If you’ve ported an application to Java 11, then you’re probably familiar with the following scary-sounding message or something like it:

```
WARNING: An illegal reflective access operation \nWARNING: An illegal reflective access operation \nWARNING: Illegal reflective access by ReflByt \nWARNING: Please consider reporting this to th \nWARNING: Use --illegal-access=warn to enable \nWARNING: All illegal access operations will b
```

While that warning is not as famous as "It is a period of civil war. Rebel spaceships, striking from a hidden base, have won their first victory against the evil Galactic Empire," these warnings provide a highly recognizable opening crawl to the logs of most modern Java applications. Many developers have even become desensitized to it—and that hides a problem that will potentially affect large numbers of applications soon.

**But what does the message mean?**

The error message above exists to warn of the slow, inexorable progress of a long-term project within the Java ecosystem: to
strongly encapsulate the internals of the Java runtime.

The encapsulation project was originally intended to form part of Java 8, but it was late for the train (as Mark Reinhold, chief architect of Java, wrote on his blog), and instead the first steps towards encapsulation were delivered in Java 9.

The need to encapsulate the runtime is fundamentally caused by Java's nature as an open programming environment. This long-ago design decision has had unintended consequences.

Note that the phrase “open programming environment” can be used in several different ways, so I need to be clear about the specifics of its meaning in this context. Specifically, it means the following:

- In Java 8 and before, you can call public methods on any public class you like, both directly and reflectively.
- After the Java module system arrived, these calls became subject to additional restrictions.

Those restrictions represented a fundamental change in the way Java access control works, but it might not seem like it. If you’re a Java developer who plays by the rules, you have never called an API in an internal package directly. However, you might well have used a library or a framework that does, so it’s good to understand what has changed behind the scenes.

**Encapsulation of direct access**

Here is an example of direct access to internal classes, similar to the way that some libraries might use it, using a piece of Java 8 code that extends an internal class to get access to a low-level URL canonicalizer.

A URL canonicalizer is a piece of code that takes a URL in one of the various forms permitted by the URL standard and converts it to a standard (canonical) form. The intent is that canonical URLs can act as a single source of truth for the location of content that can be accessed via multiple, different possible URLs.

By the way, the familiar `java.net.URL` class does some canonicalization, and for ordinary uses this should be sufficient because your code should never access internal classes directly. However, library authors may need to have better, more-specific control over how some aspects of canonicalization are handled.

The following code is for demonstration purposes only, to provide a concrete example to discuss access control:

```java
import sun.net.URLCanonicalizer;
public class MyURLHandler extends URLCanonicalizer {
    public boolean isSimple(String url) {
```
If you try to compile it using Java 8, `javac` warns that the code is accessing an internal API.

Despite the warnings, the compiler still allows the access. The result is a user class that is tightly coupled to the internal implementation of the JDK.

If enough developers abuse this openness, this leads to a situation in which it is difficult or impossible to make changes to the internals, because to do so would break deployed libraries and applications. This is one of the problems that modules were invented to solve.

Let's see what happens when you try to compile that code under Java 11.

```
$ javac src/ch02/MyURLHandler.java
src/ch02/MyURLHandler.java:3: error: package import sun.net.URLCanonicalizer;
^
src/ch02/MyURLHandler.java:5: warning: URLCan
public class MyURLHandler extends URLCanonical;
^
2 warnings
```

The messages above aren't warnings; they are errors. The form of the error message explicitly says that the `sun.net` package is now invisible.

These changes have the most impact for library developers. Most application developers shouldn't need to do anything more than upgrade to newer versions of the libraries they depend upon. In fact, this can even be a good thing, because more recent versions are probably more secure and performant and have more features.

**Encapsulation for modular JDKs**
The new reality is that when Java code is compiled with a modular JDK (all JDKs from version 9 onwards are modular), only methods on exported packages are accessible. It is no longer the case that a public method on a public class is automatically accessible to all code everywhere.

In other words, the platform can finally enforce the long-standing convention that in the JDK, a package that starts java or javax is a public API and everything else is internal-only.

In Java 8 and before, the convention is just that: a convention. Until the arrival of modules, there was no VM or class loading mechanism that enforced that, as I’ve shown.

The basic semantics of the Java module system close off the ability for libraries and applications to link directly to the JDK internals. This means that all applications that have upgraded from Java 8 are already safer—because they are guaranteed only to directly access the JDK via its public API.

However, when it has suited them, programmers have also coupled to the internals using reflection, and this aspect of encapsulation is more complex.

The treatment of reflection is at the heart of what the high-level log message means. To explore what the message means in detail, I’ll explain how modules impact reflection.

Java has supported reflection since almost the very beginning of its existence. Reflection allows you to access and work with types indirectly and, at runtime, in a way that does not require compile-time knowledge of those types. This flexibility is at the heart of many of the most popular Java frameworks and libraries.

As you have already seen, modules add a new concept to Java’s access control model—the idea of exporting a package—which declares a package to be part of the module’s API.

Reflection, however, is not direct linkage. The Java Reflection API has always had a huge encapsulation hole in the form of a method called setAccessible().

The setAccessible() method can be called on an object that represents, for example, a method on an unknown (at compile time) class. The method tells the JVM that it should skip access-control checks when trying to access or call the underlying (unknown at compile time) method. This method is very useful for framework designers, but it represents a major leakage of encapsulation safety. Modules needed to address this case as well.

**Reflective access to modularized code**

In general, modules can declare their reflective access policy as part of their module-info.class, which is done with the
The intent is that, by default, only exported packages can be accessed reflectively, but the designers of the modules system realized that sometimes developers want to give reflective (but not direct) access to certain packages.

Thus, the `opens` keyword allows a module to declare that a certain set of packages is available for reflective access, even if the packages are not part of the module’s public API. Developers can also specify fine-grained access by using the syntax `opens ... to ...` to allow a named set of packages to be opened reflectively to specific modules but not more generally.

I’ll make this concept more concrete with an example that uses an internal utility method to parse the bytecode name of a method. For the sake of this example, suppose I’m building a library and want it to build on all Java versions from 8 through 15 without the compiler errors shown earlier. To do that, I’ll refer to the class `sun.invoke.util.BytecodeName` indirectly via reflection.

Here is the reflective code. It compiles without error because it avoids the direct linkage.

```java
// Exception handling elided
Class<?> clz = Class.forName("sun.invoke.util
Method method = clz.getDeclaredMethod("parseB
Object res = method.invoke(null, "java/lang/S
System.out.println(Arrays.toString((Object[])
```

The output when the code is run under Java 8 is straightforward.

```
$ java ReflBytecodeName
[java, /, lang, /, String]
```

However, if the code is run under Java 11, the results are not as happy.

```
$ java ReflBytecodeName
WARNING: An illegal reflective access operati
WARNING: Illegal reflective access by ReflByt
WARNING: Please consider reporting this to th
WARNING: Use --illegal-access=warn to enable
WARNING: All illegal access operations will b
[java, /, lang, /, String]
```

These are, of course, the same warning messages highlighted at the start of this article.

The change in behavior arrived with Java 9 as part of the ecosystem moving towards proper encapsulation—which includes reflection. This change encodes the message that, in a future version of Java, the currently permissive usage will be
disallowed. The helpful message also calls out the command-line switch `--illegal-access` as the user-level control.

The expressed intent is for that command-line switch to eventually default to `deny` instead of `permit`, which has been the default setting in use since Java 9.

**What changed in Java 16?**

The change to reflective access obviously cannot happen overnight, because if the reflection switch were suddenly set to `deny`, huge swaths of the Java ecosystem would break and nobody would upgrade.

Java 9 was released in September 2017. Java 11 was released a year later, in September 2018. If the expressed intent was to eventually remove the backdoor reflective access to the JDK’s internals, it is legitimate to ask: “How long will this continue? How much warning is enough?”

Here’s the answer: In Java 16 (released in March 2021), the situation changed. If you rerun the test app under Java 16, you’ll see the following message:

```java
$ java ReflBytecodeName
java.lang.IllegalAccessException: class ReflB
at java.base/jdk.internal.reflect.Reflectio
at java.base/java.lang.reflect.AccessibleOb
at java.base/java.lang.reflect.Method.invok
at ReflBytecodeName.run(ReflBytecodeName.ja
at ReflBytecodeName.main(ReflBytecodeName.j
```

The previous warning is now an error. As of Java 16, the default permission for reflective access to the JDK internals has changed to `deny` (from previously permitting the access and issuing a warning). This is explained in detail by [JEP 396: Strongly encapsulate JDK internals by default](https://openjdk.java.net/jeps/396), which codifies the change.

This means that, unlike other recent Java feature releases, Java 16 has an additional barrier to adoption, whether you are moving from Java 8 or from Java 11.

(By the way, just to be clear, the package `sun.invoke.util` is not an officially supported or public package like `java.*`, so this example was focused on effectively demonstrating how the convention that “everything else is internal-only” is now rather more than a convention.)

As always in the Java ecosystem, the ramifications of a change like this are not limited to just what your application itself does explicitly but also to the behavior of the libraries you depend upon and all those libraries’ transitive dependencies.
This change means that, without user intervention (such as upgrading library versions), applications that depend on libraries that still leverage encapsulation-breaking access to the internals will now stop working.

In turn, this change means that before you and your team upgrade to Java 16 (and soon, to Java 17), you will need to ensure that the frameworks you depend upon have been certified to work with the new version of Java. This specific issue is being discussed on the Twitter hashtag #AllTestsGreenOnJDK16, led by Mark Reinhold.

**What's changing in Java 17?**

In Java 16, it is still possible to restore the situation that existed previously by using the `--illegal-access` command-line switch to allow general reflective access to the JDK internals. However, in Java 17, the situation changes again: This release removes that command-line switch. This topic is covered in detail by JEP 403: Strongly encapsulate JDK internals.

To summarize this somewhat complex situation, see Table 1, which shows the version changes in the permitted access to JDK internals.

<table>
<thead>
<tr>
<th>Java version</th>
<th>Direct access permitted?</th>
<th>Reflective access permitted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>11</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>16</td>
<td>N</td>
<td>N*</td>
</tr>
<tr>
<td>17</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

* In Java 16, reflective access can be restored with a command-line switch that activates the previous behavior.

**Table 1.** Permitted access to internals by Java version

**What about Unsafe?**

Unfortunately, for some developers, this closing off of reflective access has been conflated with a discussion of the famous (or possibly notorious) class `sun.misc.Unsafe`, which was discussed in my recent article, “The Unsafe class: Unsafe at any speed.”

In reality, Unsafe has long been recognized as a special case, and it is handled separately from the new access controls discussed in this article, but for the sake of completeness I’ll explain how strong encapsulation relates to Unsafe and its usage.

In Java 9, the core runtime, rt.jar, was split up and, in particular, the actual implementation code of `sun.misc.Unsafe` (and some related code referred to as critical internal APIs, as defined in JEP 260: Encapsulate most internal APIs), was moved out of the core and into a separate
module, jdk.unsupported. The naming is quite deliberate. The use of “jdk” rather than “java” as a prefix warns developers that by using these APIs, they are consciously choosing to delve into the internals, and “unsupported” speaks for itself.

(Note: jdk.unsupported is visible to code on the classpath by default, so sun.misc.Unsafe is, in practice, just as available in Java 9+ as it was in Java 8.)

The jdk.unsupported module is intended to provide a bridge for library developers, allowing them to migrate to fully supported APIs over time, without creating a hard cutoff and subsequent cliff.

Here’s a very brief look at the current implementation of this bridge in jdk.unsupported. While developers cannot rely upon this implementation, it provides an interesting example of a real-world code migration technique that may be a useful reference for anyone thinking of refactoring code to modules.

For sun.misc.Unsafe, the implementation currently works like this.

```java
public final class Unsafe {
    private Unsafe() {}
    private static final Unsafe theUnsafe = new jdk.internal.misc.Unsafe;
    // ...
}
```

Calls are forwarded to the real implementation of Unsafe, which lives in jdk.internal.misc in the protected part of the java.base module, as in the following:

```java
public int getInt(Object o, long offset) {
    return theInternalUnsafe.getInt(o, of);
}
```

Leaving aside the details of the implementation, ordinary Java applications cannot directly access the Unsafe object (nor should they try). This code

```java
Unsafe unsafe = Unsafe.getUnsafe();
```

will compile but will throw a SecurityException at runtime—and this is as true in Java 11 as it is in Java 8. Only trusted code inside the JDK can get direct access to the Unsafe object.
This is by design. *Unsafe* is potentially very dangerous because it provides a way to do certain things that are otherwise impossible in the Java platform. One goal of the JDK developers at Oracle is to mitigate this danger by creating official, supported public APIs to provide capabilities that are currently supplied by *Unsafe*.

Two great examples of this mitigation project are

- Hidden classes (delivered in Java 15), which replaced *Unsafe::defineAnonymousClass*, as mentioned in my article on *Unsafe*
- Off-heap memory access, which can now be achieved using the now-incubating JEP 412: Foreign function and memory API

The goal is that, over time, neither application programmers nor library developers should need to use the methods that *Unsafe* provides. However, until that future arrives, the well-established reflective tricks used by library developers to get hold of a reference to an *Unsafe* object still work, for example

```
Field f = Unsafe.class.getDeclaredField("theU
f.setAccessible(true);
Unsafe unsafe = (Unsafe) f.get(null);
```

This code still works in Java 16, and it will continue to work in Java 17, because the *jdk.unsupported* module is declared as both *exports* and *opens*:

```
module jdk.unsupported {
  exports com.sun.nio.file;
  exports sun.misc;
  exports sun.reflect;

  opens sun.misc;
  opens sun.reflect;
}
```

The code is, therefore, unaffected by the changes to strong encapsulation that formed part of these releases.

Finally, although there are some capabilities that are still not addressed by fully supported replacements, the amount of code that is present in *jdk.unsupported* is now fairly small.

To get some indication of how little of that old code is actually left, you can download a copy of the current OpenJDK source code from GitHub. If you navigate to the directory containing the source for *jdk.unsupported* (*jdk/src/jdk.unsupported/share/classes*), doing a quick bit of Linux shell scripting can provide an idea of how much code remains.
For those unfamiliar with the Linux `wc` command, the first column gives the number of lines of code in each source file. There are about 35 lines of license information at the start of each file. This means that, with the exception of `sun.misc.Unsafe`, these files are really very small.

Counting lines of code is, of course, an imperfect measure for the complexity of the tasks that remain to fully remove these files, but it does provide a simple metric that can be tracked from release to release.

The use of an unsupported module is obviously not ideal, but it’s clear that, over time, replacements for these remaining pieces should emerge and the `jdk.unsupported` module will shrink and eventually disappear.

**Conclusion**

Java 16 and Java 17 are providing a foundation of enhanced encapsulation that will provide a safer and simpler future for both application developers and JDK internals programmers alike.

[The second edition of *The Well-Grounded Java Developer*, by Ben Evans, Jason Clark, and Martijn Verburg, will be published Spring 2022. -Ed.]

**Dig deeper**

- The Unsafe class: Unsafe at any speed
- The role of preview features in Java 14, Java 15, Java 16, and beyond
- JEP 260: Encapsulate most internal APIs
- JEP 396: Strongly encapsulate JDK internals by default
- JEP 403: Strongly encapsulate JDK internals
- JEP 412: Foreign function and memory API

**Ben Evans**

Ben Evans (@kittylyst) is a Java Champion and Principal Engineer at New Relic. He has written five books on programming, including *Optimizing Java* (O'Reilly).
Previously he was a founder of jClarity (acquired by Microsoft) and a member of the Java SE/EE Executive Committee.