass*
Implementation. The pattern is implemented using the *defineClass* method, which takes a byte array containing the bytecode of the class to load.
Found in: 21 artifacts ***Used by:*** 294 artifacts
Most used artifacts. *org.elasticsearch:elasticsearch*[63], *org.apache.geronimo.ext.openejb:openejb-core*[64], *net.openhft:lang*
Issues. The pattern violates the Java security model. Untrusted code could be introduced into the same protection domain as trusted code.
Alternatives. This feature could be added to the standard library, with a *SecurityManager* used to explicitly relax the Java security model.

7. Related Work

Oracle software engineer Paul Sandoz [28] performed a survey to study how Unsafe is used[65]. The survey consists of 7 questions that help to understand what pieces of *sun.misc.Unsafe* should be mainstreamed. We go beyond that survey to look at how *sun.misc.Unsafe* is used in the wild, by examining artifacts from the Maven Central software repository to analyze how and why *sun.misc.Unsafe* is being used. We first describe the related work about mining software repositories to understand a specific language feature. Then, we present the ongoing work to rectify the unsafe situation. Finally, we show where unsafe fits in the broader spectrum, i.e., how to do low-level coding in a high level language.

[54] `http://ehcache.org/`
[55] `https://github.com/jbellis/jamm`
[56] `http://openjdk.java.net/projects/code-tools/jol/`
[57] `http://neo4j.com/`
[58] `https://github.com/orientechnologies/orientdb`
[59] `http://www.mapdb.org/`
[60] `http://stackoverflow.com/questions/16723244`
[61] `http://stackoverflow.com/questions/19047584`
[62] `https://github.com/xerial/larray`
[63] `https://github.com/elastic/elasticsearch`
[64] `http://geronimo.apache.org/`
[65] `http://www.infoq.com/news/2014/02/Unsafe-Survey`

7.1 Mining repositories to assess language features

Several researchers have mined software repositories with the goal of analyzing and understanding if, how and when certain programming language features are being used.

Dyer et. al. [6] studied the adoption of Java language features over time. Richards et. al. [23] present an in-depth study on the `eval` function in JavaScript. Mayer et. al. [16] studied the impact of type systems on software development. Callaú et. al. [4] performed an empirical study to assess how much the dynamic and reflective features of Smalltalk are actually used in practice. Holkner and Harland [12] did a similar study on production-stage open source Python programs. Richards et. al.[24] also did a study on the dynamic behavior but applied to JavaScript programs. Gorla et. al. [9] mined a large set of Android applications, clustering applications by their description topics and identifying outliers in each cluster with respect to their API usage. Grechanik et. al.[11] also mined large scale software repositories to obtain several statistics on how source code is actually written.

7.2 Ongoing work to improve the Unsafe situation

To estimate how *sun.misc.Unsafe* is used, Sandoz [28] performed a survey on the OpenJDK mailing list. He describes several proposals to improve the situation of *sun.misc.Unsafe*, listed in Table 2.

Table 2. Use cases of *sun.misc.Unsafe* (slide 28 from [28])

Use Case	Feature
Enhanced atomic access	JEP 193 Enhanced Volatiles [15]
De/Serialization	JEP 187 Serialization 2.0
Reduce GC	Value types [25] JEP 189 Shenandoah: Low Pause GC [5]
Efficient memory layout	Value types Arrays 2.0 [26] Layouts [19]
Very Large Collections	Value types Arrays 2.0 & Layouts
Communicate across JVM boundary	Project Panama [27] JEP 191 FFI [18]

The work of Sandoz is essentially based on personal opinions of people, while we performed an analysis based on large-scale software repositories.

7.3 High-level language semantics for low-level coding

Oracle provides the *sun.misc.Unsafe* class for low-level programming, e.g, synchronization primitives, direct memory access methods, array manipulation and memory usage. Although the *sun.misc.Unsafe* class is not officially documented, there is literature based on it.

Korland et. al. [13] presented a Java STM framework, intended as a development platform for scalable concurrent applications and as a research tool for designing STM algorithms. They chose to use *sun.misc.Unsafe* to implement fast reflection, as it proved to be vastly more efficient than the standard Java reflection mechanisms. Pukall et. al. [21] introduced a runtime update approach based on Java that offers flexible dynamic software updates with minimal performance overhead. They used the `allocateInstance` method, because it eases the creation of instances even if the class has no default constructor. Gligoric et. al. [8] proposed a new approach to serialization/deserialization via code generation, using *sun.misc.Unsafe* to allocate instances and to set the fields. The Jikes RVM [1] is a Java Virtual Machine targeting researchers in runtime systems. It is a Java-in-Java virtual machine because is itself built in Java, a style of implementation termed meta-circular. The Jikes RVM provides an implementation of *sun.misc.Unsafe* with the *magic* framework. Frampton et. al. [7] proposed `org.vmmagic` to provide an escape hatch to low-level alternatives needed to build virtual machines; however, they require compiler support.

8. Conclusions

sun.misc.Unsafe is an API that was designed for limited use in system-level runtime library code. The Unsafe API is powerful, but dangerous. The improper use of Unsafe undermines Java's safety guarantees. We studied to what degree Unsafe usage has spread into third-party libraries, to what degree such third-party usage of Unsafe can impact existing Java code, and which Unsafe API features such third-party libraries actually use. We studied the questions and discussions developers have about Unsafe, and we identified common usage patterns. We thereby provided a basis for evolving the Unsafe API, the Java language, and the JVM by eliminating unused or abused unsafe features, and by providing safer alternatives for features that are used in meaningful ways. We hope this will help to make unsafe safer.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-source Research Community. *IBM Syst. J.*, 44(2):399–417, January 2005.

[2] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. Extracting structured data from natural language documents with island parsing. In *Proceedings of ASE 2011 (26th IEEE/ACM International Conference On Automated Software Engineering)*, pages 476–479, 2011.

[3] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[4] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features

of programming languages: The case of smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 23–32, New York, NY, USA, 2011. ACM.

[5] Roman Kennke Christine H. Flood. JEP 189: Shenandoah: An Ultra-Low-Pause-Time Garbage Collector. `http://openjdk.java.net/jeps/189`, 2014.

[6] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *36th International Conference on Software Engineering*, ICSE'14, pages 779–790, June 2014.

[7] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying Magic: High-level Low-level Programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[8] Milos Gligoric, Darko Marinov, and Sam Kamin. CoDeSe: Fast Deserialization via Code Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 298–308, New York, NY, USA, 2011. ACM.

[9] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035, New York, NY, USA, 2014. ACM.

[10] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[11] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 11:1–11:10, New York, NY, USA, 2010. ACM.

[12] Alex Holkner and James Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[13] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive Concurrency with Java STM. In *Communications of the ACM, Invited Review Paper*, page 19 pages, 2010.

[14] Eugene Kuleshov. Using the ASM framework to implement common Java bytecode transformation patterns. In *Conference on Aspect Oriented Software Development (AOSD): Industry Track*, 2007.

[15] Doug Lea. JEP 193: Enhanced Volatiles. `http://openjdk.java.net/jeps/193`, 2014.

[16] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 683–702, New York, NY, USA, 2012. ACM.

[17] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of WCRE 2001 (8th Working Conference on Reverse Engineering)*, pages 13–22. IEEE CS, 2001.

[18] Charles Oliver Nutter. JEP 191: Foreign Function Interface. `http://openjdk.java.net/jeps/191`, 2014.

[19] OpenJDK. Project Sumatra. `http://openjdk.java.net/projects/sumatra/`, 2013.

[20] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Stormed: Stack overflow ready made data. In *Proceedings of MSR 2015 (12th Working Conference on Mining Software Repositories)*, page to be published. ACM Press, 2015.

[21] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. JavAdaptor-Flexible runtime updates of Java applications. *Software: Practice and Experience*, 43(2):153–185, 2013.

[22] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 53–65, New York, NY, USA, 2009. ACM.

[23] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[25] John Rose, Brian Goetz, and Guy Steele. State of the Values. `http://cr.openjdk.java.net/~jrose/values/values-0.html`, 2014.

[26] John R. Rose. Arrays 2.0. `http://cr.openjdk.java.net/~jrose/pres/201207-Arrays-2.pdf`, 2012.

[27] John R. Rose. The isthmus in the VM. `https://blogs.oracle.com/jrose/entry/the_isthmus_in_the_vm`, 2014.

[28] Paul Sandoz. Safety not guaranteed: sun.misc.Unsafe and the quest for safe alternatives. `http://cr.openjdk.java.net/~psandoz/dv14-uk-paul-sandoz-unsafe-the-situation.pdf`, 2014. Oracle Inc. [Online; accessed 29-January-2015].

[29] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 9–17, New York, NY, USA, 2000. ACM.