

## [TracNav](#)

- [JPFWiki](#) - Welcome Page

### [Introduction](#)

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

### [Installing JPF...](#)

### [User Guide...](#)

### [Developer Guide...](#)

### [Projects...](#)

- [Summer Projects](#)
- [External Projects](#)
- [Change\(B\)log](#)

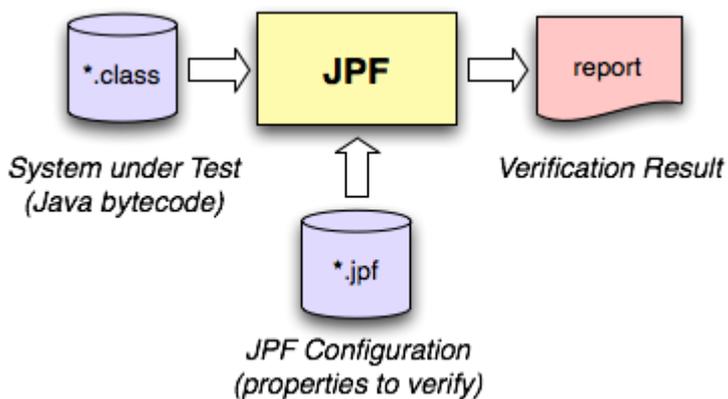
### [About...](#)

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

## What is JPF?

That depends on how you configure it. First and foremost, there is no single, monolithic JPF - it is a runtime configured combination of different components. JPF was designed so that it is easy to extend. We therefore restrict ourselves here to what the *jpf core* is, but keep in mind it is only primus inter pares among JPF components.

### The Core : a VM that supports Model Checking



The JPF core is a Virtual Machine (VM) for Java™ bytecode, which means it is a program which you give Java programs to execute. It is used to find defects in these programs, so you also need to give it the

properties to check for as input. JPF gets back to you with a report that says if the properties hold and/or which verification artifacts have been created by JPF for further analysis (like test cases).

JPF is a VM with several twists. It is implemented in Java itself, so don't expect it to run as fast as your normal Java. It is a VM running on top of a VM. While execution semantics of Java bytecodes are clearly defined in [http://java.sun.com/docs/books/jvms/second\\_edition/html/VMSpecTOC.doc.html/Sun's Java Virtual Machine Specification](http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html/Sun's%20Java%20Virtual%20Machine%20Specification), we have little hardwired semantics in JPF - the VM instruction set is represented by a set of classes that can be replaced.

The default instruction set makes use of the next JPF feature: **execution choices**. JPF can identify points in your program from where execution could proceed differently, then systematically explore all of them. This means JPF (theoretically) executes *all* paths through your program, not just one like a normal VM. Typical choices are different scheduling sequences or random values, but again JPF allows you to introduce your own type of choices like user input or statemachine events.

The number of paths can grow out of hand, and it usually will. This is what software model checking calls the *state explosion problem*. The first line of defense employed by JPF is **state matching**: each time JPF reaches a choice point, it checks if it has already seen a similar program state, in which case it can safely abandon this path, **backtrack** to a previous choice point that has still unexplored choices, and proceed from there. That's right, JPF can restore program states, which is like telling a debugger "go back 100 instructions".

So what are these features used for? Normally to find defects in the program you want to verify, but what kind of defects? By now you know the answer: it depends on how you configure JPF. The core checks for defects that can be identified by the VM without you having to specify any property: deadlocks and unhandled exceptions (which also covers Java `assert` expressions). We call these *non-functional* properties, and no application should violate them. But JPF doesn't stop there - you can define your own properties, which is mostly done with so called **listeners**, little "plugins" that let you closely monitor all actions taken by JPF, like executing single instructions, creating objects, reaching a new program state and many more. A typical example of such a listener-implemented property is a race detector, which identifies unsynchronized access to shared variables in concurrent programs (the JPF core comes with two of them).

One additional feature that comes in handy in case JPF finds a defect is the availability of the complete execution history that leads to the bug, down to every executed bytecode instruction if you need it. We call this a program **trace**, and it is invaluable to find out what really caused the defect. Think of a deadlock - usually there is not much you can directly deduce from a snapshot of call stacks.

All this explains why JPF is called "a debugger toolbox on steroids": first it automatically executes your program in all possible ways to find defects you don't even know about yet, then it explains you what caused these defects.

### **Caveat : not a lightweight tool**

Of course that is the ideal world. In reality, this can require quite a lot of configuration and even some programming. JPF is not a "black box" tool like a compiler, and the learning curve can be steep. What makes this worse is that JPF cannot execute Java libraries that make use of native code. Not because it doesn't know how to do that, but because it often doesn't make sense: native code like system calls to write to a file cannot easily be reverted - JPF would lose its capability to match or backtrack program states. But of course there is a remedy, and it is configurable: *native peers* and *model classes*. Native peers are classes that hold methods that are executed in lieu of real native methods. This code is executed by the real Java VM, not JPF, hence it can also speed up things. Model classes are simple replacements of standard classes, like `java.lang.Thread` that provide alternatives for native methods which are fully observable and backtrackable.

If you are familiar with Java implementations, you know about the humongous proportions of the included libraries, and hence it is obvious that we cannot handle all of them, at least not yet. There is no theoretical limit, and implementing missing library methods is usually pretty easy, but you should be prepared to encounter `UnsatisfiedLinkErrors` and such if you let JPF loose on large production systems. Notorious white spots are `java.io` and `java.net`, but there are people working on it. Who knows, maybe you are interested to join the effort - JPF is open sourced and there is nothing you can't see. We now have more than two dozen major collaborators in industry, government and academia around the world.

So what makes it worthwhile to invest in JPF? After all, it is a heavyweight tool, not a quick and simple bug finder. First, if you are looking for a research platform to try out your new software verification ideas, chances are you can get along with JPF in a fraction of time compared to native production VMs, which are typically optimized towards speed and care little about extensibility or observability.

The second answer is that - as of this writing - there are bugs *only* JPF can find (before the fact, that is), and there are more and more Java applications that cannot afford to learn about these bugs after the fact. JPF is a tool for mission critical applications, where failure is not an option. No surprise it was started by NASA.