Abstract
We present an overview of the Java language, Java virtual machine, and the large set of standard libraries and extensions available for Java, and provide a discussion of issues that effect Java performance. This is followed by an introduction to Java Analysis Studio (JAS), a tool written in Java, and in which Java is the language used to perform data analysis. We will also explore how OO techniques have been used to built a system from modular components including visualization, fitting and data-access components that can be used together, or on their own.

1. JAVA
1.1 Introduction to Java
James Gosling first developed Java in 1991 at Sun[1]. In was initially designed as a small language targeted at consumer electronics devices (cable boxes, VCR’s, smart toasters etc). It had little success in this arena, but in 1994, the HotJava web browser appeared[2], written in Java and with the then unique ability to execute “applets” – small programs written in Java and downloaded into the browser via the web. Amongst much hype Java was adopted as the “Web Programming Language” and licensed from Sun by many key Internet companies including Netscape, Microsoft, Oracle, etc.

Over the past five years Java has come a long way from its beginnings as a cool language for embedding spinning coffee cups in web pages. Nowadays it is a mature OO language, with a large set of powerful standard libraries, which allow graphical programs to run without change across most platforms, including Unix, PC, Mac, etc. Although Java’s role in the web browser has diminished it has become the dominant language in the development of server side web applications, and in terms of demand for programmers now meets or exceeds the demand for C++ programmers. Java has made inroads in many other arenas, including scientific programming[3].

The primary motivation behind the design of Java is reliability; the intention being that the compiler should find as many errors as possible, and once successfully compiled the program should have a high probability of running as intended. In comparison C++ has been designed with efficiency as its number one priority, thus the user is given a lot of control over implementation details: Objects on the stack vs. on the heap, manual memory allocation/deallocation etc. The downside of this is that the user (especially the naïve user) has many opportunities for subtle errors, often resulting in hard to debug problems. As we shall see later, modern optimization techniques applied to Java code mean that the advantage in performance gained by C++ is often very small.

Where the functionality of C++ and Java overlap the Java syntax is normally identical to the C++ syntax, however many of the historic features of C++, or features judged to add excessive complexity for small gains, have been dropped. For example Java has:

- No pointers – instead object “references” are used. References are similar to pointers except that there is no pointer arithmetic, and no explicit dereference operators.
- No explicit delete operator, instead objects are freed when they are no longer reachable (via a process known as garbage collection).
• No operator overloading.
• No global variables or functions, everything in Java is part of a class or interface.

1.2 Java “Virtual Machines”

The Java language is designed to be used in a slightly different way than languages such as Fortran or C/C++. The Java compiler does not directly convert Java source code into machine specific instructions, instead it generates machine independent “pseudo-machine-code” called Java bytecodes. To run a Java program you must have a Java “Virtual Machine” (VM) on the target machine. The earliest Java VM’s simply interpreted the bytecodes at runtime to execute the program, although more modern Java VM’s convert the bytecodes to machine code at runtime, often employing sophisticated optimization techniques which we will describe in more detail later.

1.3 Java Libraries

In addition to the Java compiler and Java virtual machine, the third component of the Java “platform” is the large set of machine independent standard libraries supplied with Java or available from third parties. The large set of libraries make programming in Java more efficient in two ways, first the developer does not have to waste time reimplementing common functionality, and secondly it is much easier to reuse components written by someone else since they are unlikely to be based on some incompatible graphics or utility library.

Some examples of common Java libraries are the Collections library, the Swing GUI library, 2D and 3D graphics libraries, JDBC and ODMG database interface libraries, and libraries for distributed programming including CORBA and RMI.

1.4 Java Performance

Optimization of C++ and Fortran programs is typically done at compile time. Static optimization as this process is now called is a very mature technology developed over many decades. Over the past five years a lot of research has been done on dynamic optimization[4,5], a technique better suited to languages such as Java, resulting in very impressive performance improvements over earlier versions of Java.

![Fig. 1 Results of a benchmark comparing Java and C++ performance. For a detailed explanation see ref. [6].](image)

Dynamic optimization is performed by the virtual machine at run-time. The program is continuously monitored as it executes, and only code that is actually found to be a bottleneck is optimized. Since dynamic optimization is performed at run-time, more information is available to the optimizer than in the case of compile time optimization. For example, a dynamic compiler is able to generate machine code optimized for the actual processor being used (e.g. Pentium, Pentium II, Pentium III etc), and only generate thread safe code if the program is running in a multi-threaded environment. In addition, the optimizer knows whether a particular method has been overridden by any loaded subclasses, and can eliminate virtual call overhead in many cases. Since Java is a dynamic
language, new classes can be loaded at any time, so the optimizer has to be smart enough to invalidate and recompile code segments that were compiled under conditions that are no longer hold.

Because of dynamic optimization the performance of Java programs have steadily improved and have now become very close to that of C++ programs. Recent benchmarks[6] show Java performance averaging about 60% of C++ performance (see Fig. 1), although with wide fluctuation about this average, probably due to the relative immaturity of the dynamic optimizers technique. Further improvements in Java performance can be expected as the optimizers mature.

2. JAVA ANALYSIS STUDIO

2.1 What is Java Analysis Studio?

Java Analysis Studio is a desktop data analysis application aimed primarily at offline analysis of high-energy physics data. The goal is to make the application independent of any particular data format, so that it can be used to analyze data from any experiment. The application features a rich graphical user interface (GUI) aimed at making the program easy to learn and use, but which at the same time allows the user to perform arbitrarily complex data analysis tasks by writing analysis modules in Java. The application can be used either as a standalone application, or as a client for a remote Java Data Server. The client-server mechanism is targeted particularly at allowing remote users to access large data samples stored on a central data center in a natural and efficient way.

2.2 Graphical User Interface

When the application is first started, it presents the user with the interface as shown in Fig. 2. The goal is to present the user with a consistent interface from which all analysis tasks can be performed. The graphical user interface also features a complete help system, wizards to help new users get started, facilities for viewing and manipulating plots, and is extensible via Plugins written in Java to provide user or experiment specific features.

![Fig. 2 The Java Analysis Studio Graphical User Interface](image)

2.3 User Analysis Modules

Although some other graphical analysis environments allow users to define their analysis by wiring together pre-built analysis modules, we believe that the complexity of real-life physics analysis problems quickly makes such approaches unworkable. Although JAS allows some simple analysis operations to be performed using the graphical user interface, serious analysis is done by writing analysis modules in Java. To this end JAS includes a built-in editor and compiler (Fig. 3) so that analysis modules can be written, compiled, loaded and run from within the JAS environment. One of
the advantages of Java is that it supports dynamic loading and unloading of classes, which results in a very short compile-run-debug cycle that is excellent for data analysis tasks.

Java Analysis Studio is provided with a package of classes for creating, filling and manipulating histograms that can be used from the user’s analysis module. Binning of histograms is delegated to a set of partition classes, which allows great flexibility in defining different types of built-in or user-defined histograms. The built in partition classes support histograming dates and strings as well as integers and floating-point numbers, and also support either traditional HBOOK style binning while filling, or delayed binning which allows histograms to be rebinned and otherwise manipulated using the GUI after they have been filled.

2.4 Data Formats

Unlike most other data analysis applications which force the user to first translate the data into a particular format understood by that application, Java Analysis Studio is able to analyze data stored in almost any format. It does this by requiring that for any particular data format an interface module be available which can provide the glue between the application and the data. The application is distributed with several built-in Data Interface Modules (DIMs), which provide support for paw n-tuples, hippo n-tuples, SQL databases (implemented using Java’s JDBC database interface), StdHEP files and flat-file n-tuples.

Support is provided for analyzing either n-tuples or arbitrarily complex object hierarchies. While analyzing n-tuple data a number of graphical user interface options are available for plotting columns of data singly or in pairs, as well as applying cuts. The intention is to provide an interface similar to that provided by HippoDraw[7]. While analyzing n-tuple data can sometimes be convenient it is also rather limiting, and therefore we also support analysis events consisting of arbitrarily complex trees of objects.

JAS can read data stored on the user’s local machine, or stored on a remote data server. The application has been designed from the outset with this client-server approach in mind, and as a result the interface that is presented to the user is identical whether the data being analyzed is stored locally or on a remote server. When running in client-server mode the user's analysis modules are still edited and compiled locally, but when run the analysis modules are sent over the network and executed on the data server.

Since the analysis modules are written in Java and compiled into machine independent class files it is easy to move them from the users machine to the remote data server. The Java runtime
provides excellent built-in security features to prevent user analysis modules from interfering with the operation of the data server on which they are running. When the user requests to see a plot created by an analysis module, only the resulting (binned) data is sent back over the network, resulting in a very modest bandwidth and latency requirements even when analyzing huge data sets. Due to its modest network requirements JAS works quite well even when accessing a remote data server via a 28.8 kb modem.

It is hoped that the client-server features built into Java Analysis Studio will prove particularly useful to researchers who typically access data from universities where it is not possible to store the Petabyte sized data samples typically generated by today’s HEP experiments. Using Java Analysis Studio such researchers can still take advantage of the powerful graphical features of their desktop machines, while analyzing data which is stored remotely. The performance of JAS is such that it is quite possible to forget that the data is not located on the local machine.

2.5 Histogram and Scatterplot Display

One of the key components of Java Analysis Studio is the JASHist bean, which is responsible for the display of histograms and scatter plots. The charts are very efficient at redrawing themselves, so that they can easily display rapidly changing data. By interacting with the GUI, end users can easily change the title or legends just by clicking on them and typing new information, and can change the range over which data is displayed just by clicking and dragging on the axes (Fig. 4).

The JASHist bean is designed using the model-view-controller pattern, so that data to be displayed need only implement a simple Java interface and need have no other dependence on the JAS package. This makes interfacing arbitrary data to the plot bean very straightforward. Care has been taken in the design and implementation of the JASHist bean to ensure that it is a modular component that can be used easily in other applications.

The current JASHist bean includes support for:

- Display of 1-D histograms, 2-D histograms and scatter plots. Scatter plot support is optimized to handle up to millions of points.
- Overlaying of several histograms or scatter plots on one plot.
- Interactive fitting of arbitrary functions to 1-D histograms.
- Numeric or time axes, plus axes with named bins.
- Many display styles that can be set interactively or programmatically.

Fig. 4 Histogram Display in JAS
• Dynamic creation and display of slices and projections of 2-D data.
• Direct user interaction, by clicking and dragging.
• Data that is constantly changing, including very efficient redrawing to support rapidly changing data (handles over 100 updates/second).
• Printing using both Java 1 and Java 2 printing models. High quality print output is available when using Java 2.
• Saving plots as GIF images or as XML. Support for encapsulated postscript and PDF is in progress.
• Custom overlays that allow data to be displayed using user defined plot routines for specialized plots.

2.6 Extending Java Analysis Studio

Java Analysis Studio is designed to be extended by end users and/or by experiments. A number of API's have been defined to make it possible to build extensions without having to understand the details of the Java Analysis Studio implementation. Currently supported extension API’s include:

• The Data Interface Module API for writing interfaces to new data types.
• The Function API for writing new functions.
• The Fitter API for writing new fitters. This allows user defined fitters to be used in place of the built-in least squares fitter. (A fitter which used Java's Native Interface to call Minuit would be a useful extension).

A plugin API for writing user or experiment specific extensions to the JAS client. The Plugin API allows extensions to be tightly integrated into the client GUI by by supporting creation of new menu items, creation of windows and dialogs, and interaction with the event stream. The above shows two plugins that have been developed for Linear Collider Detector studies, one shows the MC particle hierarchy as a Java tree, and the other is a simple event display. A recent plugin allows a full WIRED event display to run within JAS (Fig. 5).

Fig. 5 The WIRED event display running as a JAS plugin.
2.7 Implementation

The application has been built as far as possible on industry standards and using commercial components where consistent with the goal of making the final application redistributable with no runtime license fees.

2.8 Open Source Model

JAS is now an open source project, with source code browsable directly from the JAS web site (using jCVS servlet), or accessible using any CVS client. The instructions for gaining read-only access to the CVS repository are available on the JAS web site, and read-write access is available to registered developers. Our intention is to continue to refine the design of JAS to make it easier to integrate with other applications and our hope is that making the source available will make it easier for other to understand how it works, and to contribute fixes and improvements.

In order to further facilitate cross-platform development we have adopted jmk, a pure Java utility similar to make. This enables JAS to be built on any platform with a Java Development Kit (JDK) available.

2.9 Availability

At the time of writing Java Analysis Studio version 2.2.1 is available for download from our web site at: http://jas.freehep.org. Currently we support for Windows (NT/95/98/2000), Linux and Solaris, however since the application is written entirely in Java (except for the optional Paw, Hippo and Stdhep DIM's) it should work on any platform with a JDK 1.1 (or greater) compliant Java Virtual Machine.

REFERENCES