NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer’s or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.
How IChart Utilizes Advanced Java Features to Support Code Maintenance

by Frederick S Brundick

*Computational and Information Sciences Directorate, ARL*
How IChart Utilizes Advanced Java Features to Support Code Maintenance

When an application is written it is important to employ the best practices of the language being used. If the developer is using an object-oriented language, then that is the paradigm they should follow rather than the procedural methodology that they learned in college. It will make it easier for future writers to maintain and modify the code. This technical note contains a discussion of some Java features that were incorporated into the IChart application developed for the Joint Staff/J-8 Directorate.

14. ABSTRACT

15. SUBJECT TERMS

Java, programming, object-oriented, best practices, code maintenance
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>iv</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Packages and Utility Classes</td>
<td>1</td>
</tr>
<tr>
<td>3. Properties</td>
<td>2</td>
</tr>
<tr>
<td>4. Configuration Class</td>
<td>4</td>
</tr>
<tr>
<td>5. Enumerations</td>
<td>4</td>
</tr>
<tr>
<td>5.1 Simple Enumerations</td>
<td>4</td>
</tr>
<tr>
<td>5.2 Complex Enumerations</td>
<td>5</td>
</tr>
<tr>
<td>5.3 Extended “for-each” Loops</td>
<td>6</td>
</tr>
<tr>
<td>6. Methods</td>
<td>7</td>
</tr>
<tr>
<td>6.1 Overriding</td>
<td>7</td>
</tr>
<tr>
<td>6.2 Reflection</td>
<td>7</td>
</tr>
<tr>
<td>7. Appletations</td>
<td>8</td>
</tr>
<tr>
<td>8. Conclusion</td>
<td>9</td>
</tr>
<tr>
<td>9. References</td>
<td>11</td>
</tr>
<tr>
<td>Appendix. LineStrokeEnum Class Listing</td>
<td>13</td>
</tr>
<tr>
<td>List of Symbols, Abbreviations, and Acronyms</td>
<td>17</td>
</tr>
<tr>
<td>Distribution List</td>
<td>18</td>
</tr>
</tbody>
</table>
List of Figures

Fig. 1  IChart packages .................................................................2
Fig. 2  IChart properties ...............................................................3

List of Tables

Table 1  TableEnum enumeration class ...........................................5
Table 2  TableEnum helper methods ...............................................5
1. Introduction

It is important to fully utilize unique features of a programming language (where appropriate) when designing a new application. Far too often when developers begin to use a new (to them) language they continue to write in the style of a language they are comfortable with. In the early days of artificial intelligence (AI) research the common procedural languages were FORTRAN, BASIC, and COBOL. The language of choice for AI was LISP—a functional language—which is from “LISt Processor”. The syntax of LISP is based on lambda calculus and is fundamentally different from procedural languages such as FORTRAN. However, anecdotes persist about people “writing FORTRAN in LISP”.\(^1\) They wanted to do AI research but they wrote in the style of their familiar FORTRAN instead of embracing the LISP paradigm. The same thing happens today when someone puts a C++ wrapper around a C program and claims that it is a C++ program. The underlying logic is procedural and not object oriented.

In 2004 the US Army Research Laboratory (ARL) was tasked with creating an application for the Global Force Management Data Initiative (GFM DI) by the Joint Staff/J-8 Directorate. The application, IChart, is a viewer and editor for organizational and force structure data.\(^2\) IChart was initially written in Java 1.4 for its portability, graphics abilities, and connectivity to the open source MySQL database server.\(^3\) In the succeeding years the code was modified and simplified as new features were added to the Java language.

This technical note contains a discussion of various Java features such as packages, properties, enumeration, and reflection and how they are used in IChart. The result has been code that is easy to read, maintain, and modify as new features are required.

2. Packages and Utility Classes

The ARL portion of the IChart source is compartmentalized into 8 different packages as shown in Fig. 1. Packages logically divide an application into related groups of classes and allow them to share information while hiding it from other packages. For example, the “database” class contains all Structured Query Language (SQL) code to make SQL transactions transparent to the rest of the application. There are 6 utility classes\(^4\) in the various packages to provide often re-used methods.
One of the classes in the database package is SQLUtility. As is the case with many utility classes, an SQLUtility object is never instantiated. Instead the class contains static methods that are invoked to perform database actions without embedding SQL-specific code throughout the application. The invoker passes arguments such as the database connection, table names, and field names, but the inner workings and SQL syntax are hidden. This decouples classes in other packages from the SQL details, allowing major changes to be made to SQLUtility without affecting code that invokes its methods.

3. Properties

Property files are a way of allowing the user to supply program parameters in a simple and easy-to-read manner. A property may have a default value defined within the application, values may be updated with one or more properties file, and options on the command line may supply last-minute value changes. (Properties may also be modified by the running program.)

IChart not only uses several property files but includes a utility class named MyProperties to define property constants and methods to manipulate property values. There are currently 183 properties and their names are all found in this class instead of being scattered throughout the code. Every class that loads and processes properties has an addPropertiesToPane method that adds property names and values to a panel using methods defined in MyProperties. (At this time 9 classes define this method.) The user may display this panel to find property names and see their current values as shown in Fig. 2.
### General Properties in Config
- `app.version.number` L 8.0
- `build.number` L Rev448
- `revision.number` L 448
- `build.date` L 2016/05/05 13:57

### Database Properties in Config
- `database.product.name` mysql
- `database.product.version`

### Security Properties in Config
- `security.level`
- `security.level.max`
- `security.border.size` L 20
- `security.CL1.color`
- `security.CL2.color`
- `security.CL3.color`

### GUI Properties in Config
- `status.normal.color`
- `status.warning.color`
- `status.severe.color`

### Symbol/Info Properties in Config
- `use.symbols` L ✔
- `show.billet.service` L
- `show.utc.info`
- `show.node.level`
- `show.link.labels` ✔
- `show.link.types`
- `show.admin.link.type` ✔
- `show.all.detail.columns` ✔
- `show.count.node` L ✔
- `show.count.item` L ✔
- `show.countестаб`
- `show.count.mtttype`
- `show.count.perstype`
- `show.count.children`

---

**Fig. 2** IChart properties
4. Configuration Class

The Config class referenced in Fig. 2 is the way that IChart shares common variables and property values among classes in different packages. It is similar to the named common blocks that were popular in FORTRAN. When a class needs access to the values (usually in its constructor) it requests an instance of the object with the static method Config.getInstance. If the singleton object exists, then the object is returned; otherwise, one is instantiated via Config’s private constructor and then returned.

5. Enumerations

5.1 Simple Enumerations

A popular technique in C is to define literal values as macros and use the preprocessor to insert them throughout the code. Java’s approach is to declare variables as “final” and then assign values to them. Likewise, sometimes a set of constants will be declared that have related values (e.g., the days of the week). In Java 5 enumerations were implemented as classes. A simple example used by IChart, LineStrokeEnum, is shown in its entirety in the Appendix.

LineStrokeEnum enumerates a set of line strokes with the symbolic names “solid”, “dashed”, and “dotted”. The only accessor method is getStroke which returns a BasicStroke object to draw a line with the desired stroke. (An element may have multiple values, each accessible with its own method.)

When combined with properties a powerful abstraction is possible. The IChart property with the symbolic name EXT_LINK_STYLE defines the stroke to be used when a link is drawn to an external node. The allowable values are the same as the elements in LineStrokeEnum. The first line of code gets the value of the property with a default of “solid”. (Property values are always strings and must be converted to other types as required.)

```java
String styleName = props.getProperty(
    MyProperties.EXT_LINK_STYLE, "solid");
```

The second line stores a LineStrokeEnum element with the name stored in style-Name:
LineStrokeEnum extLinkStyle = 
    Enum.valueOf(LineStrokeEnum.class, styleName);

A chained conditional construct is not required because Enum.valueOf compares
the string with the element names. If the user-supplied value is not one of the enu-
meration labels then an exception is thrown by valueOf.

5.2 Complex Enumerations

The previous example is a static definition of an associative data structure that maps
a Java String to a BasicStroke. The same effect could have been obtained with a
Map. However, an enumeration is a class that may also contain multiple values and
other methods. An example of this is the TableEnum enumeration that contains 5
values and related accessor methods as shown in Table 1. Additional methods are
shown in Table 2.

<table>
<thead>
<tr>
<th>Field</th>
<th>Method</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>getTableName</td>
<td>name of MySQL table</td>
<td>PersType</td>
</tr>
<tr>
<td>fieldName</td>
<td>getFieldName</td>
<td>name of primary field</td>
<td>pers_type_id</td>
</tr>
<tr>
<td>fileName</td>
<td>getFileName</td>
<td>name of data file</td>
<td>PersType</td>
</tr>
<tr>
<td>cls</td>
<td>getCls</td>
<td>class of this element</td>
<td>PersType.class</td>
</tr>
<tr>
<td>parent</td>
<td>getParent</td>
<td>enumeration of parent</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Argument</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>isTopLevel</td>
<td>—</td>
<td>does this element not have a parent?</td>
</tr>
<tr>
<td>getTableFromClass</td>
<td>cls</td>
<td>get table name given the class</td>
</tr>
<tr>
<td>getFieldFromClass</td>
<td>cls</td>
<td>get field name given the class</td>
</tr>
<tr>
<td>valueFromTable</td>
<td>tblName</td>
<td>get enumeration given the table name</td>
</tr>
<tr>
<td>valueFromClass</td>
<td>cls</td>
<td>get enumeration given the class</td>
</tr>
</tbody>
</table>

IChart defines a class for each SQL table in a GFM DI dataset and data may be saved
into a collection of text files for manual editing. The same name was originally used
for all 3 and a simple array of table names was sufficient. Over time the array was
replaced with the enumeration and multiple values. As with LineStrokeEnum the
approach is more abstract and not hardwired to physical values.
A key feature is the way that the class of each element is used as a value. This is not an instance of the class but the class itself. Given an instance, the getClass method may be used to determine the class of the instance or the class may be obtained directly from the “class” field.

In the following statement the Person Type class is passed to the loadTable method:

```java
ResultSet rs = loadTable(PersType.class);
```

The first line of loadTable gets the enumeration with

```java
TableEnum tbl = TableEnum.valueFromClass(cls);
```

and later fetches the SQL table name with

```java
... tbl.getTableName() ...
```

to query the proper database table.

The programmer must know the class (in this example, PersType) and the names of the methods in Table 2 and that is all. The table names could be changed in TableEnum and as long as all code in IChart uses TableEnum instead of literal values, the application will work without any code changes.

### 5.3 Extended “for-each” Loops

In Java 5 the “for” loop was extended to include Iterables, creating a simple way to step through the elements of an enumeration. While this is used with other enumerations in IChart, TableEnum will be shown in a simple example. The code to compute the size of every SQL table in a GFM dataset and store the values in a Map is

```java
HashMap<Class,Integer> tableSizes =
    new HashMap<Class,Integer>;
for (TableEnum tbl : TableEnum.values()) {
    int size = SQLUtility.getCount(conn,
        tbl.getTableName(), null);
    tableSizes.put(tbl.getCls(), size);
}
```

This code fragment iterates through each element of the TableEnum enumeration. A utility method executes the SQL statement required to compute the size of each database table using the name assigned to the element. The element’s class is then used as the key in a HashMap to store the size.
6. Methods

6.1 Overriding

A feature of object-oriented programming (OOP) is the ability of a method in a child class to override a method with the same signature in a parent class. This is often used when a child method needs to add more functionality.

Another example is to declare an abstract method in the parent class. The writer of a new child class must define the method in the child for the code to compile. The abstract toDescString method appears in the BasicEID class. This class is the parent of all IChart classes that correspond to database tables. Every child class defines its own toDescString to produce a descriptive string of an object of that class. At runtime the Java Virtual Machine (JVM) determines the class of a particular object and invokes the appropriate toDescString method.

6.2 Reflection

Reflection allows a program to determine at runtime how to invoke a method when overriding a method is not an option. For a method to override another, the argument list and return type of the methods must be identical. This is not the case with the loadRecords method. While the argument is always a ResultSet the return type is an array of objects of that particular class. (A ResultSet is the dataset obtained when an SQL query is performed.) Here is a case where IChart needs to invoke a method given only the class of an object.

A GFM DI database may be validated for “Category B” errors; the details are not pertinent to this discussion. A table in the database is created with information about the errors that were discovered. This error table includes fields for the GFM identifier (GFMID) and class name of each object with an error along with the type of error. The following code accesses this table and creates objects for each GFM data element that contains an error.
The catB object is an instantiation of the CategoryB class that defines methods and variables to perform an analysis and process the results. At this point in the code the database table contains a set of errors. The code begins by getting the names of all of the types of errors, then iterating over each type. An extended for-each loop is used.

```java
ArrayList<String> types = catB.getErrorTypes(dbName);
for (String type : types) {
    String name = catB.getClassName(dbName, type);
    List<Long> gfmIDs = catB.getErrorEIDs(dbName, type);

    TableEnum tbl = TableEnum.valueFromTable(name);
    Class cls = tbl.getCls();

    Class[] cArgs = { ResultSet.class };
    Method meth = cls.getDeclaredMethod("loadRecords", cArgs);

    Object obj = cls.newInstance();
    for (long gfmID : gfmIDs) {
        ResultSet rs = tempDB.loadTable(tbl, gfmID);
        ArrayList<BasicEID> bEIDs = (ArrayList<BasicEID>)meth.invoke(obj, rs);
    }
```

The array bEIDs now contains a single entry that is an object loaded from the database. The object may be displayed to the user to pinpoint the error.

7. Appletations

Java programs are executed in 2 ways: as a stand-alone application or as an applet inside of a web browser. By taking advantage of the way that the JVM initializes these 2 approaches it is possible to write a program—an “appletation”—that may be run in either fashion.
Applications are run by executing the “main” method of a specified class, passing items on the command line as method arguments. (This is the same approach that C uses.) It is the programmer’s responsibility to instantiate an instance of the class and process the arguments.

Applets instantiate an instance of the class and invoke its “init” method. Since there is not a command line, the web page uses the HyperText Markup Language (HTML) tag “param” to define the arguments and the applet fetches them with the getParameter method.

IChart’s main method performs application-only initialization such as setting up the logging system. It then invokes the “init2” method giving it the command line arguments. When IChart is run as an applet the init method sets the static variable inAnApplet to true (the default is false) and invokes init2 with no argument.

Most of the remaining code may be executed regardless of how IChart was started. However, there are times when an operation is not possible (e.g., reading local property files) and that is when inAnApplet is checked. Likewise, the arguments or parameters are examined depending on the mode.

Visually, the applet is identical to the application. Rather than embed the graphics inside of the browser window, the main IChart window is displayed the same way that it is done by the application.

8. Conclusion

Developers should exploit unique features of a language or programming paradigm and remain current with language updates. It is easier for other programmers to understand and modify the code if standard practices and libraries are used instead of obscure home-built code.

OOP supports packages and the overriding of class methods. Packages provide flexibility that promotes code reuse, groups related code into distinct modules, and minimizes coupling between code modules. As an experiment, someone who had never read the IChart code before replaced the tree graphics with an open source graphics package with no assistance from IChart’s developer.
Maintenance typically accounts for 75% or more of the total software workload. While it is tempting to hardwire code during the development phase, it is more important in the long term to focus on maintainability and expandability. For example, in IChart’s ShowFieldEnum class the elements contain the field name and a boolean field for each of the 2 tree types. The names are displayed in a menu depending on the boolean values. It may have been easier to hardwire the 2 tree menus but over the year items have been added to the menus. Because the values are enumerations it is easy to examine the elements in an extended for-each loop. In only one method in IChart are the enumerated values explicitly referenced.

New libraries have been added to IChart as they were published. However, some sophisticated new techniques have not been incorporated because they would require a fundamental rewrite of existing code. The TreePanel class in the graphics package in IChart is the major exception to the concepts in this note. Part of it is a direct translation of legacy C code into Java and it is long overdue for a major rewrite.
9. References


INTENTIONALLY LEFT BLANK.
Appendix. LineStrokeEnum Class Listing

This appendix appears in its original form, without editorial change.

Approved for public release; distribution is unlimited.
/*
 * The class LineStrokeEnum enumerates all named
 * line styles.
 */

package mil.army.arl.gfm.graphics;

import java.awt.BasicStroke;
import java.awt.Stroke;

/**
 * The LineStrokeEnum class enumerates all named
 * line styles.
 * It works the same as an associative array.
 *
 * @author $Author: fsbrn $
 * @version $Revision: 453 $
 * $Date: 2016-11-23 13:36:23 -0500 (Wed, 23 Nov 2016) $
 */

global enum LineStrokeEnum
{
// Construct all objects with their values
/** Solid line. */
solid
   (new BasicStroke(2.0f)),
/** Dashed line. */
dashed
   (new BasicStroke(2.0f,
                   BasicStroke.CAP_SQUARE,
                   BasicStroke.JOIN_MITER,
                   10.0f,
                   new float[] { 4.0f },
                   0.0f)),

Approved for public release; distribution is unlimited.
/** Dotted line. */
dotted
  (new BasicStroke(1.0f,
       BasicStroke.CAP_SQUARE,
       BasicStroke.JOIN_MITER,
       10.0f,
       new float[] { 2.0f, 2.0f },
       0.0f));

/** Stroke for this enumeration. */
private Stroke strk;

/** Builds a LineStrokeEnum with the desired info.
 * @param strk Stroke to associate with this
 * enumeration.
 */
LineStrokeEnum(Stroke strk)
{
  this.strk = strk;
}

/**
 * Gets the stroke for this enumeration.
 * @return stroke for this enumeration.
 */
public Stroke getStroke() { return strk; }
# List of Symbols, Abbreviations, and Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>ARL</td>
<td>Army Research Laboratory</td>
</tr>
<tr>
<td>DI</td>
<td>Data Initiative</td>
</tr>
<tr>
<td>GFM</td>
<td>Global Force Management</td>
</tr>
<tr>
<td>GFMID</td>
<td>Global Force Management Identifier</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
</tbody>
</table>