https://jakartaee.github.io/jakartaee-platform/CompatibilityRequirements

References: https://semver.org/, https://wiki.openjdk.org/display/csr/Kinds+of+Compatibility

Obsolete, see Jakarta EE Versioning, Change, and Deprecation Process - DRAFT.docx

Backwards Compatibility Requirements for Jakarta EE Specifications

These requirements define the kind of changes that are permissible when a Jakarta EE specification project updates a Jakarta EE API specification from one version to the next. They were written in conjunction with the Jakarta EE specification project leads and are intended to apply equally to all Jakarta EE specifications included in the Platform or Profile.

Classes of users

The requirements below specify the permitted impact of changes to a specification on three distinct classes of "user". These are defined as follows:

A. Vendor implementations of the specification. They provide classes that implement all the mandatory interfaces in the specification.

B. Applications. They will write code that uses the mandatory interfaces defined in the specification in order to use the facilities of a particular vendor implementation while remaining portable between implementations. They will also implement any interfaces defined by the spec that are expressly intended to be implemented by applications, such as listener or callback interfaces. This is by far the largest class of users.

C. Wrapper implementations of the specification. These provide classes that implement interfaces defined in the specification but they do so by simply wrapping a vendor implementation and adding some additional value. Examples are framework developers and resource adapter developers.

Permitted changes

This section defines the kinds of changes that are permissible when defining a new version of a specification. The kinds of changes that are permissible depend on their impact on each class of users.

1. Impact on vendor implementations: it is expected that a new version of the specification will require existing vendor implementations to be modified and extended to implement the new version.

2. Impact on applications: it must not be necessary for applications that use the existing version to recompile or modify their existing code. Changes must be source-code compatible, binary-compatible, and behavior-compatible for applications that make use of the previous version of the specification. Adding a new method to an interface that is intended to be implemented by the application, such as a listener or callback interface, will always violate this requirement. Adding a new method to an interface that the application uses but does not implement will not typically violate this requirement.

3. Impact on wrapper implementations: the same restrictions apply as for applications, but are less absolute. It should not be necessary for applications that use the existing version to recompile or modify their existing code. Changes should be source-code compatible, binary-compatible, and behavior-compatible for applications that make use of the previous version of the specification. However if it is considered that there will be no or few such implementations, or if they would always be provided as an add-on by the developers of vendor implementations, then this requirement may be relaxed after careful consideration by the specification project.

Behaviour compatibility

4. Changes must be behaviour-compatible for applications that make use of the previous version of the specification. This means that applications that change to using the new version of the specification do not see changes to behaviour that were mandated by the previous version of the specification.

If the previous version of the specification states that if the application calls method X, then Y should happen, this must continue to be the case in the new version of the specification.

However, where behaviour was undefined or optional in the previous version of the specification, it may be permissible to define more explicit behaviour in a later version. If the previous version of the specification does not define or mandate how a particular API method behaves in a particular use case then it may be permissible for a later version to define it. In deciding whether such a change is permissible, the specification project must give

consideration to the behaviour of existing implementations. Just because a particular behaviour wasn't specified doesn't mean applications haven't come to depend on it.

Determining the compatibility impact of cases that throw exceptions can be tricky. In many cases exceptions are defined to be thrown if certain error conditions exist. If those conditions correspond to programming errors or configuration errors, it might be reasonable to redefine those cases to have a new non-error function, such that the exception would no longer be thrown. However, if the exception is thrown in a case where the application might reasonably be written to depend on this exception in normal operation, changing the behavior would break the application. Knowing the difference between these cases is a judgment call.

Example: if the previous version of the specification either stated that the effect of calling X under certain circumstances is undefined, or made no mention of it, then it is permissible for a later version of the specification to state what the required behaviour under these circumstances is.

Example: if the previous version of the specification stated explicitly that a certain use case was not permitted and that implementations may throw an exception but were not required to do so, then it is permissible for a later version of the specification to make such an exception mandatory, or to define new behavior for this case.

5. The only circumstances when it is permissible to break behavior-compatibility is when there is a serious security or functional defect in the behavior defined by the specification. In this exceptional case it is permissible to correct the behavior or cause it to throw an exception.

Making an interface or method optional

The Jakarta EE specification, section EE 6.1.3 "Pruned Java Technologies" defines a process to "prune" technologies from the platform. "The result of successfully applying this policy to a feature is not the actual deletion of the feature but rather the conversion of the feature from a required component of the platform into an optional component. No actual removal from the specification occurs, although the feature may be removed from products at the choice of the product vendor."

This process is intended to apply to entire specifications (in which case all its API definitions become optional) though in rare cases it could apply to major pieces of functionality in an existing specification, such as entity beans, which may become optional in Jakarta EE 9. This process is not intended to apply to individual methods, and is not intended for the case where the functionality remains unchanged but one set of interfaces or methods is replaced by another, especially when many applications are known to still use the existing interfaces or methods.

XXX - This section needs to be updated with new rules for the actual removal of APIs and specifications from the Jakarta EE platform as anticipated in Jakarta EE 9, similar to what's done for the Java SE platform.

Deprecated methods

6. A specification project may decide to designate a method or interface as being "deprecated" when its use is no longer recommended or when the specification project considers that a different interface or method should be used instead. This can be stated in the specification and in the relevant javadoc. However the @Deprecated annotation must not be used as this is noisy and annoying for users. The only exception to this is if there is a serious security or functional defect in the method or interface.

Serialization compatibility

7. Some Jakarta EE specifications define classes that implement the javax.io.Serializable interface. The Serialization Specification has details on the allowed changes to a class that retain serialization compatibility. Applications using a newer version of a serializable class must be able to read streams written by older versions of the same class, and must also be able to write streams that can be read by older versions of the same class. In general, a serializable class should define a serialVersionUID field in the first version of the class, to ensure subsequent versions are recognized as the same class. Remember that Exception classes are always serializable.

Some specific implications

This brief FAQ summarizes the implications of these backward-compatibility requirements on various possible types of change:

Q1. Is it permissible to add a new interface to an existing specification?

Yes.

Q2. Is it permissible to remove an existing interface from an existing specification?

No, because this would violate rules 2 and 3 above. Note that it is also not permissible to mark it as deprecated.

Q3. Is it permissible to add a new method to an existing interface?

If applications are required to implement this interface (e.g., if it were a listener or callback interface) then this is never permissible (see Rule 2 for a full description). If it is considered that there will be no or few wrapper implementations of the interface, or if they would always be provided as an add-on by the developers of vendor implementations, it may be permissible to add a new method to an existing interface after careful consideration by the specification project (see Rule 3 for a full description).

Q4. Is it permissible to remove an existing method from an existing interface?

No, because this would violates rules 2 and 3 above.

Q5. Is it permissible to change the behavior of an existing method?

Where behavior was undefined or optional in the previous version of the specification then it may be permissible to define more explicit behavior in a later version. In deciding whether such a change is permissible, the specification project must give consideration to the behavior of existing implementations. Just because a particular behavior wasn't specified doesn't mean applications haven't come to depend on it. Specification projects should strive to completely define behavior, leaving behavior explicitly undefined or optional only when necessary to accommodate existing implementations or anticipated implementation flexibility.

If the change would change behavior that is explicitly defined in the previous version of the specification, then it is not permitted because it would violate rules 2 and 3 above. However, if there is a serious security or functional defect in the method, then the specification project may decide to waive this restriction.

Q6. Is it permissible to define new behavior for what was previously an error case?

Sometimes. It depends on the purpose of the error case. If a program could reasonably expect a method to throw an exception when it detects bad input from a user, that behavior should not change unless the definition of valid user input changes. If a method throws an exception when it detects a programming error, such as a null parameter that is required to be non-null, it may be changed to define new meaning for that previously invalid use. Obviously judgment is required to determine whether a reasonable program might depend on detecting the error case.