

## TracNav

- [JPFWiki](#) - Welcome Page

[Introduction...](#)

[Installing JPF...](#)

[User Guide...](#)

[Developer Guide...](#)

[Projects](#)

- [jpf-core](#)
  - [jpf-actor](#)
  - [jpf.awt](#)
  - [jpf.awt-shell](#)
  - [jpf-concurrent](#)
  - [jpf-cv](#)
  - [jpf-delayed](#)
  - [jpf-guided-test](#)
  - [jpf-mango](#)
  - [jpf-racefinder](#)
  - [jpf-rtembed](#)
  - [jpf-statechart](#)
  - [net-iocache](#)
  - [jpf-aprop](#)
  - [jpf-numeric](#)
  - [jpf-symbc](#)
  - [jpf-concolic](#)
  - [jpf-symbc-load?](#)
  - [jpf-extended-test-gen](#)
  - [jpf-parallel-spf?](#)
  - [eclipse-jpf](#)
  - [netbeans-jpf](#)
  - [jpf-inspector](#)
  - [jpf-shell](#)
  - [jpf-template](#)
  - [jpf-trace-server](#)
  - [standard NB example](#)
- [Summer Projects](#)
  - [External Projects](#)
  - [Change\(B\)log](#)

[About...](#)

- [Events](#)
- [Presentations](#)
- [Papers](#)
- [FAQ](#)
- [History?](#)
- [Support](#)
- [People?](#)
- [Playground](#)
- [Table of Context](#)

## jpf-aprop

The jpf-aprop project contains [Java annotation](#) based program property specifications, together with corresponding [listeners](#) to check them.

The general idea is to use annotations so that SUT behavior outside JPF execution is not modified, and the SUT can be run normally outside JPF. Ideally, such annotations are useful program documentation, and can be processed by a variety of verification tools (e.g. static analysis tools), like the `@Nonnull` example below, which is part of the [JSR-305 proposal](#) and can also be checked by [FindBugs™](#) or the [Checker Framework](#) which implements the upcoming [JSR-308](#).

```
class X {...
@Nonnull Y computeY() { // this is never allowed to return null, so callers don't have to check
...
}
}
```

JPF being a dynamic analysis tool, we can go beyond simple checks (e.g. extend `@Nonnull` to fields), and we do. The scheme is always the same - each annotation is paired with a corresponding checker (usually a listener, but there are also standalone tools). The annotations are also made available in a `jpf-aprop-annotations.jar` so that SUTs can be compiled without having to include any executable code from JPF projects.

To run within JPF, add the corresponding listener like

```
> bin/jpf +listener=.aprop.listenerNonnullChecker ...
```

or specify "autolisteners" in your `jpf.properties` project properties file (annotation/listener pairs, causing JPF to automatically load the listener as soon as it encounters a corresponding annotation):

```
listener.autoload=\
javax.annotation.Nonnull,\
...
listener(javax.annotationNonnull=.aprop.listenerNonnullChecker
...)
```

### Please Note:

many of the annotation checkers use the JPF attribute system, which doesn't yet support multiple attributes per entity. For now, only use autolisteners if you know there are not several different attribute types used at the same time

### Repository

The Mercurial repository can be found on <http://babelfish.arc.nasa.gov/hg/jpf/jpf-aprop>

### Listeners

Currently, the following annotations and annotation groups are supported:

- [@Nonnull](#) - check method return values and field assignments for null values
- [@Const](#) - check for object modifications within method scope
- [@SandBox](#) - check for modification of external fields from sandbox objects
- [@GuardedBy](#) - lock policy specifications for fields
- [@NonShared?](#) - check for concurrent use of non-threadsafe objects
- [@Requires, @Ensures and @Invariant](#) - pre-/post-conditions and invariants
- [@Sequence, @SequenceEvent, @SequenceMethod, @SequenceObject?](#) - automatic UML sequence diagram creation
- [@Test?](#) - in-source method test specifications

### JPF Options

Other than the generic `listener.autoload` mentioned above, there are no general properties of relevance for `jpf-aprop`. Annotation specific JPF properties are described on the related annotation pages.

## Related Work and Comparison

There are two related JSR efforts:

(1) [JSR-305 "Annotations for Software Defect Detection"](#) Thematically, this effort is the best fit for jpf-aprop, since it is focused on the semantics of annotations that can be used across tools to detect general defects. As of this writing (01/2010) JSR-305 seems to have become somewhat sidelined by other efforts (like JSR-308), and the proposed javax.annotation types are not yet part of the OpenJdk (jdk7) code base. However, distributions of the [FindBugs™](#) static analyzer do contain such types in a separate jsr305.jar, and include code to check for related properties.

(2) [JSR-308 "Type Annotations"](#) This effort is aimed at allowing Java annotations for all type specifications, not just for classes, fields, methods and method arguments (as in Java 6). The specification is not focused on particular annotations or uses (like @Nonnull), but seems to emerge as the primary testbed and integration platform for new annotation proposals through the [Checker Framework](#), which is the JSR-308 reference implementation. This framework includes not only verification related annotation types, but also corresponding 'javac' compiler plugins to check the respective properties at compile time. These checkers are implemented as standard [javac annotation processors](#), which makes them available in the context of a production compiler.

The associated tools are static analyzers, but the related annotations are general enough to make them useful in a runtime tool like JPF, provided these annotations have a `RUNTIME` retention policy (as opposed to `CLASS` or `SOURCE`, which would be sufficient for a static analyzer).

As a general rule of thumb, JPF can more thoroughly check such annotation properties but requires test cases. Since JPF can use its [runtime attributes](#) or [listeners](#) to keep track of related state of relevant objects, it is well suited to implement annotation semantics with global scope, whereas static analyzers in such cases have to resort to expensive escape analysis, limit themselves to intraprocedural analysis, or risk suffering from false positives. On the other hand, JPF depends on good test cases to find such defects, hence it is not the most efficient tool to use in early development stages.