# The FreeHEP Java Library

Mark Dönszelmann, Tony Johnson, Massimiliano Turri, Victor V. Serbo, Joseph Perl, Gary Bower
*SLAC, Stanford, CA 94025, USA*
Charles Loomis, Julius Hrivnac
*LAL, Orsay, France*

The FreeHEP Java Library is an Open-Source library of common Java classes and tools. The library tries to reduce unnecessary duplication of effort by making common functionality available to the entire HEP community and others. It consists of a HEP specific part, a generic part and a set of tools. We discuss ???

## 1. INTRODUCTION

???

Both Java Analysis Studio (JAS) and the WIRED Event Display make heavy use of the FreeHEP library. Many people outside the field of HEP are using the generic part of the library.

### 1.1. HEP specific components

The HEP specific libraries include: JAIDA, a full Java reference implementation of AIDA (Abstract Interfaces for Data Analysis); HepRep, a standard for representable items for event display; IO modules for reading HBOOK, ROOT, MCFIO and StdHEP files; and YaPPI, a particle property database. Both JAIDA and HepRep are available to C++ programmers via JNI wrappers (AIDAJNI, HepRepJNI) which come as the sole C++ part of FreeHEP.

### 1.2. General Components

The generic libraries include: a layered application framework for building Java applications using plug-in modules and services; a vector graphics output module to output formats such as PostScript, PDF, EMF (Enhanced Meta Format) and SVG (Scalable Vector Graphics); a postscript viewer; and several general and XML-related utility classes.

### 1.3. Tools

Among the tools are: AID, to describe Abstract Interfaces and compile them into other languages (Java, C++, ...); and JNeeds, to find the relation between different Java packages. The infrastructure and tools to support the distributed development of the FreeHEP Java Library is also available as part of the toolset.

## 2. THE APPLICATION FRAMEWORK

Tony???

## 3. THE VECTOR GRAPHICS PACKAGE

The VectorGraphics package of FreeHEP Java Library enables any Java program to export to a variety of vector graphics formats as well as bitmap image formats. Among the vector formats are PostScript, PDF, EMF, SVF, SWF and CGM, while the image formats include GIF, PNG, JPG and PPM.

The package uses the standard java.awt.Graphics2D class as its interface to the user program. Coupling this package to a standard Java program is therefore quite easy. It also comes with a dialog box which allows you to choose between all the formats mentioned above and set specific parameters for them.

### 3.1. Using VectorGraphics

Drawing in java takes place in the paint(Graphics g) method of a component in which the user calls methods of a java.awt.Graphics or java.awt.Graphics2D context to do the actual drawing. The VectorGraphics package extends the Graphics2D class to allow users to keep drawing to the same old and familiar Graphics2D context, while adding the functionality of the new output formats. The user code for drawing the picture therefore stays the same for both displaying on the screen as for writing it to some format.

To use the VectorGraphics package the programmer needs to add code to display the SimpleExportDialog and have the user select a format. This can be added as a MenuItem in either a PopupMenu or in the File menu somewhere.

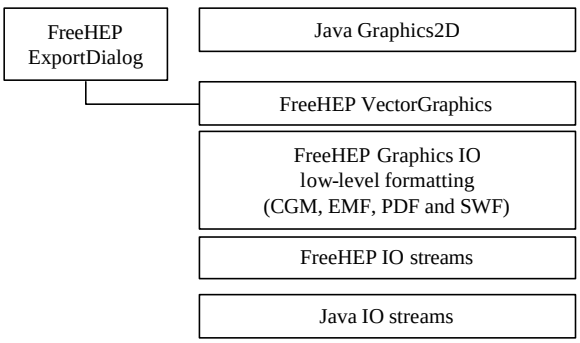


Figure ???: The layering of the VectorGraphics package sandwiched in between the Java IO Streams and the Java Graphics2D context.

### 3.2. Extensions in the VectorGraphics class

The VectorGraphics class extends Graphics2D with a number of features to write better output formats. These features are described below:

#### 3.2.1. Draw, Fill, Clear and Clip methods

All draw, fill, clear and clip methods (drawLine(), fillOval(), clearRect, clipRect, ...) which normally take integer parameters have been overloaded with methods with equal names which take double parameters. This allows for high precision drawing using these easy methods, which was only available using the draw(Shape) or fill(Shape) methods in java.awt.Graphics2D. The higher precision is necessary for Vector Formats on higher than screen resolution targets, such as printers.

The methods setLineWidth() and getLineWidth() allow you to change the linewidth without explicitly creating a Stroke object.

#### 3.2.2. PrintColors

The PrintColor extends java.awt.Color to allow for colors which are visible both on the screen and a printer. The printer may be either color, black and white or grayscale. PrintColors can be created with the correct mappings, and behave differently on the screen than they do in any of the output formats. The PrintColor class can be used anywhere a normal java.awt.Color class would be used. The default java colors are also defined, with correct mappings for printers. PrintColor will only be handled specially if a VectorGraphics context is used.

The methods setColorMode() and getColorMode() can be used to switch the output between COLOR, BLACK_AND_WHITE and GRAYSCALE.

#### 3.2.3. Symbols

To draw a large number of similar symbols (small squares, triangles, etc...) one could use a the drawSymbol() or fillSymbol() methods in the VectorGraphics context. A number of predefined symbols are available in VectorGraphicsConstants (and in VectorGraphics). Advantage of using these symbols, rather than coding them yourself in terms of lines and other primitives, is that the output format can translate these calls to procedures. This can be done in some output formats which allow procedures to be defined. Among those formats are PostScript and SVG. This way the output file gets shortened considerably.

#### 3.2.4. TagStrings

The class TagString allows you to display strings which can be marked up with the HTML like tags described in table ???

Table ???: HTML-like tags for use in TagStrings.

| Start Tag | End Tag | Description | Look |
|---|---|---|---|
| <B> | </B> | Bold Face | **Bold Face** |
| <I> | </I> | Italic Face | *Italic Face* |
| <OVER> | </OVER> | OverLine | OverLine |
| <SUB> | </SUB> | SubScript | $Sub_{Script}$ |
| <SUP> | </SUP> | SuperScript | $Super^{Script}$ |
| <TT> | </TT> | Monospaced | `Monospaced` |
| <U> | </U> | UnderLine | UnderLine |

The writing of &, <, >, " and ' can be done using the entities: & < > " and '.

The drawString() methods have been overridden to take a TagString as a parameter and handle the tags accordingly.

#### 3.2.5. Other Methods

The startExport() and endExport() methods are available, and called when outputting to a file, to allow the implementation of a VectorGraphics output format to write some header and footer, and properly close the file. These methods do not have to be called explicitly by the user, they are called by the ExportDialog.

The setCreator() and getCreator() methods are available to document the creator of the output format in the file. These calls do nothing when displaying on the screen.

The printComment() method, available in the VectorGraphicsIO subclass of VectorGraphics, allows users to add comments to their output formats, if permitted.

The setDeviceIndependent() and isDeviceIndependent() methods allow the specific VectorGraphics output format to omit device dependent information in its output, such as dates. This allows us to do testing on the output formats.

### 3.3. Extending VectorGraphics

blah

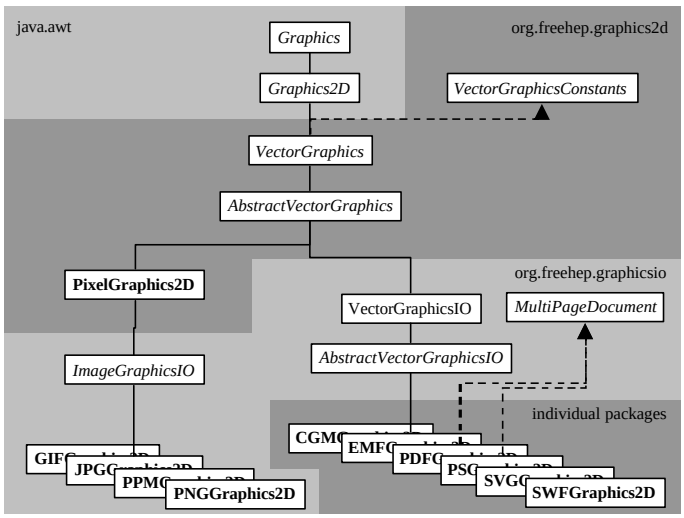


Figure ???: The overall class diagram of the VectorGraphics package showing a split between Bitmap image graphics on the left and Vector graphics on the right side.

### 3.4. The Implementation of VectorGraphics

blah

## 4. AID COMPILER

AID is a tool to generate language specific interfaces and classes (Java, C++, ...) from abstract interface definition source files (.aid).

The tools reads in one or more .aid files, parses them for correctness and keeps all information in a runtime-type-identification (RTTI) tree in memory. It then uses several generators to generate the language specific interfaces and/or classes. It also uses a simple pre-processor to handle cases where the method definitions differ between different languages.

The AID system is depicted in figure ???.

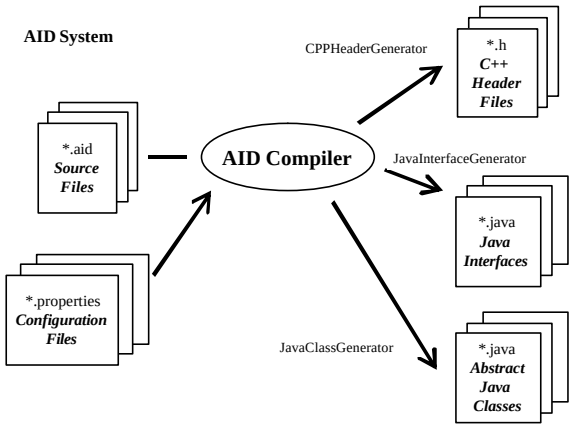


Figure ???: The AID system, in which the AID Compiler generates language dependent interfaces and header files from language independent AID files and configuration files.

## 4.1. Abstract Interface Definition Language

Multiple interface definitions may be put in one aid file. However interfaces/classes are output as one file per interface/class.

Comments of type (/**...*/, /*...*/ or //...) may exist in any place, however only the following comments are copied to the generated files:

Comments in front of the **package** statement.

Comments in front of the **interface** statement.

Comments in front of fields (values and **enum** definitions).

Comments in front of methods.

Comments in front of the closing bracket of the interface.

Comments at the end of the file.

It should be noted that comments inserted between parameters, variables and values of **enum** definitions are **NOT** copied.

The syntax of the AID language is similar to the java language for its definition part, using the following constructs:

package

interface

class, implementing interfaces, and having static methods defined in @java {} closures

methodheaders

constant values

and adds the following keywords and symbols:

**const** to define the constness of parameters, return types as well as functions, as needed by C++.

**&** for references in C++, for example ITuple &.

***** for pointers in C++, for example ITuple*[].

**enum** for C++ constants.

**=** for default values in method definitions.

<> to handle reference (but not definition) of parameterized types such as Collection<String>.

If methods differ too much to be uniquely defined for different languages, then one can use the pre-processor included in aid. It allows the user to define different sections for different target languages. The following statements are allowed:

**@ifdef** property

**@ifndef** property

**@else**

**@endif**

The properties defined depend on the selected generator, see below. The section that runs through the preprocessor without being filtered, is properly parsed by aid, so should adhere to the aid language. Imports/includes and types will be properly generated.

A language specific closure may exist using the syntax `"@cpp { non-parsed language specific text}"` at any place where comments may exist and are copied. The closure will be copied for that language only. Any language acronym can be defined and picked up the generator, as long as it is an identifier starting with "@". The sections between the curly braces are NOT parsed by aid, has to conform the target language and any typing/imports/includes (see below) will not be generated.

The tool could easily be extended for other language bindings if necessary.

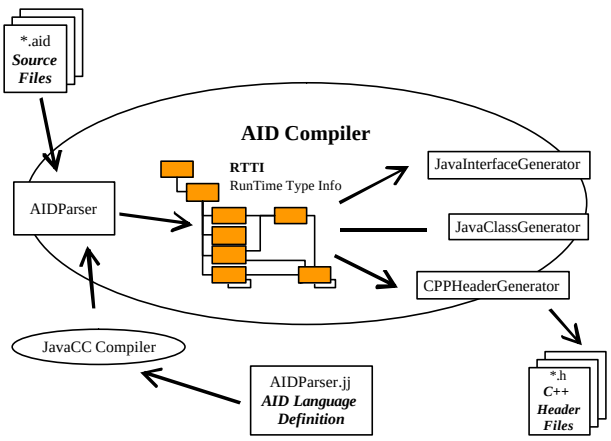


Figure ???: The internals of the AID Compiler, for which the AID language parser is generated by JavaCC, and the parsed information is stored in a Runtime Type Information system.

Blah

### 4.2. Extending AID

The generators use a set of tables to do translations of **types**, lookup which **include/imports** are needed and define initial **values**. These tables can be extended by the user.

Aid translates types (return types, types of parameters, types of inheritance, types of exceptions and types of variables) to language specific types. Primitive types are not looked up. To inherit a class in one language and not in another one simply defines a special type, setting it in one property table to the superclass and in the other to nothing.

Resulting language specific types are looked up in tables to see if they need imports/includes or forward declarations.

Initializer values are also looked up in tables. If mapped to the empty string, the initialization will be omitted. Care must be taken that initial values are only specified starting at the end of the parameter list, and if mapped to the empty string, no initial values appear in front of that parameter, or all initial values in front have to map to the empty string.

#### 4.2.1. JavaInterface / JavaClass Generator

The property file **aid.types.java.properties** is used to lookup of java types. Since the aid files are almost the same as java, most types will not need translation and this table may be more or less empty.

The property file **aid.imports.java.properties** is used lookup what imports are needed.

The property file **aid.values.java.properties** is used to lookup initializer values. Most values will not need a translation. However, since java does not have initializer values for methods, extra methods are generated with shorter parameter lists for the interface. The JavaClass generator will generate an Abstract class which leaves the full method abstract, but fills in all the initial values for the corresponding methods with shorter parameter lists.

**Enum** definitions are output as lists of, and **const** as public final static variables.

**Const** on methods and any modifier (**&*@**) are not output to the java language.

Parameterized types are output as extra comments on the top-level type.

Both Java generators define the property **java**, while the JavaInterface generator also defines **java.interface** and the JavaClass generator defines **java.class**.

#### 4.2.2. CPPHeader Generator

The property file **aid.types.cpp.properties** is used to lookup of C++ types. Most types will need a translation.

The property file **aid.includes.cpp.properties** is used to lookup what include files are needed. Care must be taken if fully qualified names (including ::) are used as keys, since colons need to be escaped in keys in the property file (write \:\:). If a C++ type is defined as the empty string no include statement is added to the file. This can be used for classes which need no include. If the C++ type is not defined the type will be added as a forward declaration.

The property file **aid.values.cpp.properties** is used to lookup initializer values. Most values will need a translation.

Exceptions are currently removed in the C++ header. However, if the return type of a method is void, it is replaced by bool to allow the method to signal there was an exception. The corresponding comment line describing the return value is also replaced.

The package name is used as **namespace**, but is translated by the type table. If defined as empty string, no namespace will be defined. Any dots (.) in the namespace are translated to underscores (_).

The CPP Header generator defines the properties **cpp** and **cpp.header**.

### 4.3. The Implementation of AID

The parser for AID is generated by JavaCC (JavaCompilerCompiler) from a .jj file which describes the AID language in tokens and productions of these tokens.

## 5. JAVA ROOT IO

Tony???

## 6. CONCLUSIONS

Blah blah ???

## Acknowledgments

We would also like to thank the AIDA team and the people working on LCIO for trying out and using AID and giving us essential feedback. And finally we thank the ROOT team at CERN for giving enough insights in their IO format for us to write the Java ROOT IO reader.

## References

[???] Frank Gaede, “Design Considerations for the LCIO Framework”, Desy-IT, 2003. See also http://www-it.desy.de/physics/projects/simsoft/lcio/.

[???] A. Ballaminut et al., “WIRED - World-Wide Web Interactive Remote Event Display”, Computing in High Energy Physics 2000, Padova, Italy. See also http://wired.freehep.org/.

[???] A.S. Johnson, “Java Analysis Studio”, Computing in High Energy Physics 2000, Padova, Italy. See also http://jas.freehep.org/.

[???] Joseph Perl, “HepRep**:** a Generic Interface Definition for HEP Event Display Representables**”** SLAC-REPRINT-2000-020. See also http://heprep.freehep.org/.

[???] Guy Barrand et al., “Abstract Interfaces for Data Analysis - Component Architecture for Data

Analysis Tools", SLAC-PUB-9409, CERN-IT-2001-013. See also http://aida.freehep.org/.

[???] YaPPI, Yet another Particle Property Interface. See also http://yappi.freehep.org/.

[???] Rene Brun and Fons Rademakers, "ROOT - An Object Oriented Data Analysis Framework", Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch/.

[???] WebGain and Sun Microsystems, "Java Compiler Compiler™ (JavaCC) - The Java Parser Generator", 2000. See also http://www.webgain.com/products/java_cc/.