JEP 390: Warnings for Value-Based Classes

Owner	Dan Smith
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	16
Discussion	valhalla dash dev at openjdk dot java dot net
Effort	S
Duration	XS
Reviewed by	Brian Goetz
Endorsed by	Brian Goetz
Created	2020/07/08 21:39
Updated	2021/08/30 15:42
Issue	8249100

Summary

Designate the primitive wrapper classes as *value-based* and deprecate their constructors for removal, prompting new deprecation warnings. Provide warnings about improper attempts to synchronize on instances of any value-based classes in the Java Platform.

Motivation

The Valhalla Project is pursuing a significant enhancement to the Java programming model in the form of *primitive classes*. Such classes declare their instances to be identity-free and capable of inline or flattened representations, where instances can be copied freely between memory locations and encoded using solely the values of the instances' fields.

The design and implementation of primitive classes is sufficiently mature that we can confidently anticipate migrating certain classes of the Java Platform to become primitive classes in a future release.

Candidates for migration are informally designated as *value-based classes* in the API specifications. Broadly, this means that they encode immutable objects whose identity is unimportant to the behavior of the class, and that they don't provide instance creation mechanisms, such as public constructors, that promise a unique identity with each call.

The primitive wrapper classes (`java.lang.Integer`, `java.lang.Double`, etc.) are also intended to become primitive classes. These classes satisfy most of the requirements to be designated as value-based, with the exception that they expose deprecated (since Java 9) `public` constructors. With some adjustments to the definition, they can be considered value-based classes, too.

Clients of the value-based classes will generally be unaffected by primitive class migration, except where they violate recommendations for usage of these classes. In particular, when running on a future Java release in which the migration has occurred:

1. Instances of these classes that are equal (per `equals`) may also be considered identical (per `==`), potentially breaking programs that rely on a `!=` result for correct behavior.
2. Attempts to create wrapper class instances with `new Integer`, `new Double`, etc., rather than implicit boxing or calls to the `valueOf` factory methods, will produce `LinkageErrors`.
3. Attempts to synchronize on instances of these classes will produce exceptions.

These changes may be inconvenient for some, but the workarounds are straightforward: if you need an identity, use a different class—often one you define yourself, but `Object` or `AtomicReference` may also be suitable. The benefits of migrating to primitive classes—better performance, reliable equality semantics, unifying primitives and classes—will be well worth the inconvenience.

(1) has already been discouraged by avoiding promises about unique identities in the value-based classes' factory methods. There is not a practical way to automatically detect programs that ignore these specifications and rely on current implementation behavior, but we expect such cases to be rare.

We can discourage (2) by deprecating the wrapper class constructors for removal, which will amplify the warnings that occur when compiling calls to these constructors. A significant portion of existing Java projects (perhaps 1%-10% of them) call the wrapper class constructors, though in many cases they intend only to run on pre-9 Java releases. Many popular open-source projects have already responded to the deprecation warnings of Java 9 by removing wrapper constructor calls from their sources, and we can expect many more to do so, given the heightened urgency of "deprecated for removal" warnings. Additional features to mitigate this problem are described in the *Dependencies* section.

We can discourage (3) by implementing warnings, at compile time and run time, to inform programmers that their synchronization operations will not work in a future release.

Description

The primitive wrapper classes in `java.lang` (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, and `Character`) have been designated as value-based. The description of *value-based classes* has been updated to allow for deprecated constructors and interning factories, and to better align with the requirements for primitive class migration (for example, a value-based class should not inherit any instance fields).

To discourage misuse of value-based class instances:

- The primitive wrapper class constructors, originally deprecated in Java 9, have been deprecated for removal. Wherever the constructors are called in source, `javac` by default produces `removal` warnings. The `jdeprscan` tool may be used to identify usage of deprecated APIs in binaries.
- `javac` implements a new warning category, `synchronization`, which identifies usages of the `synchronized` statement with an operand of a value-based class type, or of a type whose subtypes are all specified to be value-based. The warning category is turned on by default, and can be manually selected with `-Xlint:synchronization`.
- HotSpot implements runtime detection of `monitorenter` occurring on a value-based class instance. The command-line option `-XX:DiagnoseSyncOnValueBasedClasses=1` will treat the operation as a fatal error. The command-line option `-XX:DiagnoseSyncOnValueBasedClasses=2` will turn on logging, both via the console and via JDK Flight Recorder events.

Compile-time synchronization warnings depend on static typing, while runtime warnings can respond to synchronization on non-value-based class and interface types, like `Object`.

For example:

```
Double d = 20.0;
synchronized (d) { ... } // javac warning & HotSpot warning
Object o = d;
synchronized (o) { ... } // HotSpot warning
```

The `monitorexit` bytecode and the `Object` methods `wait`, `notify`, and `notifyAll` have always thrown an `IllegalMonitorStateException` if invoked outside of a `synchronized` statement or method. There is thus no need for warnings about these operations.

Identifying value-based classes

Within the JDK, the `@jdk.internal.ValueBased` annotation is used to signal to `javac` and HotSpot that a class is value-based, or that an abstract class or interface requires value-based subclasses.

`@ValueBased` is applied to the following declarations in the Java Platform API and the JDK:

- The primitive wrapper classes in `java.lang`;
- The class `java.lang.Runtime.Version`;
- The "optional" classes in `java.util`: `Optional`, `OptionalInt`, `OptionalLong`, and `OptionalDouble`;
- Many classes in the `java.time` API: `Instant`, `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `ZoneId`, `OffsetTime`, `OffsetDateTime`, `ZoneOffset`, `Duration`, `Period`, `Year`, `YearMonth`, and `MonthDay`, and, in `java.time.chrono`: `MinguoDate`, `HijrahDate`, `JapaneseDate`, and `ThaiBuddhistDate`;
- The interface `java.lang.ProcessHandle` and its implementation classes;
- The implementation classes of the collection factories in `java.util`: `List.of`, `List.copyOf`, `Set.of`, `Set.copyOf`, `Map.of`, `Map.copyOf`, `Map.ofEntries`, and `Map.entry`.

Wherever the annotation is applied to an abstract class or interface, it is also applied to all subclasses in the JDK.

Some classes and interfaces in `java.lang.constant` and `jdk.incubator.foreign` have claimed to be value-based, but do not meet the revised requirements—for example, they inherit instance fields—and so cannot be migrated to be primitive classes. In this case, it's no longer appropriate to describe these as value-based classes, and their specifications have been revised.

Scope of changes

Java SE: This JEP modifies Java SE by refining the specifications of the primitive wrapper classes, existing value-based classes, and related interfaces and factory methods. It also deprecates for removal the primitive wrapper class constructors. It does *not* make any changes to the Java Language or Java Virtual Machine specifications.

JDK: In the JDK, this JEP also adds new warning and logging capabilities to `javac` and HotSpot. And it defines the annotation `@jdk.internal.ValueBased` and applies it to a number of JDK classes.

Alternatives

We could abandon efforts to migrate these classes to be primitive classes. However, there are significant benefits that developers will enjoy when we complete the migration, and the relative impact on developers who depend on problematic behavior is small.

It would be possible to supplement the compile-time deprecation warnings with run-time warnings. This is left as future work for another JEP (see below).

There may be other classes that could be migrated to be primitive classes, including API classes and classes generated by features like `java.lang.invoke.LambdaMetafactory`. This JEP limits itself to the wrapper classes and classes already designated as value-based. Again, additional warnings could be introduced as future work.

Dependencies

Migrating value-based classes to be primitive classes will require a reasonable amount of lead time with these warnings in place. Most significantly, a JEP to make the wrapper classes primitive classes cannot proceed until some number of releases after completion of this JEP.

Another prerequisite to making the wrapper classes primitive classes is sufficient tooling to identify and work around legacy uses of their constructors. Two followup features will be investigated in separate JEPs:

- HotSpot runtime warnings for usages of deprecated APIs, including the wrapper class constructors. This would complement the warnings produced by `javac` and `jdeprscan`.
- Tooling to support execution of binaries that are unable to update their usages of wrapper class constructors. This might, for example, give programmers the option to rewrite their bytecodes to make use of the `valueOf` factory methods.