JEP 12: Preview Features

Owner	Alex Buckley
Type	Informational
Scope	SE
Status	Active
Discussion	jdk dash dev at openjdk dot java dot net
Effort	M
Duration	M
Reviewed by	Alan Bateman, Brian Goetz, Mark Reinhold
Endorsed by	Mark Reinhold
Created	2018/01/19 01:27
Updated	2023/09/21 21:14
Issue	8195734

Summary

A *preview feature* is a new feature of the Java language, Java Virtual Machine, or Java SE API that is fully specified, fully implemented, and yet impermanent. It is available in a JDK feature release to provoke developer feedback based on real world use; this may lead to it becoming permanent in a future Java SE Platform.

Goals

- Allow Java platform developers to communicate whether a new feature is "coming to Java" in approximately its current form within the next 12 months.
- Define a model for partitioning new language, VM, and API features based on whether they are *permanent* or *impermanent* in the Java SE Platform (that is, whether they will exist in the same form for all future releases, or will exist in a different form or not at all).
- Communicate the intent that code which uses preview features from an older release of the Java SE Platform will not necessarily compile or run on a newer release.
- Outline the relationship between preview features on the one hand, and "experimental" (HotSpot) / "incubating" (API) features on the other hand.

Non-Goals

- It is not necessary for all new language, VM, and API features to be available initially as preview features.
- It is not necessary for this JEP to mandate specific mechanisms for collecting and evaluating developer feedback about preview features.
- Nothing in this JEP should be interpreted as encouraging or allowing fragmentation of the Java SE Platform.

History

- This JEP was introduced in 2018, circa JDK 12, to allow two kinds of *preview feature* in the Java SE Platform: preview language features and preview VM features. Preview features are present in all Java compilers and JVM implementations but disabled by default. The JEP outlined design properties that are required of all preview features, and gave rules for how developers enable and use preview features at compile time and run time. The JEP also discussed how preview features are represented in the JEP Process that governs all JDK feature development.
- In 2019, circa JDK 13, this JEP was enhanced to recognize that preview language features are often co-developed with new APIs in the `java.base` module, principally new classes in `java.lang`. This led to a taxonomy of co-developed APIs: "essential", "reflective", and "convenient".
- In 2020, circa JDK 15, this JEP was enhanced to allow a third kind of preview feature in the Java SE Platform: preview APIs. Preview APIs subsumed the idea of APIs co-developed with preview language features, and further allowed the definition of Java SE APIs unrelated to other preview features; this led to a taxonomy of "essential", "reflective", "convenient", and "standalone" preview APIs. The JEP framed preview APIs so that they share the same properties as preview language and VM features, and are enabled by developers in exactly the same way. However, the JEP also recognized differences between the four categories of preview API, and between preview APIs and preview language features, that motivated more relaxed rules for using preview APIs than for using preview language and VM features.
- In 2023, circa JDK 20, this JEP was modified to set expectations that (1) previewing an API can lead to change that is qualitatively different than the change seen for a language or VM feature, and (2) previewing may take longer than 12 months under certain circumstances. For background, see the retrospective Preview Features: A Look Back, and A Look Ahead.

Motivation

The Java SE Platform has global reach, so the cost of a mistake in the design of a Java language feature, JVM feature, or Java SE API is high. A mistake may be a hard technical error (such as a flaw in Java's type system), a soft usability problem (such as a surprising interaction with an older feature), or a poor architectural choice (such as one that forecloses on directions for future features).

To build confidence in the correctness and completeness of a new feature -- whether in the Java language, the JVM, or the Java SE API -- it is desirable for the

feature to enjoy a period of broad exposure *after* its specification and implementation are stable but *before* it achieves final and permanent status in the Java SE Platform. To achieve the broadest possible exposure, and to maximize the likelihood of swift feedback, the feature may be included in a JDK feature release on a *preview* basis. Previewing a feature in the JDK will encourage tool vendors to build good support for the feature before the bulk of Java developers use it in production.

Over the six-month course of the JDK feature release, and perhaps the following release too, the feature's "real world" strengths and weaknesses will be evaluated to decide if it has a long-term role in the Java SE Platform. Ultimately, the feature will either be granted final and permanent status (with or without refinements) or be removed.

Description

A *preview feature* is:

- a new feature of the Java language ("preview language feature"), or
- a new feature of the JVM ("preview VM feature"), or
- a new module, package, class, interface, method, constructor, field, or enum constant in the `java.*` or `javax.*` namespace ("preview API")

whose design, specification, and implementation are complete, but which would benefit from a period of broad exposure and evaluation before either achieving final and permanent status in the Java SE Platform or else being refined or removed.

By "complete", we do not mean "100% finished", since that would imply feedback is pointless. Instead, we mean the preview feature meets two criteria:

1. (Readiness) The preview feature has a *high probability* of being 100% finished *within 12 months*. This timeline reflects our experience that two rounds of previewing is the norm, i.e., preview in Java $N and $N+1 then final in $N+2. For APIs that have exceptionally large surface areas or engage deeply with the JVM, and for language features that integrate with other language features as a matter of necessity, we anticipate additional rounds of feedback and revision, as such features will underpin the Java ecosystem for decades to come.
2. (Stability) The preview feature could credibly achieve final and permanent status with no further changes. This implies an extremely high degree of confidence in the concepts which underpin the feature, but does not completely rule out making "surface level" changes in response to feedback. (This is especially relevant for an API, which necessarily exposes concepts via a larger surface area than a language feature; a semantically stable API might undergo considerable syntactic polishing (e.g., renaming classes, adding helper methods) during its preview period, before achieving final status.)

The key properties of a preview feature are:

1. *High quality.* A preview feature must display the same level of technical excellence and finesse as a final and permanent feature of the Java SE Platform. For example, a preview language feature must respect traditional Java principles such as readability and compatibility, and it must receive appropriate treatment in the reflective and debugging APIs of the Java SE Platform.
2. *Not experimental.* A preview feature must not be experimental, risky, incomplete, or unstable. An experimental feature must not be previewed in a JDK feature release, but rather be iterated and stabilized in its own project, which in turn produces binaries that are clearly distinguished from binaries of the JDK Project. For the purpose of comparison, if an experimental feature in HotSpot is considered 25% "done", then a preview feature in Java SE should be at least 95% "done". To make a further comparison, the level of completeness and stability expected for a preview API is considerably higher than the level expected for an incubating API.
3. *Universally available.* The Umbrella JSR for the Java SE $N Platform enumerates the preview features of the platform. The desire for feedback and the expectation of quality means that these features are not "optional"; they must all be supported in full by every implementation of Java SE $N. In particular, all preview features must be disabled by default in a Java SE implementation, and an implementation must offer a mechanism to enable all preview features. An implementation must not allow preview features to be enabled on an individual basis, since all preview features have equal status in the Java SE Platform.

As a general rule, if a question arises about some property of a preview feature and the question is not answered explicitly in this JEP, then the implicit answer is "No different for a preview feature than for a final and permanent feature."

Design of Preview Features

There are no constraints on the shape of a preview feature:

- A preview language feature may add declaration, statement, expression, and literal forms to the syntax of the Java language; it may modify the static semantics (typing) of pre-existing declarations, statements, expressions, and literals; and it may modify the dynamic semantics (execution) of pre-existing statements or expressions.
- A preview VM feature may add and modify elements of the `class` file format; it may alter the rules for loading, linking, and initializing classes; and it may extend and modify the JVM instruction set. Preview VM features are different from the low-level features of the HotSpot JVM implementation which are configured via `java -X` or `java -XX`. Preview VM features are

also distinct from "experimental" HotSpot features, which are early versions of low-level features that need to be explicitly unlocked in HotSpot at run time. (For example, when HotSpot's Z Garbage Collector was experimental in JDK 11, it was enabled via `java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC`.)

- A preview API may add `public` members to classes and interfaces in `java.*` and `javax.*`; add `public` classes and interfaces to `java.*` and `javax.*`; add and export packages in the `java.*` and `javax.*` namespaces; and add modules in the `java.*` namespace. A preview API may modify the narrative specifications (though not the signatures) of pre-existing methods, fields, classes, interfaces, packages, and modules. A preview API typically resides in the `java.base` module, but may reside additionally or exclusively in other `java.*` modules, including those introduced just for the preview API.
- The Java SE Platform includes non-Java APIs, such as JNI and JVM TI, and language independent protocols, such as JDWP and Java Object Serialization. Non-Java APIs and language-independent protocols may be extended to support a preview feature. Such extensions are considered to be preview APIs in their own right. For example, JNI functions may be added as a preview API that lets C code interrogate run-time artifacts corresponding to a preview API in `java.*`. As another example, the Java Object Serialization protocol may be modified to give special treatment to objects whose classes use preview language or VM features.
- The design of a preview feature in the Java SE Platform may sometimes require evolution of well-known utility APIs that are not part of the Java SE Platform but are nevertheless present in most Java runtimes. These APIs are principally `com.sun.*` packages exported by `jdk.*` modules. They include the Compiler Tree API, the HTTP Server API, and the Java Debug Interface. These "supported" `com.sun` APIs may be extended to support a preview feature. Such extensions are considered to be preview APIs in their own right. For example, methods may be added as a preview API to classes in the Java Debug Interface, to support debugging of a preview VM feature.

There is no requirement that a preview feature adopts a striking syntactic form to highlight its non-final status. On the contrary, a preview feature must adopt the sober syntactic form that is intended for its final and permanent version. For example, a preview language feature must not break convention by introducing an upper-case keyword, and a preview API must not add temporary packages to `java.*`. (This stands in contrast to incubating APIs, which use the `jdk.incubator` namespace to highlight the non-final status of their packages and modules.)

There are no constraints on the size of a preview feature. A preview feature may be any size, from very large to very small.

> As an example of a small preview language feature, consider the changing role of the _ character ("underscore"). It was redefined from an identifier to a keyword in order to reserve it for future language features. To provide a migration period for existing code, the redefinition was spread over two JDK releases: in JDK 8, the use of _ as an identifier caused `javac` to give a warning, while in JDK 9, it caused an error. In the regime described by this JEP, Java SE 8 would specify a preview language feature that defines _ as a keyword. `javac` in JDK 8 would give a warning for the use of _ as an identifier, unless preview features are enabled, in which case `javac` would give an error. Subsequently, Java SE 9 would specify the definition of _ as a keyword on a final and permanent basis, so `javac` in JDK 9 would always give an error for the use of _ as an identifier, even when preview features are not enabled.

Specifications of Preview Features

Preview language and VM features are specified in easy-to-read documents that are incorporated by reference into the Java Language Specification (JLS) and the Java Virtual Machine Specification (JVMS).

Preview APIs are specified alongside permanent APIs in the API Specification of Java SE. The author of a preview API element, whether as large as a module or as small as a method, must record its preview status by annotating its declaration with `@jdk.internal.javac.PreviewFeature`. This annotation causes the `javadoc` tool to add a message to the element:

```
$ELEMENT_NAME is a preview API of the Java platform.
Programs can only use $ELEMENT_NAME when preview features are enabled.
Preview features may be removed in a future release, or upgraded to permanent features of the Java platform.
```

The author of a preview API element must also provide an `@since` tag in the element's javadoc comment, indicating the release when `@preview` was first added. If the API element is eventually made final and permanent in Java SE $N, then the `@since` tag must be changed to indicate release $N, as the element's history prior to $N is not of long-term interest.

If a preview feature affects a non-Java API or language-independent protocol, the author must update the narrative specification of the API/protocol to make clear that some elements are a preview API. The update must remind developers that "Preview features may be removed in a future release, or upgraded to permanent features of the Java platform."

The JLS and JVMS (including their incorporated-by-reference specifications of preview language and VM features), along with the API Specification and other narrative specifications, are incorporated by reference into the Java SE Platform Specification by the Umbrella JSR for the Java SE Platform.

The Umbrella JSR provides a definitive list of the preview API elements in the release. For convenience, the `javadoc` tool enumerates all preview API elements in the release on a "PREVIEW" page, analogous to the "DEPRECATED" page which enumerates all deprecated API elements.

The Java Compatibility Kit checks the conformance of an implementation's preview features to the appropriate assertion in the JLS, JVMS, or API Specification.

Relationship of Preview Language/VM Features and Preview APIs

Most preview APIs will be *standalone*, with no connection to preview language or VM features. For example, consider the HTTP Client API which incubated in Java SE 9 and 10: it could have shipped as a standalone preview API in Java SE 11 to ensure that feedback on the incubated API (`jdk.incubator.http`) still applied to a first-class API in Java SE (`java.net.http`).

In contrast, some preview APIs will be *co-developed* with preview language or VM features. There are three kinds of co-developed preview API -- *essential*, *reflective*, and *convenient* -- defined as follows:

1. *Essential.* The preview API exists because code cannot enjoy the preview language or VM feature without it. An essential preview API lives in the package `java.lang` or `java.lang.annotation`. The JLS refers to an essential preview API in normative text. For example, if the enhanced-for statement had been a preview language feature, then `java.lang.Iterable` would have been its essential preview API.
2. *Reflective.* The preview API exists to expose the preview language or VM feature in the Core Reflection API, Method Handle API, Language Model API, Annotation Processing API, Compiler API, or the Compiler Tree API. A reflective preview API lives in `java.*` or `javax.*` or `com.sun.source.tree.*`. The JLS will not refer to a reflective preview API in normative text, but may refer to it in non-normative text. Typically, a reflective preview API is only necessary when a preview language feature has a declaration form rather than a statement, expression, or literal form. For example, repeatable annotation types prompted additional methods in `java.lang.Class` (`getAnnotationByType`, `getDeclaredAnnotationsByType`).
3. *Convenient.* The preview API is a collection of useful classes, interfaces, and methods that promote or assist usage of the preview language or VM feature but are not essential for it. The JLS is unlikely to refer to a convenient preview API in any way. For example, the Streams API is convenient for developers who use lambda expressions, but the semantics of lambda expressions do not rely on the API.

A *normal* preview API is a standalone, essential, or convenient preview API. The treatment of code that uses normal preview APIs differs greatly from the treatment of code that uses reflective preview APIs, as described in "Use of Preview Features" below.

The author of a reflective preview API element must annotate its declaration with `@jdk.internal.javac.PreviewFeature(..., reflective=true)`. This causes the `javadoc` tool to add a different message to the element than when `reflective=false`: (the sentence "`Programs can only use $ELEMENT_NAME when preview features are enabled.`" is omitted, in accordance with "Special rules for use of reflective preview APIs" below)

> `$ELEMENT_NAME is a reflective preview API of the Java platform. Preview features may be removed in a future release, or upgraded to permanent features of the Java platform.`
>
> In Java SE 14, prior to the introduction of preview APIs, the JLS enumerated the "essential" API elements that were unavoidably associated with preview language and VM features. The JLS made it a compile-time error (respectively, a warning) to use such elements if preview features were disabled (respectively, enabled). This arrangement was the precursor to how the JLS treats use of standalone, essential, and convenient preview APIs (now enumerated in the Java SE Platform Specification rather than the JLS).
>
> Also prior to the introduction of preview APIs, the author of an API element associated with a preview language or VM feature would use the `@preview` tag in the element's javadoc comment to explain the non-final status of the element.
>
> The earliest version of this JEP proposed to use the deprecation mechanism for flagging APIs associated with preview features. Consequently, in Java SE 12 and 13, the APIs associated with preview features were terminally deprecated at birth, that is, annotated with `@Deprecated(forRemoval=true, since=...)` when they were introduced. For example, Java SE 13 declared an essential API associated with text blocks, and a reflective API associated with switch expressions. However, the deprecation-based approach was eventually dropped because it was confusing to see an API element introduced in the same Java SE release as it was deprecated, that is, to see `@since 13` and `@Deprecated(forRemoval=true, since="13")` on the same API element.

Relationship of Preview Language/VM Features and Incubating APIs

A preview feature must not rely on an incubating API, since an incubating API is not part of the Java SE Platform. For example, if a preview language feature was to expand the `throw` and `catch` statements to support a third kind of exception class (alongside checked and unchecked exception classes), then it would not be appropriate for the root class of this new kind (the analog of `java.lang.Exception` and `java.lang.RuntimeException`) to reside in an incubating API. Rather, the root class should be a preview API that is co-developed with the preview language feature.

A preview feature may be associated informally with an incubating API that is offered for developer convenience, perhaps as an alternative to the idea of a "convenient" preview API described above. For example, if a preview language feature introduced multi-line string literals, then it may be advertised alongside new string-processing methods (e.g., to strip newlines, adjust indentation, etc)

which incubate in the `jdk.incubator.strings` package long before they preview in the `java.lang.String` class.

The implementation of a preview language feature by `javac`, or the implementation of a preview VM feature by HotSpot, may rely on incubating APIs. For example, if `javac` implemented multi-line string literals (a preview language feature) by storing them in a new kind of constant pool entry (a preview VM feature), then `javac` may emit `class` files whose bytecode retrieves multi-line string literals by invoking methods in the package `jdk.incubator.strings.classfile` (an incubating API). As a convenience to developers, the incubator module which contains this package would be resolved automatically when the `--enable-preview` flag is passed to `javac` or the `java` launcher.

Preview APIs in `java.lang`

A feature owner may wish to preview an API in the `java.lang` package. The preview API may be co-developed with a preview language or VM feature (that is, an "essential" or "convenient" preview API, or perhaps a "reflective" preview API), or the preview API may be developed without any connection to a preview language or VM feature (that is, a "standalone" preview API). For example, in Java SE 14, the preview API `java.lang.Record` was co-developed with a preview language feature (Records).

Adding classes and interfaces to the `java.lang` package is significant because a Java source file imports *all* the classes and interfaces in `java.lang` automatically, via an implicit `import java.lang.*;`. The meaning of `import java.lang.*;` does *not* depend on whether preview features are enabled or disabled. Even if preview features are disabled, a `Record` class being previewed in `java.lang` is imported by every source file, and may interfere with pre-existing imports of a `Record` class or interface from another package. To see the effect of previewing `Record` in `java.lang`, first suppose there is a package `com.myapp` which declares a class `Record`, and then:

- Any source file in the `com.myapp` package will continue to compile, because the package's own `Record` class shadows the `Record` class imported from `java.lang`.
- Any source file outside `com.myapp` that says `import com.myapp.Record;` will continue to compile. In such a source file, the simple name `Record` is not ambiguous: the `Record` class imported via a fully qualified name shadows the `Record` class imported from `java.lang` via a wildcard (`import java.lang.*;`).
- A source file outside `com.myapp` that says `import com.myapp.*;` will continue to compile *unless it uses the simple name `Record`*. Such use causes a compile-time error because it is ambiguous: both `Record` in `com.myapp` and `Record` in `java.lang` are imported via a wildcard, so neither class shadows the other.

The potential source incompatibility from introducing a new class in `java.lang` is unfortunate, but unavoidable if the Java language is to evolve. A feature owner should therefore choose the most expressive simple name without undue concern for the possibility of a name clash with classes and interfaces in other packages. For example, given the broad appeal of "Records" as a preview language feature in Java SE 14, it would have been inappropriate for the co-developed preview API to be called, say, `java.lang.JavaRecord` or `java.lang.Record2` just in case some library somewhere already had a class called `Record`. Only code that uses a wildcard import (`import com.myapp.*;`) will be affected by the introduction of `java.lang.Record`, and it can easily adapt by using a specific import to get the library's version of `Record` (`import com.myapp.Record;`).

It may seem inconsiderate for a preview API to break a program (forcing it to adopt a specific import from a library) even if neither the program nor the library has any interest in the preview API. However, responsibility lies with the feature owner's decision to expand `java.lang`; the preview API merely reveals the impact of the decision sooner rather than later.

Use of Preview Features

Because preview features have not achieved final and permanent status in the Java SE Platform, they are unavailable by default at compile time and run time. Developers who wish to use preview language features and APIs in their programs must explicitly enable preview features in the Java compiler and runtime. That is, developers must "opt in" twice: once at compile time, when Java source code uses preview language features and APIs, and again at run time, when the corresponding `class` files are executed. These `class` files are special; they should not be distributed beyond the control of the developer who chose to use preview features.

(Non-Java APIs and/or language-independent protocols in the Java SE Platform may be extended to support a preview feature. Even though such extensions have not achieved final and permanent status in the Java SE Platform, they are enabled by default because it is impractical to require developers to "opt in" to their use at compile time and run time. In this regard, they are similar to *reflective preview APIs*, which are Java APIs that expose preview language and VM features.)

At compile time, the developer experience is as follows:

1. When compiling with preview features *disabled*, any source code reference to (i) a preview language feature, or (ii) a class or interface declared using a preview language feature, or (iii) a normal preview API (whether referenced by name, or invoked, or overridden), causes a compile-time error. An error is justified because a later release might remove the language construct or API element, or incompatibly change its behavior. (Source code references to reflective preview APIs are exempted, as described in "Special rules for use of reflective preview APIs" below.) For example, suppose a class `LargeFile` is added to

`java.io` as a normal preview API. With preview features disabled, `import java.io.LargeFile;` would be a compile-time error; `import java.io.*;` would not be an error, though any subsequent use of the simple type name `LargeFile` would be, such as `new LargeFile()` or `LargeFile.MAX_SIZE`.

2. When compiling with preview features *enabled*, any source code reference to a preview language feature causes a *lint warning*. This warning is not mandated by the JLS, but rather is specific to `javac`. It cannot be suppressed with `@SuppressWarnings("...")` or `-Xlint:none`.
3. When compiling with preview features *enabled*, any source code reference to (i) a class or interface declared using a preview language feature, or (ii) a normal preview API (whether referenced by name, or invoked, or overridden), causes a *preview warning*. This warning is mandated by the JLS so it occurs on all Java compilers; it is not specific to `javac`. It can be suppressed with `@SuppressWarnings("preview")`.

In our judgment, it is important to remind developers about the use of preview language features and normal preview APIs in source code, even when preview features are enabled at compile time. After all, the developer compiling the code might not be the developer who wrote it. We believe that a suppressible warning is sufficient to flag the use of a normal preview API (or a class or interface declared using a preview language feature), since API names passed to `new` or used in method calls are easy to spot even if the warning is suppressed. In contrast, we believe a suppressible warning is not sufficient to flag the use of a preview language feature, since language constructs can be subtle and might be overlooked if the warning is suppressed. Since warnings mandated by the JLS are suppressible with `@SuppressWarnings("...")`, they are not strong enough to flag the use of a preview language feature. Warnings specific to `javac` can be arbitrarily strong, hence the use of a `javac` lint warning in the third case above.

In order to allow the runtime system to detect when "opt in" is necessary, a compiler must, when compiling with preview features enabled, emit `class` files that record the use of preview language features and preview APIs by the original source code, and/or preview VM features by constant pool entries and `class` file attributes. In particular:

- If Java source code uses (i) a preview language feature of Java SE $N, or (ii) a class or interface declared using a preview language feature of Java SE $N, or (iii) a normal preview API of Java SE $N (whether referenced by name, or invoked, or overridden), then a Java compiler must record that the emitted class file *depends on the preview features of Java SE $N*. This applies even if a preview language feature is "syntactic sugar" that can be compiled without relying on preview VM features such as novel `class` file constructs.
- If a Java or non-Java compiler emits a `class` file that uses a preview VM feature of Java SE $N, then the compiler must record that the emitted class file *depends on the preview features of Java SE $N*. For a Java compiler, this requirement applies even if the Java source code uses no preview language features, because a Java compiler may choose to implement a final and permanent language feature by translation to a preview VM feature.

A `class` file denotes that it *depends on the preview features of Java SE $N* by having a `major_version` item that corresponds to Java SE $N and a `minor_version` item that has all 16 bits set. For example, a `class` file that depends on the preview features of Java SE 17 would have version 61.65535.

> Using `minor_version` is preferred to `access_flags`, which is already very busy insofar as every bit is allocated to a flag when viewing the union of `access_flags` across the `ClassFile`, `field_info`, and `method_info` structures. Unused bits in any one structure are best left for use by actual features.
>
> Using `minor_version` is also preferred to constructs like a custom attribute or a special constant pool entry. The VM's processing of the constant pool may depend on whether preview features are enabled, and such constructs would be read too late.
>
> The unsubtle all-bits-set pattern in `minor_version` is deliberately chosen in preference to setting an individual bit. We wish to avoid well-meaning but ultimately misguided guesses at future roles, and therefore bit patterns, for `minor_version`. We also wish to avoid the particular confusion around the lowest bit: if a Java SE 17 class file with preview content had version 61.1, it would be easy to interpret incorrectly as the "next" version after 61.0.

A JVM implementation for Java SE $N must not define a `class` file that depends on the preview features of a different Java SE release, even if the JVM implementation would otherwise understand the `class` file's version. Specifically, `ClassLoader.defineClass` must fail for the bytes of the `class` file. This prevents such `class` files from running years into the future, long after the preview features that were enjoyed by the developer have been removed or finalized in a different form. In essence, Java SE $N+1 does not claim backwards compatibility with the preview features of Java SE $N.

If preview features are disabled at compile time, developers should not expect their Java compiler to compile source code which uses the preview language features or preview APIs of *any* Java SE release, or to compile source code against `class` files which depend on preview VM features of *any* Java SE release. Similarly, if preview features are not enabled at run time, a JVM implementation will not load a class file that depends on the preview features of *any* Java SE release.

Special rules for use of reflective preview APIs. In order for frameworks to deliver early support for a preview language feature, frameworks should utilize the reflective preview APIs that are co-developed with it. This will allow frameworks to keep up when programs are written to use the permanent version of the language feature. However, it is not desirable for a framework's classes to be marked as depending on preview features simply because the framework uses reflective preview APIs. Consequently, the rules for use of a reflective preview API are more relaxed than the rules for use of a normal preview API given in "Use of Preview Features". In particular:

- A source code reference to an element of a reflective preview API is always allowed, even when preview features are disabled. Such a source code reference causes a suppressible preview warning, whether preview features are enabled or disabled. (This is the same warning as specified above for use of a normal preview API when preview features are enabled.)
- The source code reference does not result in the emitted `class` file being marked as depending on the preview features of Java SE $N. (In effect, *all* source code is considered participating for a reflective preview API element; see "Special rules for declarations of preview APIs" for the rules about participating source code.)

  Preview APIs are "hidden" at compile time when preview features are disabled, due to gatekeeping by the Java compiler. In contrast, preview APIs are not "hidden" at run time, even when preview features are disabled. A sufficiently motivated program can use reflection to detect, instantiate, and invoke the elements of a preview API at run time, whether preview features are enabled or disabled. Using reflection to refer to normal preview APIs allows a program to be compiled without enabling preview features, and without its `class` files being marked as depending on preview features. This implies that the program expects to run on multiple Java releases -- one release where the preview API is found and exploited via reflection, and other releases where it is unavailable -- and this in turn implies that the program expects to be distributed. It is very strongly recommended that any program which uses normal preview APIs is *not* distributed. The special rules for use of reflective preview APIs aims to accommodate frameworks that react properly to the presence of preview language features in compiled user code (which, being user code, is not distributed).

Special rules for declarations of preview APIs. The rules in "Use of Preview Features" apply to source code that *uses* preview API elements; they do not apply to source code that *declares* preview API elements. For source code that *declares* preview API elements, the following rules apply:

1. Source code that is directly or indirectly part of the declaration of a preview API is known as *participating source code*. Participating source code may be compiled whether preview features are enabled or disabled. If preview features are disabled (respectively, enabled), then no compile-time error (respectively, preview warning) is due where participating source code refers to its own preview API elements.

   For example, if a package `java.io.fast` is introduced as a preview API, then all of the code in the package would be "participating" and would not generate errors/warnings when referring to the package's own types. Similarly, if a class `java.io.LargeFile` is introduced as a preview API, then all of the code in the class -- including nested classes -- would be participating and could freely refer to `LargeFile` without causing errors/warnings.
2. The `class` files emitted for participating source code are *not* marked as depending on the preview features of Java SE $N. Per the rules above, their use by other code requires preview features to be enabled at compile time and run time; however, their mere existence at compile time or run time has no effect on any code.
3. To allow for intra-JDK use of a preview API, source code in the same module as a preview API is deemed to be participating; it is *not* required to "opt in" at compile time and run time in order to use the API without error or warning. Accordingly, when preview features are disabled, the effect of using a preview API element exported by a module is a compile-time error *only for code in other modules*. (This is similar to how the effect of using an `@Deprecated` element is a warning *only for code that is not itself deprecated*.)

Example Use of Preview Features

While this JEP mandates that preview features are disabled by default and that there exists a mechanism to enable them, it does not specify an actual mechanism. Therefore, to aid understanding, this section outlines a possible mechanism for enabling preview features in the command-line tools of the JDK: `javac`, `java`, `javadoc`, `jshell`, and possibly `jlink`. The mechanism is a single command-line flag, `--enable-preview`.

Compile time

`javac` in JDK $N accepts the `--enable-preview` flag only in combination with `--release $N` or `-source $N`. `--enable-preview` is illegal if the operand to `--release` or `-source` is not $N. (For brevity, only `--release` is mentioned in the rest of this JEP.)

The meaning of `--enable-preview` changes from one JDK to the next. Requiring the developer to spell out a concrete version number with `--release` sets the expectation that code which relies on the preview features of JDK $N is tied to that release, and may not compile on JDK $N+1.

`--enable-preview` itself does not take a version number because it would be easy to misinterpret. For example, on JDK 18, the (hypothetical) flag `--enable-`

`preview` 19 would appear to suggest support for the preview features of JDK 19, but those features are not known at the time of JDK 18's release.

On JDK 18:

```
javac Foo.java                                  // Do not enable any preview features
javac --release 18 --enable-preview Foo.java  // Enable all preview features of Java SE 18
javac --release 17 --enable-preview Foo.java  // DISALLOWED
```

On JDK 19:

```
javac Foo.java                                  // Do not enable any preview features
javac --release 19 --enable-preview Foo.java  // Enable all preview features of Java SE 19
javac --release 18 --enable-preview Foo.java  // DISALLOWED
```

When the preview features of Java SE $N are enabled (`--release $N --enable-preview`):

- $N must be the Java SE Platform release implemented by the JDK containing the compiler. Only a compiler in JDK $N may support the preview features defined by Java SE $N. A compiler in JDK $N+1 cannot be expected to support the preview features defined by Java SE $N, since they may have been changed or dropped since Java SE $N. Developers should not expect `javac` in JDK $N to compile source code which uses preview language features or preview APIs of other Java SE releases, or to compile source code against `class` files which depend on preview VM features (see below) of other Java SE releases.
- If a compile-time error occurs due to the incorrect use of a preview language feature or any preview API of Java SE $N, then `javac` may indicate that the error is associated specifically with a preview feature. This helps to highlight that the feature, as well as the compile-time errors associated with it, may vary in future Java SE releases.
- If no compile-time error occurs, then a `class` file emitted by `javac` *depends on the preview features of Java SE $N* if the source code corresponding to the `class` file refers to (i) a preview language feature of Java SE $N, or (ii) a class or interface declared using a preview language feature of Java SE $N, or (iii) a normal preview API of Java SE $N (whether referenced by name, or invoked, or overridden). (Source code references to reflective preview APIs are exempted, as described in "Special rules for use of reflective preview APIs" above.)
- For any source code which refers to a class or interface declared using a preview language feature of Java SE $N, `javac` issues a suppressible "preview warning". `javac` issues the same warning for any source code which refers to a normal preview API.

Whether preview features are enabled or disabled, `javac` in JDK $N prints a message if it detects the use of a preview language feature of Java SE $N in source code. This message cannot be turned off by using `@SuppressWarnings` in source code, because developers must be unfailingly aware of their reliance on the Java SE $N version of a preview language feature; the feature may change subtly in Java SE $N+1. The message looks like this:

```
Note: Some input files use a preview language feature.
Note: Recompile with -Xlint:preview for details.
```

The off-by-default "preview" analysis suggested by the message provides information about each use of a preview language feature of Java SE $N. Additionally, on a best effort basis, the analysis may provide information about the attempted use of preview language features of other Java SE releases.

> The use of deprecated program elements causes `javac` to print a similar message, suggesting to `Recompile with -Xlint:deprecation for details.` However, developers can hide that message by using `@SuppressWarnings("deprecation")` or `@SuppressWarnings("removal")`. The suppression of removal warnings, which indicate use of *terminally* deprecated elements, is discouraged, but it is permitted because the effect of removing such elements in a future Java SE release will be immediately and abundantly clear to developers (their programs will fail to recompile). In contrast, a preview language feature could be changed in a source-compatible but not behaviorally-compatible manner in a future Java SE release; programs which used the feature would recompile successfully, but have a different meaning. In effect, the future of a preview language feature is less predictable than the future of a terminally deprecated program element. Accordingly, the risk of using a preview language feature is higher than the risk of using a terminally deprecated program element, so the warning message for a preview feature is stiffer.

`javadoc` and `jshell` in JDK $N support the preview language features and APIs of Java SE $N only if `--enable-preview` is specified at startup.

Given `--enable-preview`, if `javadoc` processes (i) an element that was declared using a preview language feature, or (ii) an element whose declaration refers to *another* element that was itself declared using a preview language feature, or (iii) an element whose declaration refers to a preview API in which the element is not participating (as described in "Special rules for declarations of preview APIs"), then `javadoc` adds a message to the element: (customized as appropriate)

```
$ELEMENT_NAME relies on preview features of the Java platform:
- $ELEMENT_NAME is declared using $FEATURE,
  a preview feature of the Java language.
- $ELEMENT_NAME refers to one or more preview APIs: ...
- $ELEMENT_NAME refers to one or more types which are declared
  using a preview feature of the Java language: ...

Programs can only use $ELEMENT_NAME when preview features are enabled.
```

```
Preview features may be removed in a future release,
or upgraded to permanent features of the Java platform.
```

Furthermore, if $ELEMENT_NAME is a module, package, class, or interface, then `javadoc` flags every *use* of the element by placing a "PREVIEW" superscript next to the element's name wherever an API signature refers to it.

`javap` in JDK $N does not accept the `--enable-preview` flag. It automatically highlights where a `class` file depends on preview features of Java SE $N. For a `class` file that depends on preview VM features of an earlier Java SE release, `javap` in JDK $N makes a best effort to render the preview content.

Run time

The `java` launcher in JDK $N takes the `--enable-preview` flag to enable the preview features of Java SE $N. This enables execution of any `class` file that depends on those preview features, either because it relies directly on preview VM features or because it was compiled from source code which used preview language features or preview APIs.

On JDK 18:

```
java Foo                            // Do not enable any preview features
java --enable-preview Foo           // Enable all preview features of Java SE 18
java --enable-preview -jar App.jar  // Enable all preview features of Java SE 18
java --enable-preview -m App        // Enable all preview features of Java SE 18
```

On JDK 19:

```
java Foo                            // Do not enable any preview features
java --enable-preview Foo           // Enable all preview features of Java SE 19
java --enable-preview -jar App.jar  // Enable all preview features of Java SE 19
java --enable-preview -m App        // Enable all preview features of Java SE 19
```

When the preview features of Java SE $N are enabled (`--enable-preview`):

- $N must be the Java SE Platform release to which the JVM implementation conforms. Only a JVM implementation which conforms to Java SE $N may support the preview features defined by Java SE $N. (It is extremely unlikely that a JVM implementation for Java SE $N+1 would want to support the preview features defined by Java SE $N.)
- If a `class` file's use of a preview VM feature of Java SE $N causes an exception during loading, linking, or execution, then the JVM implementation may indicate that the exception is due to an preview VM feature of Java SE $N.

Whether preview features are enabled or not, `class` files that do not depend on preview features are loaded, linked, and executed in the ordinary way.

The JVM implementation in JDK $N has an off-by-default ability to show which loaded classes depend on the preview features of Java SE $N. To enable the ability, pass `-Xlog:class+preview` to the `java` launcher.

Process issues

The JDK Project uses the JEP Process to manage feature development. Every Java language feature, JVM feature, and Java SE API feature starts life as a draft JEP that moves through various phases of candidacy and endorsement before being integrated into a JDK feature release. It is envisaged that a JEP can be drafted, reviewed, endorsed, have development work start, and be submitted as a JDK candidate (to receive a JEP number) without regard to whether the feature will be integrated as a preview feature. This affirms, for a given release of the Java SE Platform, that the technical quality of a preview feature is equal to that of a final and permanent feature. In a nutshell, previewing should not be top of mind when a feature is designed and developed.

At some point, a JEP owner may warm to the idea of making the feature available on a preview basis. The owner should discuss the implications of previewing in the JEP, such as factors which might cause the feature to be unsuccessful. If the JEP covers a preview language feature that is co-developed with a preview API, then the JEP should caution developers about relying on the preview API in contexts with no connection to the preview language feature. The JEP owner is also free to discuss previewing in the feature's implementation (such as the choice of package names for internal classes) and in related processes (such as compatibility review).

The formal step to preview availability occurs when the JEP reaches Proposed To Target status. At that time, the JEP owner must state if they wish the feature to be available as a preview in the proposed JDK feature release. A good approach is to say `(Preview)` in the title, and to state `This is a preview language feature in Java $N` in the Summary section, e.g., see JEP 325. If and when the JEP reaches Targeted status, then the preview feature will be listed alongside permanent features in the Umbrella JSR of the Java SE Platform, e.g., see Java SE 21. After the JEP is Targeted, it cannot easily change from preview to permanent, or vice versa; it must return to Candidate status first.

After a preview feature's code has been integrated into JDK $N, the JEP is closed as usual. Much evangelism lies ahead for the JEP owner! The choice of techniques and channels to request and collect feedback from developers is left to the discretion and experience of the JEP owner.

The JEP owner should be aware that while the JDK Project ships a release every six months, that does not mean there are six months between JDK $N and JDK $N+1 to gather and incorporate feedback. Due to the release process, there are only three months between the release of JDK $N and the inception of JDK $N+1 (when the `jdk/jdk`*`N+1`* repository is forked from the main-line `jdk/jdk` repository). During those three months, feedback on a preview feature in JDK $N can safely trigger changes in the design and implementation of the feature for JDK $N+1. However, once the inception of JDK $N+1 occurs, the three months until the release of JDK $N+1 are reserved for "stabilization"; this time is not available to re-design or re-implement a preview feature. Accordingly, a JEP owner will probably

want to re-preview their feature in JDK $N+1 to collect further feedback, even if the feature is unchanged relative to JDK $N. To re-preview, they must file a new JEP for JDK $N+1. A good approach is to say `(Second Preview)` in the title, and to record any evolution of the feature in a History section, e.g., see JEP 354.

Eventually, the JEP owner must decide the preview feature's fate. If the decision is to remove the preview feature, then the owner must file an issue in JBS to remove the feature in the next JDK feature release; no new JEP is needed. On the other hand, if the decision is to finalize, then the owner must file a new JEP, noting refinements informed by developer feedback. The title of this JEP should be the feature's name, omitting the earlier suffix of `(Preview)` / `(Second Preview)`, and without adding any new suffix such as `(Standard)` or `(Final)`. This JEP will ultimately reach Targeted status for the next JDK feature release.

Alternatives

Under the approach outlined above, a `class` file might denote that it depends on the preview features of Java SE $N *even if nothing in the `class` file truly does*. This would typically occur if source code uses a preview language feature that "compiles away"; if not for the `minor_version` item of 65535, the `class` file could be executed on a JVM implementation for Java SE $N in the ordinary way. An alternative approach would be less strict: require a `class` file to denote that it depends on the preview features of Java SE $N *only if something in the `class` file truly does*. (This could be coupled with a run-time check that `class` files with a `minor_version` of 65535 genuinely do use preview VM features and/or preview APIs.) However, `class` files produced under this alternative approach could be widely distributed and executed for many years, long after the preview language feature is changed or removed. This is undesirable because the developer has no motivation to re-code to avoid the preview language feature; they can treat it as effectively final and permanent.

The JDK Project could publish Early Access (EA) binaries during stabilization of the release, prior to General Availability (GA). EA binaries may give new features the broad exposure necessary to provoke useful feedback. If refinements could be designed, specified, and implemented prior to GA, then there would be no need to preview the features in a GA release. Historically, however, EA binaries have not been widely downloaded and tested by developers at large; this is likely to be even more true when GA releases occur every six months.

In some cases, project-specific EA binaries are a plausible carrier for features that need feedback:

- when the feature is not yet of the technical quality required for inclusion in a GA release of the JDK Project; or
- when the target audience of the EA binaries is small or heavily focused on the project, and the high visibility of a GA release is not yet desired; or
- when preview features are somehow pervasive and thus difficult to factor as separately packaged APIs, isolated language features, or discrete VM mechanisms.

A different way to include disabled features in a GA release is to hide them behind existing flags for conditional behavior. These include -Xlint:future in javac and -Xfuture in HotSpot. Alternatively, source code which uses an preview language feature could have an annotation on the class or method declaration to indicate that `javac` should enable the preview feature automatically. (One benefit of this local, source-driven scheme is that the annotation could denote which Java SE release included the preview feature. Later versions of `javac` would refuse to compile the code, avoiding the problem where the feature's semantics, but not syntax, changed after preview and would thus "silently" modify the meaning of the code.)

Risks and Assumptions

Under the six-month cadence, a language, VM, or API feature which "misses the train" has only a short wait before the chance to catch the next one. The decision to preview a feature means including the feature in the stabilization branch, so the train has been caught, so to speak, yet the final and permanent feature is not truly "on board". This could cause confusion. Worse, it might tempt a feature owner who is in danger of missing the train to label their feature as a preview in order to catch it.

It is assumed that, with six months between JDK feature releases, the time frame for feedback on a preview feature is sufficient to allow non-trivial course corrections. However, six months may be too short to allow for thorough exploration of a preview feature by developers. This is especially true given that exploiting new language features and APIs can demand substantial refactoring of existing programs. In addition, not all of the six months leading up to the next release are available for making changes to the release. Accordingly, it is assumed that the owner of a preview feature in one release will almost always re-preview it in the next release. To illustrate this assumption: suppose that a feature previews in JDK $N in March, re-previews in JDK $N+1 in September, and is made permanent in JDK $N+2 the next March; this gives approximately eight months for feedback (April-November) before the release process locks down JDK $N+2.