



# **Java for Scientific Computing: An Introduction to Java**

Daniel Hanlon  
CLRC e-Science Centre,  
CLRC Daresbury Laboratory, Warrington, Cheshire, WA4 4AD.

*Revision: 1.0*

## **Abstract**

When examining Java, it is often difficult to see through the hype and buzzwords in order to appreciate what advantages the language would bring to a real application. For scientific programmers this lack of clarity and fears about performance can lead to the language being dismissed as just the latest trend and one that is best avoided.

**This is a Technical Note of the UKHEC Collaboration.**

Report available from <http://www.ukhec.ac.uk/publications/notes/javaintro.pdf>

© UKHEC 2002.

Neither the UKHEC Collaboration nor its members separately accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.

### ***Getting Started***

- Go to <http://java.sun.com/> and download the latest version of the Java Development Kit (JDK) for your machine.
- The Java tutorial at <http://java.sun.com/docs/books/tutorial/?frontpage-spotlight> provides "trails" for people just starting Java or for those wanting to learn about a particular topic.
- Experiment - Java is a very friendly language and one of the best ways to learn is by attempting to do something and solving problems along the way.

### ***About Java***

When it was first released in 1995, Java was described by its creators, Sun Microsystems, as:

A simple, object oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multi-threaded and dynamic language.

There is clearly a lot more to it than designing irritating applets that clutter up web pages! Many of the features that make Java interesting from the point of view of scientific programming are hidden in this list. Unfortunately these descriptions and explanations involve a bit of Java jargon...

### ***Object oriented***

Java is a fully-fledged OO programming language. Whilst it is possible to ignore these features it is much better to fully embrace the OO paradigm. Object orientation is a programming technique very well suited to large applications developed over a long period of time possibly by multiple people. This situation is increasingly common amongst scientific codes.

### ***Simple***

Java is fundamentally a very simple language. The syntax is based on C but people who've had a bad experience with C pointers or string handling shouldn't be put off. If you're moving from a procedure based language such as Fortran then Java is arguably about as simple an introduction to OO (Object Oriented) programming as you can get. Additionally Java comes complete with an enormous set of libraries which allows you to do everything from 3D graphics to network programming to creating easy Graphical User Interfaces with an absolute minimum of effort.

### ***Interpreted***

When you write a piece of code in Fortran, it must be compiled before being run. The same applies to Java but the output of the compiler is not normal machine code but Java byte code. This is in fact a type of machine code, designed to run on a Java Virtual Machine (JVM). The JVM is a small highly optimised utility that reads and executes these simple instructions.

### ***High Performance***

The type of interpreted execution used in Java allows many standard compiler optimisation techniques to be employed in generating the byte code. Additionally

JVM technology is improving all the time and "Just-in-time" (JIT) compiler techniques allow for further optimisation and native code generation at run time once all dependencies are satisfied.

Even with all these techniques and constant improvements Java doesn't and won't beat native code for speed of execution. This is a major problem for many scientific applications but only if you try to do everything in Java. Small critical sections can be left as native code on top of which a graphical user interface or a visualisation front-end or whatever can be built using Java.

### ***Dynamic***

The interpreted nature of Java allows for a dynamic run-time environment. For example, it is not necessary to think about memory allocation/de-allocation or the size of arrays etc. as this is all handled automatically by the JVM.

### ***Architecture neutral & Portable***

A Java application runs on top of a Java Runtime Environment (JRE). A JRE is a combination of a JVM and the standard libraries. Any platform providing a JRE, no matter what the underlying operating system or hardware, is suitable. A Java application is generally shipped in a single JAR file (Java ARchive) thus even eliminating the possibility of filing system incompatibilities.

This feature of Java is one of its main attractions. Existing operating systems providing a JRE include Solaris, HP/UX, Linux, Windows 9x/NT, OS/2, VxWorks, Inferno, Chorus, AIX, MVS/MP, RiscOS, MacOS, Irix running on architectures such as x86, PPC, ARM, StrongARM, Sparc, MIPS, Alpha. No serious new platform can exist without a JRE. This effectively means that code written in Java is safe from obsolescence.

### ***Multi-threaded***

Unlike most languages, Java provides intrinsic support for multiple threads of execution. The implementation depends on the virtual machine and what is provided by the underlying operating system. On a multi-processor system with a JVM that supports native threads, different Java threads will automatically be spread across the different processing elements.

### ***Distributed***

An integral part of the standard Java libraries is support for Remote Method Invocation. This is the basis for designing applications that work across completely different machines connected by a network.

### ***Robust***

Another consequence of the JVM is the facility to trap errors before they occur at the hardware level. The only way that a Java application can crash in an uncontrolled way is if there is a bug in the JVM. This would be highly unlikely and in general an application level bug will cause a cascade of exceptions to be generated pointing back to the error line in the code. Each exception is an object specific to the nature of the problem. Procedures that can cause the generation of errors, e.g. connecting

to a remote machine, require that the appropriate types of possible exception be trapped and dealt with in a controlled way.

### **Secure**

The ease by which it is possible to create distributed applications in Java could have led to the proliferation of malicious code. In order to deal with this problem, Java was designed from the outset such that a piece of code, or section thereof, has an associated privilege level. An untrusted piece of code such as an applet downloaded in a web page, for example, is not given access to the local filing system and may only create network connections back to the site from which it was downloaded.

Some of these features may require quite a change in programming style. It should however be obvious that there are clear incentives to make those changes. They should lead to faster application development and more easily maintained code.

### **Programming tools**

- The NetBeans Java development environment is highly recommended. It can be downloaded, free of charge, from <http://www.netbeans.org/>.
- Most text editors provide a Java mode for syntax highlighting etc...
- For those firmly wedded to emacs there is the Java Development environment for Emacs at <http://jde.sunsite.dk/rootpage.html>

### **Useful libraries**

- JNL - Java Numerical Library
- Free library from Visual Numerics providing numerical types, linear algebra and statistical functions. Java3D - Java 3D extensions based on underlying native 3D libraries/hardware for excellent performance.
- Many more listed at JavaNumerics.

### **Links**

<http://java.sun.com>

<http://math.nist.gov/javanumerics/>

<http://java.sun.com/docs/books/tutorial/?frontpage-spotlight>

<http://www.vni.com/>

<http://www.netbeans.org/>

<http://jde.sunsite.dk/rootpage.html>