

JAM

What is JAM?

JAM is an object model for Java types and their associated annotations. It provides a superset of the features offered by similar APIs (such as Reflection or X/Javadoc). It is also designed from the ground-up to be extremely extensible - you can write a JAM plugin which can easily build up a description of Java types using whatever information you happen to have available about them.

How does JAM relate to Annogen?

JAM is being developed and distributed as a module within Annogen. JAM is merely one of several such APIs which JAM supports. JAM has no dependencies on Annogen, though Annogen does have a few dependencies on JAM (primarily for code generation).

Can I use JAM without Annogen?

Yes. JAM is developed and distributed as a part of Annogen, but it is useful in its own right.

Where can I see the JAM API?

Right here: [JAM API Javadocs](#)

Why is JAM useful?

One thing JAM provides is a nice, clean, consistent, singly-rooted class hierarchy for describing Java types (e.g. parameters are first-class citizens of the model).

It also gives you a unified API whether you are dealing with source files or class files. This can be very helpful if your application cares about source artifacts which are lost after compilation, such as comments.

Artifact Agnosticism

JAM creates an insulating layer between your code and the real artifacts which describe the java types your code needs to view. The JAM API was designed from the ground up to allow you to write code that is completely *artifact agnostic* when inspecting java types. That is to say, JAM provides a set of abstractions which isolate your java type-inspecting code from the files that it is actually looking at. This means that you get a single API with which you can view java sources and classfiles.

For example, a given instance of JClass (JAM's abstract representation of a Java class) may in reality represent the contents of either a `.class` file or its corresponding `.java` source file. However, the JClass that your code sees looks exactly the same in either case. Without JAM, you may have to write your code twice using two separate APIs like Javadoc and Reflection. With JAM, your code more able to focus on the logic that is central to your application.

Extensibility

However, the advantages of JAM's beansed design extend far beyond simply providing a unified view of java source- and classfiles. This is because JAM allows you to write your own extensions which customize the proxies that it creates. With this extension mechanism, you can easily add, modify, or remove any part of a beaning built.

For example, you could write an extension which adds a default comment to uncommented methods on a given JClass. With such an extension in place, JClasses loaded by JAM will contain your generated comments *exactly* as

if they had actually been in the source file all along.

Even though this is an extremely simple example, consider how much trouble it might save you in a complex application composed of several subsystems. Say that those subsystems who ask each other to do things using java types that they pass around, and that we need them to use default comments. If we don't have JAM, we're going to have to write special logic in each subsystem to generate default comments when appropriate. However, with JAM, we only have to write one bit of code that weaves those comments into our proxied view of the Java classes. Those subsystems that are consuming this view need be none the wiser.