# Java IMR-LIB

**By**
**Ismael Mosquera Rivera**

*To my brother Jaime, who deserves the best.*

# Abstract

In this paper, we are going to describe a code library implemented in the Java language. Our library has, at this moment, three packages: matrix, media and util; the media package has two subpackages: audio and midi. In the util package we can find a class which offers functionality to clone, get a chunck and resize 1, 2 and 3-dimensional arrays of any kind of object, including arrays of the Java atomic ( primitive ) types.

In the imr.matrix package there are classes to perform the most common operations applied to matrices an vectors, support for storage and to solve linear systems of NxN equations in several ways.

The imr.media.audio package implements an audio player with functionality to play, pause and stop playing and support to play .wav and .mp3 files.

The imr.media.midi package offers classes to load and store Standard MIDI Files ( SMF ), and also classes for all the needed objects to manage MIDI data, in addition, there are wrapper classes to wrap them into the equivalent ones offered by the Java API. There is an implementation for a MIDI player with functionality to play, pause and stop playing as in the case of the audio player coded in the imr.media.audio package.

In the imr.media package, we can see two interfaces and a sound player with capability to play audio and MIDI and functionality to loop, shuffle ( play randomly ) and to select a song from a list to be played. With this player, you can play a single song, an entire folder, and even .m3u files.

We named this library IMR-LIB where IMR stands for Ismael Mosquera Rivera.

# Content

# 1 Java Overview

The Java language was developed at the early 1990s from Sun Microsystems. The motivation was to solve the compatibility issues among different hardware and software platforms in heterogeneous computer systems and networks. At that time, most programmers adopted C and C++ as their prefered languages in their projects. Although C and C++ are so efficient having a quick response, the portability problem to migrate to another hardware and software architectures can be very complex.

Java is object oriented and also a language not difficult to learn, since most of its syntax is borrowed from C++. The primitive types, arithmetic, relational and logic operands are almost the same as in C/C++; the flow control statements ( if, for, while ... ) are also equivalent; the sizes for the primitive types do not change depending on the platform, they have always the same size:

byte 8-bit two's complement
short 16-bit two's complement
int 32-bit two's complement
long 64-bit two's complement
float 32-bit IEEE 754 floating point
double 64-bit IEEE 754 floating point
char 16-bit Unicode character

notice that the byte primitive type has 8 bits as in the case of unsigned char in C/C++, in addition, Java has introduced a boolean primitive data type which can take true of false as values; the char type has 16 bits which allow support for internationalization.

Applications written in Java are portable among heterogeneous system architectures because the Java compiler generates an intermediate code called 'bytecode' which can be executed under any platform having the JVM ( Java Virtual Machine ) available. You can compile Java code under any platform, and the resulting bytecode can run without modification in the Java run-time system installed in a concrete platform target.

The potential problem of possible errors introduced by the use of pointers in C/C++ is in Java overcome by the integration of a garbage collector which runs as a low-priority thread in the JVM, so, when an object is not used, it is automatically rleased and the amount of memory previously allocated is retrieved.

Java is an OOP language where any code must be written inside a class then, an object is a concrete instance of a class which really acts as a template. Java compliants with all requirements to be object oriented: encapsulation, inheritance and polymorphism. In Java there are not multiple inheritance as in C++ so, you can derive classes by subclassing from just one parent class; nevertheless, you can implement multiple interfaces. In Java, an interface declares only prototypes for methods which must be implemented in the class that implements the interface; for example, we can have the following declaration for an interface in Java:

```
public interface Player
	{
	void set(String fileName);
	void play();
	void pause();
	void stop();
	}

	// END
```

So, a concrete class implementing this interface must define all the methods declared in it. A class can implement multiple interfaces, enabling support for polymorphism, because implement an interface is like subclassing from a class where all its methods are abstract. For instance, we can do the following declaration:

```
	public class MyClass extends AnotherClass implements Interface1,
Interface2, Interface3, ... InterfaceN
	{
	...
	}
```

and instantiate objects from this class like this:

```
AnotherClass myClass = new MyClass();
Interface2 iTwo = myClass;
```

we also can define inner classes, that is, classes defined inside a class:

```
public class MyClass
{
...
private class MyInnerClass
{
...
}
}
```

In this case, we declare the inner class with the ‘private’ access modifier, but we could use another one among the available access modifiers in Java; these modifiers are the ones like in C++: public, protected and private; a declaration with the public one allows the access from any outside class, if it is protected, only the owner class and its derived classes can have access, and if it is private, only the owner class will have access to the method or instance variable declared with that access modifier; there is another one where the scope is available only to the classes which belong to the same package, in that case we do not have to declare any modifier.

In Java, we can declare variables, constants and methods not having changes among different instances of objects from that class, they are called class variables or methods, we use the ‘static’ reserved keyword to that purpose. By the way, since the Java programming language is interpreted, it is not as fast as C/C++, to solve that issue when the response time must be faster, a

technology named JNI ( Java Native Interface ) is available; using JNI we can implement methods in C native code, in fact, all the code requiring a quick response from the Java API is implemented under this tecnology. For example, the Math class in the java.lang package is coded in that way; this class also implements constants and static methods since they not change amon eventual instantiations for objects from that class; we can notice that such a class acts like ordinary functions in C, but in Java, as we said before, all the code must be defined inside a class. We suppose that the implementation for the Math class in the Java API could be like this:

```
package java.lang;

public final class Math
{
public static final double PI = 3.14159265359;
...
public static native double cos(double x);
public static native double log(double x);
public static native double sin(double x);
...
...
static
{
System.loadLibrary("jmath");
}

private Math(){}
}
```

Using JNI we can implement all the methods having the native reserved keyword, build a shared library, load the library and call the C functions from the native declared methods. When we use such a class, we first load the needed library just once due to the call in the static block; notice that the library name has no extension so, if we were under MsWindows it maps to ‘jmath.dll’ and if we were under a Linux platform, it maps to ‘jmath.so’. Once the shared library has been built and made available, we can use it as follows:

```
public class MyClass
{
public static void main(String[] args)
{
double x = Math.sin(Math.PI / 4.0);
System.out.println(“sin(”+Math.PI/4.0+“)=”+x);
}
}
```

notice the use of the ‘+’ operand to concatenate strings with numerical types to get the desired output; and pay attention to the private declared constructor in the Math class, so that the Math class cannot be instantiated, on the other hand, the final reserved keyword in the class definition means that the Math class cannot be subclassed.

Java is designed to work in distributed environments due to its architecture neutrality; you can put an object in a network, maybe store it in a database, an retrieve it later for a future work. Since Java is an OOP multipurpose language, you can build large and scalable systems which its code well organized; normally, Java groups the code into packages, C++ does it into namespaces, this skill allows avoid posible conflicts among class names and packing the code properly for a deployment.

Another important feature is that Java integrates multithreading built into the language so, programming applications which use multiple threads of execution, specially accessing shared memory is easier that under other languages needing third party code libraries to perform multithreading.

The Java compiler is written in Java; the Java run-time system is written in ANSI C following the POSIX standard. The JVM can load classes on demand an make dinamic binding; the bytecode is first verified so that it do not break any rule, once all the checking is done, it is passed to the interpreter which can run at full speed because after checking the bytecode, it should be free of errors or malicious behaviour.

The JDK ( Java Development Kit ) comes with the Java compiler, the JRE ( Java Run-Time Environment ), a number of useful tools and a large API ( Application Program Interface ) with packages of classes to do almost any task that you can imagine. By the way, the cup of coffee ( Java logo ) represents the large quantity of coffee cups taken for the Java Team to help in the great effort made to achieve the goal.


Horstmann, Cornell: Core Java, Prentice Hall
Mycroft: Java 8 in Action: Lambdas, streams, and functional-style programming
Rusty Harold: Java Network Programming 3rd ed, O'Reilly
The Java Native Interface Programmer's Guide and Specification, Sun Microsystems
Doug Lea: Concurrent Programming in Java: Design Principles and Patterns, 2nd ed
Gilad Bracha: Generics in the Java Programming Language
James Elliott et al: Java Swing 2nd ed, O'Reilly
Li Gong: Java 2 Platform Security Architecture, Sun Microsystems
Jayson: Servlets and JavaServer Pages: The J2EE Technology Web, Addison Wesley

# 2 Library Description

In this chapter, we are going to make a brief description of the packages in this library. There are three packages: imr.util, which has only two classes, both with all their methods static; imr.matrix: in this package we can find a vector class, a matrix class, and classes to solve NxN systems of linear equations; imr.media: this package has two subpackages: imr.audio and imr.midi. The audio package has a class suitable to reproduce audio files in mp3 and wave format. In the midi package, you can find classes to manage MIDI data, wrapper classes for its equivalent ones in the java.sound API and a MIDI player. The media package has two interfaces and a class which implements a sound player having several features. After this introduction, we will explain in mor detail the content of the packages in this library. Anyway, we encourage the reader to consult the documentation in the 'doc' folder and remember that you can pick up inside the code to know about the implementation details.

*imr.util:* We describe this package first because the iArray class is used in the matrix package to resize and get chunks from matrices and vectors.

The methods in the iArray class use reflection to perform several tasks applied to arrays of any kind of object and primitive types. Since in Java an array is also an object, we can apply this skill to resize, get chunks and clone 1, 2 and 3-dimensional arrays efficiently, even for primitive types.

The usage of this class is so easy, just remember that, since the static methods in the class return Object, you must cast to the concrete return type, for example, if we have an array of double, to resize or get a chunk from it, we can do that as follows:

```
double[] v = ...
v = (double[])iArray.resize(v, 25);
// get a chunk from index 3 to index 9
v = (double[])iArray.get(v, 3, 9);
```

The same syntax is valid for 2 or 3-dimensional arrays, for instance if we have a bidimensional array of byte, to resize it we can do it as follows:

```
byte[][] m = ...
m = (byte[][])iArray.resize(m, 5, 6);
```

The other class in the imr.util package is a simple class also with all its methods declared static which you can use to get input for strings of characters and primitive data types from the command line. Its use is really easy, for example:

```
int n = InputReader.readInt();
System.out.println(n);
float t = InputReader.readFloat();
```

*imr.matrix:* This package provides classes to solve NxN linear equation systems, a vector class and a matrix class which are used in the algorithms to solve such a systems. In addition, the vector and matrix classes have some extra functionality. We will describe first the vector and matrix classes and, afterwards, we continue with a brief explanation about the skills used in the implemented algorithms to solve linear systems.

Vector.java

In our case, a vector can be view as an array of double. Our vector class has just to private instance variables:

```
private int size;
private double[] vector;
```

that is all what we need to build the entire class by properly manage this two members; in the remainder of this section, we discuss just what we think that is relevant, for the rest, you can look at the documentation or pick up inside the code.

The Vector class has available methods to perform the most common operations applied to vectors: add, sub, multiply and divide by a value, get the module of a vector, normalize a vector and perform the dot and cross product of two vectors. There are also static functions to do some of those operations.

This class offers support for storage. So, you can load and store vectors in a well defined format; that allows you to avoid to enter your data manually, which can save a lot of time. The format for our data files is as follows:

```
#elements
x0 x1 x2 ... xn
```

where #elements is an integer for the size of the vector, and x0 x1 ... are the component values of the vector separated by spaces. So, the content of such a file could be:

```
3
0.50 -6.01 1.00
```

that is a vector with size = 3, and its components are 0.50 -6.01 1.00; so, if we have a vector ‘v’ and want to load it from that file to print this content to the console:

```
v.load(“myvector.dat”);
v.print();
```

output:

```
[0.50, -6.01, 1.00]
```

we decided, by convention, to adopt the ‘.dat’ extension for our data files.

Another interesting feature implemented in the Vector class is its capability to resize and get chunks from a vector. After resizing, if the size is greater than before, the added positions are set to 0, and if the size is less than the original one, we will get a chunk of indexes range 0 .. new size -1. There is also a method to clone a vector among others.

## Matrix.java

Like in the case of vectors, we can define a matrix as a bidimensional array of double, and have two integers to keep the number of rows and columns; these instance variables could be declare inside our class as follos:

```
private int rows;
private int cols;
private double[][] matrix;
```

Managing properly these three instance variables, we can build the entire class. In the remainder of this section, we are going to describe what we think is relevant; for the rest, you can consult the documentation and pick up inside the code to know about the implementation details.

The Matrix class has methods to perform the most common operations applied to matrices: add, sub, mul, transpose, compute the determinant and inverse of a matrix, and even compute the Euler's rotation matrix; there are also some static methods to do some of these tasks. As in the Vector class, the Matrix class offers support for storage; you can load and store matrices in a well defined format:

```
#rows #columns
a00 a01 a02 ... a0n
a10 a11 a12 ... a1n
a20 a21 a22 ... a2n
...
am0 am1 am2 ... amn
```

where #rows and #columns are integers with the number of rows and columns of the matrix, respectively; and the rest are the double values of the coefficients for our matrix. Having support for storage saves a lot of time avoidin to enter the values by hand. For example, the content of such a matrix could be:

```
3 4
2.0 1.0 1.0 5.0
4.0 -6.0 0.0 -2.0
-2.0 7.0 2.0 9.0
```

if we want to print that matrix to the console we can do that as follows:

```
Matrix m = new Matrix("mymatrix.dat");
m.print() ;
```

output :

```
[2.00, 1.00, 1.00, 5.00]
[4.00, -6.00, 0.00, -2.00]
[-2.00, 7.00, 2.00, 9.00]
```

Another interesting features implemented in the Matrix class are: clone, resize, resize only rows, resize only columns and get chunks from a matrix.

After this brief overview of the Vector and Matrix classes, we are going to present the classes to solve NxN linear systems.

## CramerSolver.java

This technique uses determinants to get the solution vector from a concrete equation's system. We know that there is a method to compute the determinant of a matrix defined in the Matrix class. So, we can apply it to solve a linear system by using the Cramer's rule. Suppose that we have a linear system:

a00 a01 a02 ... a0n = c0
a10 a11 a12 ... a1n = c1
a20 a21 a22 ... a2n = c2
...
...
an0 an1 an2 ... ann = cn

To solve such a system, we first compute the determinant of the coefficient's matrix on the left side; if det(A) != 0, then we can get a solution; otherwise, the system cannot be solved, since the determinant equals to 0.

D = det(A)

once we have the value for that determinant, what we have to do is to substitute the column on the right side ( c0 c1 c2 ... cn ) on each coefficient's matrix column, solve the current determinant after sustitution and divide it by the determinant D which we computed first. If we name:

d0 d1 d2 ... dn

the values for every deterninant, then the X solution vector will be:

x0 = d0 / D
x1 = d1 / D
x2 = d2 /D
...
...
xn = dn /D

The Cramer's rule is ok to solve small systems, that is, composed by a small number of equations; to solve large systems it can be computationally expensive; in that case, it is best to apply another method as we will see.

Our CramerSolver class has two static methods to solve linear systems, one of them takes the extended matrix of the system as parameter, and the other takes the square coefficient matrix as first parameter, and the vector of independent terms as a second parameter.

The usage for this class is easy, for instance, to solve the above system which has 'm' as identifier and print the solution to the console:

```
CramerSolver.solve(m).print();
```

output:

```
[1.00, 1.00, 2.00]
```

since the solve method returns a Vector with the solution for the system, we can concatenate the print() method defined in the Vector class to show the result.

## GaussianSolver.java

This skill consists on finding 0s in the lower triangle of the coefficient's matrix extended with the vector of independent terms.

To find 0s in the lower triangle of the matrix:
the value of the i-th row, j-th column corresponding to the pivot is always multiplied by the value of the coefficient of the i-th row+1 divided by the value of the coefficient of the i-th row changed sign by adding then this ith row to the ith row+1. This process is repeated until the corresponding Upper matrix of coefficients and the corresponding extension of independent terms are obtained. During that process, row permutations may be performed, what is called 'pivoting'. Suppose that we have a system matrix as follows:

a00 a01 a02 ... a0n c0
a10 a11 a12 ... a1n c1
a20 a21 a22 ... a2n c2
...
...
an0 an1 an2 ... ann cn

after applying Gaussian elimination, we obtain the following triangular system:

a'00 a'01 a'02 ... a'0n c'0
0.00 a'11 a'12 ... a'1n c'1
0.00 0.00 a'22 ... a'2n c'2
...
...
0.00 0.00 0.00 ... a'nn c'n

once we have such a triangular system, its solution is straightforward; just begin computing the xn value:

xn = c'n / a'nn

and continue up until x0 substituing the x-th corresponding computed value in the i-th iteration. So, we eventually get the solution vector x0 x1 x2 ... xn.

As in the CramerSolver, the GaussianSolver class has two static methods to solve a NxN linear system, one of them takes as parameter the extended matrix of the system, and the other one takes the coefficient matrix as first parameter, and the vector of independent terms as second parameter.

The usage for this class is as easier as for the CramerSolver, for this example, we will use the method taking two parameters, the coefficient matrix, and the vector of independent terms:

```
m.load("mcoef.dat");
v.load("vcoef.dat");
m.print();
```

```
[2.00, 1.00, 1.00]
[4.00, -6.00, 0.00]
[-2.00, 7.00, 2.00]

v.print();
[5.00, -2.00, 9.00]

GaussianSolver.solve(m, v).print();
```

output:

```
[1.00, 1.00, 2.00]
```

The Gaussian elimination skill is much more efficient than the Cramer's rule to solve arbitrary large systems of linear equations, but as we said before, for small systems, the Cramer's rule is ok.

On the other hand, if what we wish is to use a square coefficient matrix to solve a number of systems for different vectors of independent terms, there is another skill even more efficient than Gaussian elimination.

## LinearSystemSolver.java

As we have seen, Gaussian elimination is an efficient way to solve a NxN system but, there are situations where another method is even more efficient than it. Suppose that we need to solve N systems of linear equations where the coefficient's matrix is always the same, and only the independent terms change. In that scenary, using Gaussian elimination we must compute the triangular system for every solution. There is a skill suitable to improve this issue: LU decomposition. Suppose that we have a square matrix:

```
a00 a01 a02 ... a0n
a10 a11 a12 ... a1n
a20 a21 a22 ... a2n
...
...
an0 an1 an2 ... ann
```

after applying LU decomposition on it, we get:

```
Lower:
1.00 0.00 0.00 ... 0.00
a'01 1.00 0.00 ... 0.00
a'20 a'21 1.00 ... 0.0
...
...
a'n0 a'n1 a'n2 ... 1.00
```

notice that the lower matrix has always all of 1s in its diagonal.

```
Upper:
a'00 a'01 a'02 ... a'0n
0.00 a'11 a'12 ... a'1n
```

0.00 0.00 a’22 ... a’2n
...
...
0.00 0.00 0.00 ... a’nn

Permutation’s vector:
P0 p1 p2 ... pn

now that we have a LU decomposition, to solve a system for a concrete vector of independent terms, first we check if there was any permutation, in case it was some, we have to handle it in order to put the values for the independent terms in the correct position; then, we solve the lower triangular system for the handled vector of independent terms, and the resulting vector is used to solve the upper triangular system; so, we finally get the vector of solutions.

To illustrate that in practice, we are going to apply this skill to compute the inverse matrix for a square one of order 3.

Matrix of coefficients:
[2.00, 1.00, 1.00]
[4.00, -6.00, 0.00]
[-2.00, 7.00, 2.00]

after LU decomposition:

lower:
[1.00, 0.00, 0.00]
[0.50, 1.00, 0.00]
[-0.50, 1.00, 1.00]

upper:
[4.00, -6.00, 0.00]
[0.00, 4.00, 1.00]
[0.00, 0.00, 1.00]

permutation:
[1, 0, 2]

to compute the inverse, we have to solve 3 systems, one for each vector of independent terms which are:

1.0 0.0 0.0 | 0.0 1.0 0.0 | 0.0 0.0 1.0

each solution vector is the i-th column of the inverse matrix; so, after we have solved all of three, we get the matrix:

[0.75, -0.31, -0.38]
[0.50, -0.38, -0.25]
[-1.00, 1.00, 1.00]

which is the computed inverse matrix; to check if it actually is correct, we multiply the original matrix by its inverse to get the identity.

Matrix.mul(m, m_inv).print() ;

output :

```
[1.00, 0.00, 0.00]
[0.00, 1.00, 0.00]
[0.00, 0.00, 1.00]
```

To use the LinearSystemSolver class, you have just to set a square coeficient matrix; once a matrix is set, a LU decomposition for this matrix is done; then, you can solve a system for different vectors of independent terms. For example, suppose that you have an array of vectors of independent terms and you want to solve a system for each one having a LU decomposition for a concrete coefficient matrix:

```
Vector[] varray = ...
LinearSystemSolver solver = new LinearSystemSolver(new Matrix("mcoef.dat"));
for(int i = 0; i < varray.length; i++)
{
Vector x = solver.solve(varray[i]);
// do something with 'x'
}
```

The LinearSystemSolver class has also two static methods to solve a system, one takes the extended matrix of the system as parameter, and the other takes a coefficient matrix as first parameter and a vector of independent terms as second parameter.

```
x = LinearSystemSolver.solve(m);
x = LinearSystemSolver.solve(m, v);
```

You can find an implementation, this time written in C, offering similar functionality at:

https://github.com/ismaelmosquera/linearsys

*imr.media:* As we already have mentioned, this package has two subpackages: imr.media.audio and imr.media.midi, the audio package contains a class to reproduce audio in several formats; since the AudioPlayer class implements the imr.media.Player interface, the audio player must define the methods declared in that interface:

```
void set(String fileName);
void play();
void pause();
void stop();
```

So, a Player has, at least, functionality to set an audio file, play, pause and stop it; when a concrete file is set, it is loaded inside the audio player, after doing some checks to validate it, the player is launched as a background thread so that the system keeps responsive.

The imr.media.midi package implements the needed classes to manage MIDI data and its wrapper classes for the ones in the java.sound API, classes to load and store MIDI files, and a midi player. We are going first to present a brief description for the messages available in the MIDI protocol, and the structure of Standard MIDI files ( SMF ).

A MIDI message has 3 bytes as much, the first byte is called ‘status byte’, and the second and third ( if present ) are data bytes. The most significant bit in the status byte is always set to binary ‘1’, since the message is coded in the higher nibble ( a nibble has 4 bits ) then, we have 3 bits available to codify the different channel voice messages for the MIDI protocol; the lower nibble in the status byte is used to select a concrete MIDI channel, since a nibble has 4 bits, we have 16 channels available to send messages to them; the Genral MIDI standard ( GM ) reserves the channel number 10 to drums; keep in mind that the 16 channels are internally indexed in the range 0 .. 15. A MIDI mesage can have 1 or 2 data bytes, where the most significant bit is always set to binary ‘0’ so, the range of values available is 0 .. 127 = 128 different values. A description for the set of channel voice messages in the MIDI protocol could be as follows:

NOTE OFF: status = 0x8N, data1 = pitch, data2 = velocity.
Remember that the lower nibble in the status byte is used to select a channel; the ‘N’ represents that concrete nibble; this explanation is valid for the rest of messages.

This message turns off a note, which is the value assigned to the pitch ( tone for a musical note ), and the velocity represents how rapid the 0 value is reached.

NOTE ON: status = 0x9N, data1 = pitch, data2 = velocity.
This message turns on a musical note; the pitch is the value for the note; an increment of 1 represents a semitone, since an octave has 12 semitones, we have 128 divided by 12 equals to 10.6 octaves, which is more than enough to cover the entire range for any musical insturment. The velocity is the loudness to reproduce a musical note.

POLY AFTERTOUCH: status = 0xAN, data1 = pitch, data2 = pressure.
This message affects a single note, applying it to a concrete sound we can modify volume, vibrato, timbre, etc by moving an horizontal slider or by pressing down on the key after it “bottoms out”.

CONTROL CHANGE: status = 0xBN, data1 = control type, data2 = value.
Normally, this message is sent by a device different than a keyboard, as can be a pedal, a slider, a wheel, etc. Currently, there are 120 control types 0 .. 119. They can also control sound effects like tremolo or chorus among others. Usually, the minimum value for a concrete control is set to 0 and the maximum is set to 127; for example, volume control has type = 7 and it is applied to a particular channel with minimum = 0 and maximum = 127; a balance control has control type = 8 where a value of 0 corresponds to a maximum left, a value = 64 for center and a value = 127 for maximum right.
An important control type is used to bank selection; this control is sent before a program change message. There are a number of control types but we are not going to describe all of them here.

PROGRAM CHANGE: status = 0xCN, data1 = program.
Sending this message to a particular channel with a program value in the range 0x00 .. 0x7F hexadecimal; 0 .. 127 decimal, causes a change in the selected musical instrument; that is, the following sounds in that concrete channel will be played according with the timbre of the instrument associated with that program number.

Notice that this message has only one data byte so, there is a palette of 128 different instruments available.

CHANNEL AFTERTOUCH: status = 0xDN, data1 = pressure.
The description for this message is the same as for poly after touch, except that in this case the effect applies for the entire selected channel instead for just a single note.

PITCH BEND: status = 0xEN, data1 = MS byte, data2 = LS byte.
The purpose of this message is to make smooth variations in the pitch with a limit of 2 semitones op and 2 semitones down. Usually, this effect is controlled by a wheel or similar. The values for data1 and data2 are as follows:
Maximum down: 0x00, 0x00 hexadecimal; 0, 0 decimal.
Center ( no effect ): 0x00, 0x40 hexadecimal; 0, 64 decimal.
Maximum up: 0x7F, 0x7F hexadecimal; 127, 127 decimal.

## MIDIMessage.java

In our library, the MIDIMessage class is declared as final, which means that it cannot be subclassed. The message is the most basic MIDI data type, our class has only one instance variable to represent a message:

```
private byte[] msg;
```

which is initialized using a code block in the class definition:

```
{
msg = new byte[3];
}
```

there are three constructors:

```
public MIDIMessage(byte status)
public MIDIMessage(byte status,byte data1)
public MIDIMessage(byte status,byte data1,byte data2)
```

and methods to get and set values for a message:

```
public byte get(int i)
public void set(int i,byte b)
```

To add more functionality for this class, there are also static methods to build each of the channel voice messages.

Now that we already have our MIDIMessage class defined, the next MIDI data type can be presented. We know that the MIDI protocol uses messages to comunicate between devices in order to achieve the goal; but it must be a way to indicate to the system when dispatch a message to a device. An event has a channel voice message and a timestamp messured in ticks; there are 3 types of events in the MIDI protocol; the most frequent one is composed just by a message and a timestamp; meta events and system exclusive events will be described later.

## MIDIEvent.java

As you already may guess, our MIDIEvent class has just the following instance variables declared:

```
private MIDIMessage msg;
private long ticks;
```

managin these two variables, we can define an event for the MIDI protocol; the constructor for this class is:

```
public MIDIEvent(MIDIMessage m,long t)
```

and declare and define methods to properly use the data; remember that in OOP, an object has a state, determined by the current values for its variable members, and a behaviour, determined by its methods.

```
public final byte get(int i)
public final void set(int i,byte b)
```

the above declared methods are used to get and set values from and to the message variable; to get and set the timestamp for an event, the follow methods are available:

```
public final long getTicks()
public final void setTicks(long t)
```

since a MIDIEvent can be also a meta event or a system exclusive event, as we said before, we can derive such a classes from MIDIEvent:

```
public final class MIDIMetaEvent extends MIDIEvent
```

since we do not have any interest in subclassing the MIDIMetaEvent class, we declare it as final; a meta event has status byte = 0xFF and an extra buffer of data, so, this class has an instance variable to keep that extra data, we prefered to declare it public for reasons of efficiency.

```
public byte[] data;
```

as in the case of the meta event class, we derive the MIDISysExEvent class as final from MIDIEvent:

```
public final class MIDISysExEvent extends MIDIEvent
```

the constructor for this class is like the one for meta events, and it has also a buffer for extra data:

```
public MIDISysExEvent(MIDIMessage m,long t,int dataSize)
```

the extra data buffer is declared public for the same reasons than the ones for the MIDIMetaEvent class.

```
public byte[] data;
```

the status byte in a sysex event = 0F0 and it must terminate with 0xF7; the next data type we need is a list of events, which in the MIDI protocol is named a track.

## MIDITrack.java

A MIDI track is a list of MIDI events, and also could have a name, so, we can implemented it with the instance variables:

```
private String name;
private LinkedList<MIDIEvent> eventList;
```

our implementation has a constructor and the needed methods to manage the data, we declared the MIDITrack class as final, so that it cannot be subclassed.

```
public final class MIDITrack
```

apart from the constructor, there are methods to set a name for the track, add MIDIEvents to the track, get an iterator suitable to traverse the track, check if the track has tempo events and clear the track; that functionality is enough to cover our purpose.

```
public MIDITrack()
public void setName(char[] aname)
public void addEvent(MIDIEvent e)
public ListIterator<MIDIEvent> getIterator()
public boolean hasTempoEvents()
public void clear()
```

A song, also named sequence, is a list of tracks, it can just have one track or multiple tracks, but this concept will be covered in the next section related to SMF ( Standard MIDI files ), now, we are going to describe our last class to manage MIDI data.

## MIDISong.java

In addition to a list of tracks, a song keeps the number of ticks per quarter note, also named division, as an integer value, we also declare the class as final so that it cannot be subclassed.

```
public final class MIDISong
{
public MIDISong() // constructor

...
...
```

```
private int ticksPerQ;
private LinkedList<MIDITrack> trackList;
}
```

there are, as in the case of MIDITrack, the needed methods to manage the data in a MIDISong: add a track to the song, get an iterator to traverse the list of tracks, get the number of tracks in the song, get and set the ticks per quarter note and clear the list of tracks.

```
public void addTrack(MIDITrack t)
public ListIterator<MIDITrack> getIterator()
public int tracks()
public int getTicksPerQ()
public void setTicksPerQ(int t)
public void clear()
```

now we have all the classes needed to manage MIDI data; there are also wrapper classes to its equivalent ones in the java.sound API; before presenting our classes to load and store SMF and the midi player, we are going to describe briefly the structure for Standard MIDI Files.

This standard was created to solve the isssue of sharing MIDI files among different file systems. A SMF ( Standard MIDI File ) is simply a stream of bytes with a well defined format. Basically, a SMF has two types of what we can call MIDI chumck types; They have a header chunck an one or more track chunck types. A MIDI chunck file header is placed at the beginning of the file as follows:

```
‘M’ ‘T’ ‘h’ ‘d’ < 32-bit value >
length in bytes < 32-bit value >
format < 16-bit value >
# of tracks < 16-bit value >
division < 16-bit value >
```

As we can see, the first are 4 ascii coded letters ( MThd ), following by the length in bytes as a 32-bit value, afterwards, there are 3 16-bit values: format, number of tracks and division. There are three types of formats: 0, 1, and 2; we take in account just formats 0 and 1. A SMF has format 0 if it has only one track and format 1 if it has multiple tracks; format 2 is for files containing more than one sequence but, normally, this format is not used; the division is the number of ticks per quarter note.

After the header chunck, it could be one multi channel track in case of format 0 and a number of tracks in case of format 1. The chunk type for a MIDI track is as follos:

```
‘M’ ‘T’ ‘r’ ‘k’ < 32-bit value >
track length in bytes < 32-bit value >
< track data >
<end of track >
```

The end of track is a meta message with the value 0xFF for status byte and 0x2F for meta type. So, an abstraction of the format for a SMF could have the following structure:

“MThd” < 32-bit value >
0x00000006 < 32-bit value >
< format > < 16-bit value >
< ntracks > < 16-bit value >
< division > < 16-bit value >

“MTrk” < 32-bit value >
< number of bytes in the track > < 32-bit value >
< track data >
< end of track >
“MTrk” < 32-bit value >
< number of bytes in the track > < 32-bit value >
< track data >
< end of track >
“MTrk” < 32-bit value >
< number of bytes in the track > < 32-bit value >
< track data >
< end of track >

...

The track data is the event list in the track, remember that a MIDI event has a message and a timestamp, in SMF the timestamp is messured in ticks; some numbers in SMF are represented as what is called a variable-lenght quantity; these numbers are represented 7 bits per byte, most significant bits first; all bytes except the last have bit 7 set, and the last byte has bit 7 clear; if the number is between 0 and 127, it is thus represented exactly as one byte.

The numbers represented as a variable-lenght quantity are those related to a timestamp, or to extra data in a meta or sysex message.

We identify the kind of message looking at its status byte, if it is in the range 0x8N .. 0xEN then, it is a channel voice message, if its status byte equals to 0xFF it is a meta message and the next byte indicates which type of meta message it is, and if the status byte is 0xF0, it is a system exclusive message which must terminate with 0xF7.

To implement support for MIDI storage, we declare an abstract class which some common functionality and 2 abstract methods; the classes derived from MIDIFile must define this pair of methods, and our MIDIFile abstract class cannot be instantiated directly.

```
public abstract class MIDIFile
```

the abstract methods that must be defined in the derived classes are:

```
public abstract boolean init(String fileName);
public abstract boolean run(MIDISong song);
```

there are also a method and some instance variables declared protected; the method is already implemented and its purpose is to close a file.

```
protected void fclose()  // already implemented
```

```
protected boolean hasFile;
protected RandomAccessFile file;
protected static final int[] bytesPerMsg = {3,3,3,3,2,3,3};
```

we use a random access file just for symmetry; to read a SMF, we have enough reading the file sequentialy but, to write such a file, we need a random access file; the reason will be explained later; the array of integers is used to check for the length of each channel voice message.

## MIDIFileReader.java

We derive the class to read SMF from our abstract MIDIFile class. We must define the init and run methods; the init method just opens a requested file and, if all is right returns true; the run method performs the reading and stores the result in a MIDISong object; if success returns true or false otherwise; do not get wrong about the name ( 'run' ) assigned to this method, we decided to call it from this name because we thought it is a descriptive name for the task it does and, mainly, to enable polymorphism.

The rest of methods in the MIDIFileReader class are declared private; all of them are helper methods to read variable length quantities, 32-bit, 16-bit and byte values. To read the file, we check first for the "MThd" header and, afterwards, we continue reading track by track storing the information in a MIDISong object.

## MIDIFileWriter.java

We derive this class from our abstract MIDIFile class as in the case for reading files. For the init method definition, the only difference is that we open the file for write instead for read. The run method take as parameter a MIDISong object filled with all the needed information to write a SMF; first writes the header file and continues writing track by track all the remainder data. Our class to write SMF has also helper methods to write variable length quantities, 32-bit, 16-bit and byte values; as in the case for read files, these methods are declared private. As we said before, to write a SMF, we need a RandomAccessFile object, the reason is because each track that we write has its length in bytes after the track header ( "MTrk" ), but we do not are able to know that quantity until all the bytes in the track are written; to count that value, a byte counter is incremented each time we write a byte, so, when we reach the end of track, we must turn to the position after the track header and then, write the number of bytes in the current track as a 32-bit quantity.

Apart of pick up inside the code or consult the documentation in the 'doc' folder, you can find a little bit more extensive introduction to the MIDI protocol at:

https://github.com/ismaelmosquera/cwmidi

## MIDIPlayer.java

The MIDIPlayer class has been implemented using classes from the java.sound package which comes with the java API of any JDK. To do that, we just instantiate some classes and made some attachments between them in order to configure a sequencer suitable to play any MIDI file following SMF.

The MIDIPlayer, as in the case of the AudioPlayer class, implements the Player interface so, it must define the methods declared in that interface:

```
public class MIDIPlayer implements Player

public void set(String fileName)
public void play()
public void pause()
public void stop()
```

so, a midi player has, at least, functionality to set, play, pause and stop playing; in addition we can set a MetaEventListener to know, in our case, when the end of track is reached; since both players implement the Player interface, we can do things like the following:

```
Player player;
Player audioPlayer = new AudioPlayer();
Player midiPlayer = new MIDIPlayer();
player = audioPlayer;
...
```

as you might already have guess, that feature enables polymorphism, which will be very useful, among other things, for a SoundPlayer class implementation described below.

## SoundPlayer.java

The SoundPlayer class integrates an audio player and a midi player, so, this class is able to play audio and midi files; in addition, it has support to perform loop, shuffle ( play random ) and select and index from the playing list to reproduce the desired file in the list; since the SoundPlayer class also implements the Player interface, the already mentioned methods in this interface must be defined.

There are several inner classes defined inside the SoundPlayer class which is declared as final so that it cannot be subclassed.

```
public final class SoundPlayer implements Player
```

all the inner classes declared inside the SoundPlayer class are declared private, what means that they cannot be accessed outside the outer class, but those inner classes have permission to access even the private methods and instance variables in the SoundPlayer ( their outer class ).

The PlayingList inner class acts like an automaton tracking the changes produced in the playback list, with all the functionality required to do its task. Each time

a file is set, several checkings are done in order to know the kind of file it is ( a regular file, a directory or a *.m3u file ); once that testing is made, a playing list is built with the required entries and the first item in the list is selected to start playback. To track changes in the playing list, we use the Observer Software Pattern, for that purpose, we declare the following interface:

```
public interface PlayingListListener
{
void listItemChanged(int currentIndex,String currentFile);
}
```

when there is any change in the playing list, for example, the current entry has finished playing, it is notified to the sound player which does the needed task according to the produced change. The Strategy Software Pattern is used to select the kind of player needed, that is, depending if it is an audio or a midi file for the requested playback an audio or midi player is assigned to the current one.

If you look at the ‘examples’ folder, you will see that there is a subfolder named ‘iPlayer’, in this example, we built a very simple GUI using java.swing in order to show the capabilities offered by the SoundPlayer class.

# 3 Conclusions

We design and implemented what we can name a ‘Java Library’; independently of the functionality offered by this java package, we can focus on the way you can code a java library for another purpose. This simple, or may not so simple, java library uses some skills common to any java packed java code, as could be: how comment the desired code to make it suitable to generate HTML API documentation using the ‘javadoc’ tool; also how to organize the code into packages and deploy the library building a *.jar ( Java Archive Resource ) file. Anyway, we hope that the subjects covered in this java package could be useful for an interested reader.

In the case of matrix, vectors and skills to solve linear systems, we have to remember you that the format for the files ( .dat ) are the same that in the C implementation from the ‘linearsys’ repository mentioned above, so, you can share archives between this Java an the C implementation if you wish.

# 4 Appendix

In this brief appendix, we explain how to build the library and examples. There is also a ‘doc’ folder which contains the API documentation built using the ‘javadoc’ tool. To build the library you have two simple scripts available, ‘build.bat’ to use if you are under a MsWindows command line, or ‘build.sh’ in case that you are under a Linux like shell.

To build the examples, you have scripts like the ones to build the library, and also scripts to execute those examples: ‘run.bat or ‘’run.sh’ depending in the kind of shell you are working.

To test the features offered by the SoundPlayer class, we built a very simple GUI using java.swing. We hope that this work can be useful in some way to the reader.

## About The Author

Ismael Mosquera Rivera has a degree in Computer Science by Pompeu Fabra University http://www.upf.edu, Barcelona – Spain, 2004. He also was involved in the Clam project, http://www.clam-project.org ( C++ Library for Audio and Music ); unfortunately, he is affected by a severe eye disease ( Retinitis Pigmenthosa ), which caused him to get totally blind a number of years ago. Anyway, he finally decided to get back to the computer world, so, he installed a screen reader in his old computer. The one he installed is only available for Windows OS, but he got the MinGW compiler which has also a command line shell from where you can have a little Linux in your Windows system. So, all the textual information in a computer can be read, and he began to do research. Among the things he has done until that, the development of this little package encouraged him to write a paper explaining how it was done, and this is the result. He apologizes the reader to the lack of graphics, diagrams and identation in the source code. Anyway, his expectation is that an interested reader can take benefit from this article.