JEP 396: Strongly Encapsulate JDK Internals by Default

Authors	Alex Buckley, Mark Reinhold
Owner	Mark Reinhold
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	16
Discussion	jigsaw dash dev at openjdk dot java dot net
Effort	S
Duration	S
Relates to	JEP 403: Strongly Encapsulate JDK Internals
Reviewed by	Alan Bateman, Chris Hegarty, Mandy Chung
Endorsed by	Brian Goetz
Created	2020/10/23 19:41
Updated	2023/04/13 16:11
Issue	8255363

Summary

Strongly encapsulate all internal elements of the JDK by default, except for critical internal APIs such as `sun.misc.Unsafe`. Allow end users to choose the relaxed strong encapsulation that has been the default since JDK 9.

Goals

- Continue to improve the security and maintainability of the JDK, which is one of the primary goals of Project Jigsaw.
- Encourage developers to migrate from using internal elements to using standard APIs, so that both they and their users can upgrade without fuss to future Java releases.

Non-Goals

- It is not a goal to remove, encapsulate, or modify any critical internal APIs of the JDK for which standard replacements do not yet exist. **This means that `sun.misc.Unsafe` will remain available**.
- It is not a goal to define new standard APIs to replace internal elements for which standard replacements do not yet exist, though such APIs could be suggested in response to this JEP.

Motivation

Over the years the developers of various libraries, frameworks, tools, and applications have used internal elements of the JDK in ways that compromise both security and maintainability. In particular:

- Some non-`public` classes, methods, and fields of `java.*` packages define privileged operations such as the ability to define a new class in a specific class loader, while others convey sensitive data such as cryptographic keys. These elements are internal to the JDK, despite being in `java.*` packages. The use of these internal elements by external code, via reflection, puts the security of the platform at risk.
- All classes, methods, and fields of `sun.*` packages are internal APIs of the JDK. Some classes, methods, and fields of `com.sun.*`, `jdk.*`, and `org.*` packages are also internal APIs. These APIs were never standard, never supported, and never intended for external use. The use of these internal elements by external code is an ongoing maintenance burden. Time and effort spent preserving these APIs, so as not to break existing code, could be better spent moving the platform forward.

In Java 9, we improved both the security and the maintainability of the JDK by leveraging modules to limit access to its internal elements. Modules provide *strong encapsulation*, which means that

- Code outside of a module can only access the `public` and `protected` elements of the packages exported by that module, and
- `protected` elements can, further, only be accessed from subclasses of the classes that define them.

Strong encapsulation applies at both compile time and run time, including when compiled code attempts to access elements via reflection at run time. The non-`public` elements of exported packages, and all elements of unexported packages, are said to be *strongly encapsulated*.

In JDK 9 and later releases we strongly encapsulated all new internal elements, thereby limiting access to them. As an aid to migration, however, we deliberately chose not to strongly encapsulate, at run time, the content of packages that existed in JDK 8. Library and application code on the class path could thus continue to use reflection to access the non-`public` elements of `java.*` packages, and all elements of `sun.*` and other internal packages, for packages that existed in JDK 8. This arrangement is called *relaxed strong encapsulation*.

We released JDK 9 in September 2017. Most of the commonly-used internal elements of the JDK now have standard replacements. Developers have had over three years in which to migrate away from internal elements of the JDK to standard

APIs such as `java.lang.invoke.MethodHandles.Lookup::defineClass`, `java.util.Base64`, and `java.lang.ref.Cleaner`. Many library, framework, and tool maintainers have completed that migration and released updated versions of their components. We are now ready to take the next step toward the strong encapsulation of all internal elements of the JDK — except for critical internal APIs such as `sun.misc.Unsafe` — as originally planned in Project Jigsaw.

Description

Relaxed strong encapsulation is controlled by the launcher option `--illegal-access`. This option, introduced by JEP 261, is provocatively named in order to discourage its use. It presently works as follows:

- `--illegal-access=permit` arranges for every package that existed in JDK 8 to be open to code in unnamed modules. Code on the class path can thus continue to use reflection to access the non-public elements of `java.*` packages, and all elements of `sun.*` and other internal packages, for packages that existed in JDK 8. The first reflective-access operation to any such element causes a warning to be issued, but no warnings are issued after that point.

  This mode has been the default since JDK 9.

- `--illegal-access=warn` is identical to `permit` except that a warning message is issued for every illegal reflective-access operation.

- `--illegal-access=debug` is identical to `warn` except that both a warning message and a stack trace are issued for every illegal reflective-access operation.

- `--illegal-access=deny` disables all illegal-access operations except for those enabled by other command-line options, *e.g.*, `--add-opens`.

As the next step toward strongly encapsulating all internal elements of the JDK, we propose to change the default mode of the `--illegal-access` option from `permit` to deny. With this change, packages that existed in JDK 8 and do not contain critical internal APIs will no longer be open by default; a complete list is available here. **The `sun.misc` package will still be exported by the `jdk.unsupported` module, and will still be accessible via reflection.**

We will also revise the related text in the Java Platform Specification to disallow the opening of any package by default in any Java Platform Implementation, unless that package is explicitly declared to be open in the declaration of its containing module.

The `permit`, `warn`, and `debug` modes of the `--illegal-access` option will continue to work. These modes allow end users to choose relaxed strong encapsulation if they wish.

We expect a future JEP to remove the `--illegal-access` option entirely. At that point it will not be possible to open all of the JDK 8 packages via a single command-line option. It will still be possible to use the `--add-opens` command-line option, or the `Add-Opens` JAR-file attribute, to open specific packages.

To prepare for the eventual removal of the `--illegal-access` option we will deprecate it for removal as part of this JEP. As a consequence, specifying that option to the `java` launcher will cause a deprecation warning to be issued.

Risks and Assumptions

The primary risk of this proposal is that existing Java code will fail to run. The kinds of code that will fail include, but are not limited, to:

- Frameworks that use the `protected defineClass` methods of `java.lang.ClassLoader` in order to define new classes in existing class loaders. Such frameworks should instead use `java.lang.invoke.MethodHandles.Lookup::defineClass`, which has been available since JDK 9.

- Code that uses the `sun.util.calendar.ZoneInfo` class to manipulate time-zone information. Such code should instead use the `java.time` API, available since JDK 8.

- Code that uses the `com.sun.rowset` package to process SQL row sets. Such code should instead use the `javax.sql.rowset` package, available since JDK 7.

- Tools that use the `com.sun.tools.javac.*` packages to process source code. Such tools should instead use the `javax.tools`, `javax.lang.model`, and `com.sun.source.*` APIs, available since JDK 6.

- Code that uses the `sun.security.tools.keytool.CertAndKeyGen` class to generate self-signed certificates. There is not yet a standard API for this functionality (though a request has been submitted); in the mean time, developers can use existing third-party libraries that include this functionality.

- Code that uses the JDK's internal copy of the Xerces XML processor. Such code should instead use a standalone copy of the Xerces library, available from Maven Central.

- Code that uses the JDK's internal version of the ASM bytecode library. Such code should instead use a standalone copy of the ASM library, available from Maven Central.

We encourage all developers to:

- Use the `jdeps` tool to identify code that depends upon internal elements of the JDK.
  - When standard replacements are available, switch to using those.
  - Otherwise, we welcome strong cases for new standard APIs on the Project Jigsaw mailing list. Please understand, however, that we are unlikely to define new standard APIs for internal elements that are not broadly used.
- Use an existing release, such as JDK 11, to test existing code with `--illegal-access=warn` to identify any internal elements accessed via reflection, then use `--illegal-access=debug` to pinpoint the errant code, and then finally test with `--illegal-access=deny`.

Secondary risks

- An existing application may fail to run not because the application itself makes use of internal APIs, but because the application uses libraries or frameworks which do so. If you maintain such an application then we recommend that you update to the latest versions of the components upon which your application depends. If those components have not yet been updated to remove dependencies upon internal elements then we suggest that you urge their maintainers to do so, or perhaps consider doing that work yourself and submitting a patch.
- The maintainers of some libraries, frameworks, and tools have been telling application developers that illegal reflective-access warnings can safely be ignored when using JDK 9 and later. This causes tension with application developers who always use the very latest JDK release and realize that the components upon which they depend will break as soon as the JDK's internal elements are strongly encapsulated by default. For these application developers, downgrading to JDK 8 or not moving to the latest release is not a viable approach.

Examples of the impact of this change

- Code successfully compiled with earlier releases that directly accesses internal APIs of the JDK will no longer work by default. For example,

  ```
  System.out.println(sun.security.util.SecurityConstants.ALL_PERMISSION);
  ```

  will fail with an exception of the form

  ```
  Exception in thread "main" java.lang.IllegalAccessError: class Test
    (in unnamed module @0x5e481248) cannot access class
    sun.security.util.SecurityConstants (in module java.base) because
    module java.base does not export sun.security.util to unnamed
    module @0x5e481248
  ```

- Code that uses reflection to access `private` fields of exported `java.*` APIs will no longer work by default. For example,

  ```
  var ks = java.security.KeyStore.getInstance("jceks");
  var f = ks.getClass().getDeclaredField("keyStoreSpi");
  f.setAccessible(true);
  ```

  will fail with an exception of the form

  ```
  Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
    Unable to make field private java.security.KeyStoreSpi
    java.security.KeyStore.keyStoreSpi accessible: module java.base does
    not "opens java.security" to unnamed module @6e2c634b
  ```

- Code that uses reflection to invoke `protected` methods of exported `java.*` APIs will no longer work by default. For example,

  ```
  var dc = ClassLoader.class.getDeclaredMethod("defineClass",
                                               String.class,
                                               byte[].class,
                                               int.class,
                                               int.class);
  dc.setAccessible(true);
  ```

  will fail with an exception of the form

  ```
  Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
    Unable to make protected final java.lang.Class
    java.lang.ClassLoader.defineClass(java.lang.String,byte[],int,int)
    throws java.lang.ClassFormatError accessible: module java.base does
    not "opens java.lang" to unnamed module @5e481248
  ```