

TracNav

- [JPFWiki - Welcome Page](#)

Introduction

- [What is JPF](#)
- [Testing vs model checking](#)
- [Random Example](#)
- [Race Example](#)
- [JPF classification](#)

Installing JPF

- [System requirements](#)
- [Download snapshots](#)
- [Download repositories](#)
- [Create site.properties](#)
- [Install NetBeans IDE plugin](#)
- [Install Eclipse IDE plugin](#)
- [Building and testing](#)

User Guide

- [Application Types](#)
- [JPF Components](#)
- [Configuring JPF](#)
- [Running JPF](#)
- [JPF Output](#)
- [The JPF API](#)

Developer Guide

- [Design](#)
- [Choice Generator](#)
- [Partial Order Reduction](#)
- [Attributes](#)
- [Listener](#)

MJI

- [Mangling for MJI](#)
- [Bytecode Factory](#)
- [Logging](#)
- [Report](#)
- [Embedded](#)
- [JPF tests](#)
- [JPF project layout](#)
- [Create a JPF project](#)
- [Coding Conventions](#)
- [Hosting update site](#)

Projects

- [jpf-core](#)
- [jpf-actor](#)
- [jpf-awt](#)
- [jpf-awt-shell](#)

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

[About](#)

- [About this Wiki](#)
- [About the Mailing Lists](#)
- [About the Development Process?](#)
- [About the Repository?](#)
- [How to Contribute](#)
- [JPFC contributor account](#)
- [Events](#)
- [Presentations](#)
- [Papers](#)
- [FAQ](#)
- [History?](#)
- [Support](#)
- [People?](#)
- [Playground](#)
- [Table of Context](#)

@Nonnull and NonnullChecker

The `@Nonnull` annotation was one of the motivating examples of the [JSR 305](#) initiative, which aimed at specifying general Java annotations for software defect detection - right along the alley of JPFC. The essence of `@Nonnull` is quite simple: it is used to explicitly state that a method argument, field or method return of reference type should never hold a `null` value (for fields, this is obviously relaxed to not hold `null` *after* the object got initialized). The purpose of this is to make it clear if the programmer explicitly has to check for `null`, and/or to detect violations of this, as shown in the following example:

```

import javax.annotation.Nonnull;

public class NonnullViolation {
...
int computeSomething (@Nonnull String s){
    return s.length();
}

public static void main (String[] args) {
    NonnullViolation t = new NonnullViolation();
    String s = null;
    ...
    int len = t.computeSomething(s);
}

```

With the following application property file:

```

# JPF application property file for example application NonnullViolation
target = NonnullViolation
listener+=,.aprop.listener.NonNullChecker

```

a JPF run of

```
> bin/jpf NonnullViolation.jpf
```

will produce the following output:

```

JavaPathfinder v5.0 - (C) 1999-2010 RIACS/NASA Ames Research Center

=====
system under test
application: NonnullViolation.java
...
=====
error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.AssertionError: null value of @Nonnull parameter:
    NonnullViolation.computeSomething(Ljava/lang/String;)I, parameter: 0
    at NonnullViolation.main(NonnullViolation.java:35)
...

```

While currently being somewhat hampered by constraints of possible Java annotation targets, there are already three immediate uses of this annotation:

(1) Fields

@Nonnull marked fields are not allowed to be un-initialized at the end of any constructor execution, and any assignment of a 'null' value to such a field is an error:

```

class SomeClass {
...
@Nonnull String id;
...
SomeClass () {
...
// forgotten initialization of field 'id' is an error
}

void doSomething(){
...
id = null; // 'null' assignment is an error
}

```

Technically this is a class invariant stating "assert id != null", but in a very intuitive and efficient way.

(2) Method Arguments

The second use is to specify that a method should never be called with a `null` argument value, as in:

```
class SomeClass {
...
void doSomething(@Nonnull Object o){
...
}
...
doSomething(null); // calling with null argument is an error
...
```

Interestingly, here the annotation becomes a precondition specifying the callers responsibility.

(3) Method Return Values

Lastly, we can use `@Nonnull` to express that a method is not allowed to return `null`:

```
class SomeClass {
...
@Nonnull Object returnSomething(){
...
return null; // returning 'null' is an error
}
...
```

This represents a return value postcondition, i.e. is the callee's responsibility to fulfill.

It should be noted that JSR-305 is related to the ongoing [JSR-308 Type Annotations](#) effort, which enhances the use of annotations and is slated to become a part of Java 7.

Usage and JPF Configuration

`@Nonnull` is an annotation proposed to be part of the standard Java libraries, hence it resides in the `javax.annotation` package. If you have the [JSR-305](#) jar in the compile classpath, you don't need to do anything additional in order to build systems using `@Nonnull`. If you don't have the JSR-305 jar, jpf-aprop includes an `jpf-annotations.jar` in its build directory, which is safe to import since it does not contain anything else but annotations (i.e. no code that could modify your system behavior).

In order to check for `@Nonnull` violations, you have to add the `gov.nasa.jpf.aprop.listener.NonnullChecker` to the list of configured listeners when starting JPF. There are two ways to do this:

(a) imperative listener configuration via command line or application property file

```
# JPF application property file checking for NonnullViolation
...
listener+=,.aprop.listenerNonnullChecker
```

(b) autolistener specification, which usually goes into project property files (`jpf.properties`)

```
listener.autoload=\njavax.annotationNonnull, ...\n\nlistener(javax.annotationNonnull=.aprop.listenerNonnullChecker\n...
...
```

The second option is preferable if there are a number of potential annotation properties you want to check for.

Implementation Notes

The NonnullChecker is implemented as a straight forward [JPF listener](#). It scans during `classLoaded` notifications for `@Nonnull` annotations, and then checks for property violations during `executeInstruction` notifications of the following instruction categories:

- `invoke` (argument precondition, construction postcondition)
- `return` (method return value postcondition)
- `putfield/putstatic` (field value invariant)

Comparison With Other `@Nonnull` Checking Tools

JSR 305 was mainly motivated by improving the reach of static analyzers for Java, with [FindBugs™](#) being the most widely used tool as of this writing. JPF on the other hand is a dynamic analyzer - a JVM that adds a lot of inspection to "normal" bytecode execution.

While there is nothing in the semantics of `@Nonnull` that lends itself specifically for static or dynamic analysis, the tools have different strengths and weaknesses, which is not only due to the respective analysis method, but also to the design philosophy governing the tool implementation.

FindBugs™ is designed to robustly and efficiently find "easy" bugs, without requiring a lot of configuration. There is no need to create a test driver, all that is required is a set of Java class files to analyze. Consequently, the entry barrier of applying FindBugs is low, but - apart from potential false positives - it might miss cases that would require deeper (e.g. interprocedural) analysis. A typical example as of this writing (FindBugs 1.3.10-dev-20100118) is a chain of function calls like:

```
class NN {
    Object field;

    NN (Object o){
        field = o;
    }

    Object bar (Object o) {
        return o;
    }

    @Nonnull Object foo () {
        return bar(field); // what if 'field' is not initialized?
    }
}
```

JPF is the opposite extreme - it was designed to find deep errors without false positives, but it might require a considerable amount of footwork to get there. In the above case, this involves writing a test driver

```
class NN {
    ...
    public static void main(String[] args){
        new NN(null).foo();
    }
}
```

We also have to configure the NonnullChecker listener in one of the ways described above.

While this seems to be little overhead in this example, the real crux is to come up with meaningful test drivers - JPF has no problem with reference- or call chains, and finds the `@Nonnull` violation no matter how much we complicate the example, but it doesn't report anything if we provide a non-null constructor argument in the above test driver.

In a way the tools complement each other - FindBugs as a first line of defense to iron out the "easy" bugs, and JPF to deeply analyze execution of realistic test cases. That said, these boundaries are not cast in stone, and there is little that prevents FindBugs visitor implementations to go deeper, or JPF listeners to be less precise. It's mostly the different tool philosophies that users should be aware of.

What really does matter is that we can use tools that span such a gamut with the same property annotations - the idea is that you just have to annotate your program once and can choose between a set of tools that can be applied at different stages of the project. As an additional benefit, the annotation is also useful documentation for programmers working on or using such annotated code.