Start and stop Bundles

Contents

[How to 1](#_Toc349157184)

[Activating Bundles using the declared activation policy 1](#_Toc349157185)

[Lazy Activation 1](#_Toc349157186)

[Eager Activation 1](#_Toc349157187)

[Declared and Runtime Dependencies 2](#_Toc349157188)

[Runtime Implications of starting and stopping Bundles 2](#_Toc349157189)

[Stale References 2](#_Toc349157190)

# How to

A bundle is started when first [activated](Activate%20and%20Deactivate%20Bundles.htm) . When activated, you can then stop and start multiple bundles from the Bundle main menu and individual bundles from the context pop-up menus from the Package Explorer, Bundle Detail Page View and the Bundle List Page View. There are also local tool bars in the Details and List pages for starting and stopping bundles individually.

Note that the Start menu entry and the tool bar button for a bundle is only available when the bundle is in state RESOLVED and for the *Stop* menu entry and the tool bar button when the bundle is in state ACTIVE or STARTING/<<LAZY>>.

# Activating Bundles using the declared activation policy

When activated a bundle is started using the declared activation policy. For workspace bundles, the activation policy setting is determined by the Bundle-ActivationPolicy header in the manifest file which either can be maintained manually or by using the [Eager Activation Policy](Setting%20Bundle%20Options.htm#EagerActivationPolicy) and the [Set Activation Policy to Eager on Activate](Setting%20Bundle%20Options.htm#SetActivationPolicytoEagerOnActivate) options.

## Lazy Activation

When a bundle is activated (and started), and has declared a lazy activation policy, the bundle enters state <<LAZY>> (or STARTING). If a lazy activated bundle is stopped and started again while activated, it alters between state RESOLVED and <<LAZY>> (or STARTING) respectively. The bundle will enter state ACTIVE the first time another bundle being started actually uses some capabilities from the lazy activated bundle. See [Runtime Implications of starting and stopping Bundles](#_Runtime_Implications_of) for a detailed explanation of bundle states when starting and stopping lazy activated bundles.

## Eager Activation

Bundles with eager activation policy enters state ACTIVE when first activated (and started) and alters between RESOLVED and ACTIVE when stopped and started

# Declared and Runtime Dependencies

In principle a bundle that provides capabilities and/or services to a bundle being started enters state ACTIVE, independent of the declared activation policy.

An exception to this rule is that if a bundle declares that it require capabilities (depends on) from a lazy activated bundle by only importing packages from the providing bundle and actually not creates any Java objects or use any services from the bundle. In this case the providing bundle with a lazy activation policy remains in state <<LAZY>> when the requiring bundle is started.

The reason for this is that there is a declared or a static requirement (e.g. in the import header of the requiring bundle) but at runtime no capabilities are actually used, so the class loader does not have to be loaded and the bundle remains in state <<LAZY>.

If bundles that requires capabilities from a bundle with a declared eager activation policy are started, the providing eager declared bundle enters state ACTIVE independent of if there is a runtime requirement or not.

# Runtime Implications of starting and stopping Bundles

In general, code dependencies have no influence on activation since all bundles are resolved atomically. This is accurate but leaves out some important side effects when not stated more specifically. If you activate bundle A (which is dependent on bundle B) after a successful resolve, B will in state RESOLVED load classes and create objects as requested by A, but the bundle activator (usually the Start method if any) in B has not been executed when A start to use the imported packages from B. This may lead to an incomplete initialization of B influencing the state of objects used by A.

When a bundle is started or stopped the start and stop methods of the activator class, if any, of the bundle is invoked. You can start bundles in state RESOLVED and stop bundles in state ACTIVE and STARTING/<<LAZY>>. It is also possible to stop and start bundles with build errors.

The Framework catches and wraps any unchecked and checked exceptions thrown in the Start and Stop methods in a BundleException object, forwarding any exception to the Log View.

To try to recover from an exception of type Error with origin outside a running bundle it is possible to check for interrupts within the bundle. When trapping an interrupt in the Start or Stop method, control should return as soon as possible leaving the trapped interrupt flag intact. Setting the interrupt status is best effort so there is no guarantee that an interrupt is received in the Start and Stop methods when an exception of type Error occur. The following test should make it possible to return from Start or Stop to finish the running thread:

**if** (Thread.*interrupted*())

return;

Packages exported by a stopped bundle continue to be available to other bundles. This continued export implies that other bundles can execute code from a stopped bundle, and thus the classes are not unloaded when the bundle is stopped. If the bundle is started again the start method in the already loaded activator class is invoked.

If the bundle that is being started again after being stopped has a lazy activation policy it will enter state STARTING and not ACTIVE despite that its classes are loaded. The reason for this is that no class loading is performed when an already started bundle in state ACTIVE is started the second time after being stopped. Bundles with a lazy activation policy only enter state ACTIVE when their classes are loaded on demand from other bundles being started. If the lazy activated bundle and its requiring bundle(s) is refreshed - instead of stopped and started - its classes will be reloaded and the lazy activated bundle will enter state ACTIVE.

An example of this is when a plug-in that defines a contributing extension is stopped. The extension and the referenced classes held by the plug-in defining the associated extension point is not removed unless explicit removed in the stop method of the contributing activator. Referenced classes are still invoked when the contributing extension is executed after it is stopped. The resources are first released when the bundle is unresolved, as is the case if you deactivate, reset, update (build) or refresh the plug-in.

This is different for OSGI services which are by design automatically released when the client bundles stops. No special service cleanup is needed in the activator stop method.

## Stale References

This topic is actually beyond the scope of this User Guide. It is added as a reminder due to its importance when developing bundles, and in particular when stopping and unresolving bundles. The following paragraph is an explanation of stale references copied from the OSGi Service Platform Release 4, Version 4.3.

A stale reference is a reference to a Java object that belongs to the class loader of a bundle that is stopped or is associated with a service object that is unregistered. Standard Java does not provide any generic means to clean up stale references, and bundle developers must analyze their code carefully to ensure that stale references are deleted. Stale references are potentially harmful because they hinder the Java garbage collector from harvesting the classes, and possibly the instances, of stopped bundles. This may result in significantly increased memory usage and can cause updating native code libraries to fail. Bundles using services are strongly recommended to use either the Service Tracker or Declarative Services.

In a development environment activated bundles are started and stopped continuously and stale references can accumulate and as a consequence consume an increasing amount of memory over time. As mentioned, bundles are started and stopped according to their dependency order, so if a bundle consumes a service from another bundle, the consumer is stopped first, given the possibility to release the object references and services it consumes in the stop method before the service provider is stopped releasing the provided service(s). You can inspect the providing and requiring capabilities (possible Java object references) and services a bundle use in the Details Page of a bundle. A sample use of the Service Tracker and service consumption is illustrated in the Tutorial: [Working with multiple Bundles](../GettingStarted/Tutorial%20Working%20with%20Multiple%20Bundles.htm).