

Wikiprint Book

Title: @Nonnull and NonnullChecker

Subject: Java Path Finder - projects/jpf-aprop/Nonnull

Version: 9

Date: 03/12/2013 05:34:52 PM

Table of Contents

TracNav	3
Introduction	3
Installing JPF	3
User Guide	3
Developer Guide	3
MJJ	3
Projects	3
About	4
@Nonnull and NonnullChecker	4
(1) Fields	5
(2) Method Arguments	6
(3) Method Return Values	6
Usage and JPF Configuration	6
Implementation Notes	7
Comparison With FindBugs	7

[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](#)

[MJJ](#)

- [Mangling for MJJ](#)
- [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](#)

- [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](#)

- [About this Wiki](#)
- [About the Mailing Lists](#)
- [About the Development Process?](#)
- [About the Repository?](#)
- [How to Contribute](#)
- [JPF 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 JPF. 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.listener.NonnullChecker
```

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

```
listener.autoload=\  
    javax.annotation.Nonnull, ...  
  
listener.javax.annotation.Nonnull=.aprop.listener.NonnullChecker  
...
```

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 [devel/listener 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 FindBugs