

# Overview

## What problems does Annogen solve?

Annogen is a tool which helps you work with JSR175 annotations, a new feature in JDK 1.5. JSR175 is a much-needed improvement to java. However, a few challenges may lie in store for developers of frameworks which need to process JSR175 annotations (hereafter, simply called *frameworks*). Annogen aims to provide an elegant solution for the following problems:

### Annotation Overriding

Because JSR175 annotations are 'baked-in' to .class files by javac, frameworks have no elegant way to manipulate the annotation values that were declared by the class author.

### Migration to JSR175 from Older JDKs

Many existing projects may want to start using JSR175 annotations, but will still need to support older JDKs for some time. This migration presents a dilemma because JSR175 annotation types cannot even be classloaded without JDK 1.5.

### Bridging Disparate Access Models

Sun has provided new java.lang APIs for strongly-typed access to JSR175 annotation values. However, this can only be used to access annotations which have a 'runtime retention policy;' annotations without this must be accessed via javadoc, which provides a completely different, untyped API for viewing annotations. Mediating between these two access models could be a problem for frameworks, particularly those which must access source artifacts not available via reflection, such as javadoc comments.

## What does Annogen do?

Annogen uses your JSR175 annotation types to code-generate special java beans called *AnnoBeans*. AnnoBeans look exactly like your JSR175 annotations; in fact, they can actually directly implement your JSR175 annotation interfaces. AnnoBeans carry exactly the same data that appears on your JSR175 annotations. You write your framework code to use AnnoBeans when it needs to understand what annotations are on a java class.

## Why is that useful?

### Overrides

One big advantage to this approach stems from the fact that AnnoBeans can be modified. AnnoBeans start out reflecting the values on the actual annotations they represent. You are then given the option to add, modify, or remove annotation values before they are consumed by your framework.

### JDK 1.4 support

Another advantage is that you can run Annogen under JDK 1.4 and still load AnnoBeans. In this case, you can retrieve metadata from other source (such as XML or javadoc tags) and expose it to your code through the same API as your JSR175 annotations.

### Unified, strongly-typed access

A third advantage is that AnnoBeans can be used in conjunction with a variety of Java introspection APIs, including

reflection, javadoc, Mirror, QDox, and JAM. (Support for other similar APIs can easily be added in the future). This means that whether your framework is driven off of class files (via reflection) or java sources (via javadoc), you have a single, consistent, strongly-typed API for viewing your annotations.

## What do I have to do to use Annogen?

Use of Annogen is divided into three distinct phases: Generate, Override, and View. Each phase has a corresponding sub-package in the Annogen class hierarchy.

### Generate

In the generate phase, Annogen code-generates AnnoBeans from your JSR175 annotation classes. This should be done as part of the build of your framework, before you javac your framework sources. Annogen includes an ant task to make this easier. You will also need to make sure that each of your annotation classes are themselves annotated with a simple annotation, *AnnoGenInfo*, which tells Annogen the name of the AnnoBean to be generated.

### Override

Once you have your AnnoBeans, you can create an *AnnoOverride* which adds, modifies, or remove AnnoBeans associated with a given java class. How and when this is done depends upon your needs. For example, say you had a 'cache size' deployment annotation which your users might occasionally want to override at deployment time. You could implement an *AnnoOverride* which retrieves the user's override preferences (say, from an XML file) and then modifies the AnnoBeans accordingly.

### View

In the final step, your framework code reads the AnnoBeans and uses them to do something interesting with them, such as deploy a user application. You retrieve instance of the AnnoBeans through the *AnnoViewer* interface. You simply ask the *AnnoViewer* to give you the AnnoBean for a given annotation type on a given class. The AnnoBean you get back may or may not have been modified by your *AnnoOverride*; the nice thing is here that you don't have to care. Annogen allows you to maintain a clean separation between the code which gathers and manipulates annotation values and the code which acts on those values.

## Can AnnoBeans implement my JSR175 AnnotationTypes?

In the Generate phase, you have an option called *implementAnnotationTypes*. If true, the generated AnnoBeans will directly implement your JSR175 types. The advantage to this approach is that your framework code in the View phase does not ever have to know about AnnoBeans - you just use JSR175 annotation types just as you would with regular reflection. They will be implemented by AnnoBeans, but your code doesn't have to know that. This approach is somewhat simpler, cleaner and may make it easier to integrate with other code which use the same JSR175 types.

The only disadvantage to it is that you sacrifice compatibility with older JDKs, where your JSR175 annotation types won't even classload. But if JDK 1.4 support is not important to you, then you probably should set *implementAnnotationTypes* to true and write your View-phase code against your JSR175 types.

Note that in either case, your Override-phase code still must import AnnoBeans (because JSR175 types don't have setters) and your View-phase code needs to use an *AnnoViewer* (because the JDK doesn't know about your overrides).