Making Music with Java

An introduction to computer music, Java programming, and the jMusic library.

By Andrew R. Brown

© 2005 Andrew R. Brown

All rights reserved.

Version 1.0

Brown, Andrew R.

Making Music with Java: An introduction to computer music, Java programming, and the jMusic library / Andrew R. Brown

cm.

Includes bibliographical references(p.)

ISBN 978-1-4092-8133-7

Composition (music). 2. Java language. 3. Computer programming. 4. Generative art.

786.7/13

http://jmusic.ci.qut.edu.au/

http://explodingart.com/mmwj/

http://explodingart.com/arb/

Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries.

Contents

Preface 11

Representing Music 12

Integrating Java into your Musical Workflow 14

The Computer as a Musical Instrument 14

Acknowledgements 16

1: Getting Started 17

Installing Java 17

Installing jMusic 17

Java Basics 18

Making it go Bing! 19

Compiling 19

Running 19

Introducing Object-Oriented Programming 20

The Anatomy of a Class 22

The shell of a class 22

Fields 23

Methods 23

Importing existing classes 24

Knowing where to start 25

Accessing methods in other objects 25

Following the Flow of Control 26

Comments 27

JavaDoc 27

Documenting Java Code 28

2: Sound as Digital Audio 31

Digital Audio 31

Java audio 32

Accessing recorded sound 33
Saving samples as an audio file 34
Digital Audio Details 35
Clipping 36
Aliasing 36
Quantization 37
Interpolation 38
Audio File Types 38
Audio Dynamics 39
Volume 40
Normalize 40
Fading 41
Limiting 42
Compressing 42
Java Details 43
Java Data Types 44
Arrays 44
Iteration 45
Casting 46

3: Manipulating Digital Audio 47
Dividing and Combining Audio 47
Mixing 47
Dividing audio files 49
Lopping 50
Inserting silence 51
Envelopes 51
Zero crossings 52
Random recombination 53
Audio Data Parameters 55
Resolution 55
Changing the bit depth 56
Resampling 56
Changing sample rate 56
Changing the number of channels 57
Pitch shifting 58

Interpolation *60*
Pitch shift through interpolation *61*
Resampling details *62*
Java Details *62*
The Math class *62*
A method's return type *63*
And, or, not *64*

4: Music as Digital Data 65
Musical Components and Structure *65*
The jMusic data structure *66*
Notes and rests *66*
Phrases *68*
Parts *70*
The MIDI Standard *71*
Playing many parts *73*
Building Musical Structure *74*
Phrase phasing *74*
Scores *75*
Repetition *76*
Phasing *76*
Piano phase *78*
Musical data structure *79*
Java Structures *82*
Import statements *82*
Constructors *83*
Instance variables *83*
Local variables *84*
Scope *84*
TheGrid: A Rhythmic Phrase generator *86*
TheGrid1 ..... *86*

5: Playing with Polyphony 91
Managing Musical Parts *91*
Making a cannon *93*

Creating the musical data *96*
Making duplicate phrases *96*
Transposing *97*
Structuring the music data *97*
Saving and playing the cannon *99*
The jMusic Mod Class *99*
Arpeggiator *101*
Java Details *103*
JavaSound playback *103*
Keywords *103*
Methods *104*
Passing arguments *104*
Access modifiers *106*
Accessor methods *107*
Method overloading *107*
TheGrid: Playing several parts *108*
TheGrid2 ..... *108*

6: Design and Layout **111**
Viewing and Arranging Music *111*
HelperGUI *112*
Random notes *114*
Start times *115*
Automatically appending phrases *117*
Panning *117*
Dynamics and mixing *118*
Viewing as a piano roll *119*
Viewing as printed data *120*
Viewing as music notation *121*
Scattered phrases *122*
Random walk *122*
Structure *125*
Program Structure – Arranging code into two classes *126*
Java GUI Classes *131*
Introducing the AWT package *131*
Listeners *131*

Introducing the Swing package 132
Components 133
Frames 133
Buttons 133
CheckBoxes 134
Sliders 135
TheGrid: Adding a Visual Appearance 136
TheGrid3 137
TheGrid4 138

7: Building Interfaces 143
Graphical User Interfaces 143
Adding GUI details 145
Adding music 146
Changing variables via the interface 147
Reading MIDI Files 150
Java Details 150
The 'this' keyword 150
The 'static' keyword 151
Static variables 151
Static methods 151
TheGrid: Connecting Music and Vision 152
TheGrid5 152
TheGrid6 159

8: Combination and Recombination 165
Musical Chunks 165
Chords 165
CPhrases 165
Recombination 167
Assembling rhythms 167
Phrases and form 168
Macrostructure & microstructure 171
Appending sections 171
Granular organization 172

TheGrid: Organizing Files and Audio Playback *174*
TheGrid7 ..... *175*
TheGrid with audio *175*
TheGrid8 ..... *175*
TheGrid9 ..... *178*

9: Following Rules 183
Music From Text *183*
Guido *184*
Serialism *187*
Arvoish *188*
Java Details *192*
if then else *192*
while and do *192*
switch statements *193*
TheGrid: Controlling Parameters *193*
TheGrid10 ..... *193*

10: Uncertainty 207
Aleatoric and Stochastic Music *207*
Music from randomness *208*
Probability and music *208*
Gaussian distribution *209*
Random walk *210*
Markov process *211*
Viewing histograms *216*
Noise *218*
Java Randomness and Determinism *220*
Math.random() *220*
The Random class *220*
TheGrid: Random Velocities *221*
TheGrid11 ..... *221*

11: Emergence 223
Inspired by Nature *224*

A Life 224
Evolution 228
Cellular Automata 231
CA rhythms (1 dimensional cellular automata) 232
CA phrases (2D) 236

12: Making Waves 241
Describing Musical Gestures as Curves 241
Sketch 241
Generative Music 243
Music from functions 243
Sine wave melody 244
Fractal music 245
Audio Instruments 247
Audio rendering 249
Seeing the audio data 250
Print instrument 250
SineTest 253
Rendering with samples 254
Rendering audio with any file 255
Audio and the HelperGUI 256
Adding audio to an arpeggio program 256
Java Data Structures 258
Arrays 258
Vectors 259
Inheritance 259

13: Sonic Spectra 261
Music in the Physical World 261
Pitch in hertz 261
Microtonality 262
Tuning systems 263
The Harmonic Series 267
Additive synthesis 267
Score-based spectrum 268

More jMusic Instruments *269*
Instrument-based spectrum *269*
Timbral morphing *272*

14: Interactivity **275**
Real-time music generation *275*
Real Time GUI Control *279*
External Control *286*
MIDI input and output *287*

Appendix A: Constants in the JMC **289**
RhythmValues *289*
Pitches *289*
Dynamics *290*
Panning *290*
General MIDI program changes *290*
General MIDI drum and percussion sounds *290*
Scales and modes *291*
Tuning systems *291*
Audio waveforms *291*
Audio noise types *291*
Phrase position alignments *291*

Appendix B: Methods in the Mod class **293**

Appendix C: Installing jMusic **299**
Install Java *299*
Install jMusic on Linux / Unix *299*
Install jMusic on Mac OS X *299*
Install jMusic on Microsoft Windows *300*

References **303**

Preface

This book introduces you to music making through software development in the Java programming language using the jMusic library. It explains the musical and programming concepts in a co-ordinated way. Programming languages, such as Java, are powerful tools for music making because they allow you describe music structures and sound processing in a precise way, they also enable you to construct software applications that can generate music automatically and/or be played as musical instruments. Therefore, as you make music with Java you can be the composer, sound designer, instrument builder and performer. Making music with Java can be challenging, particularly at first, but it is very empowering, because it allows you to design both your music and the tools which shape your music. This provides a great degree of control, but also responsibility, attributes that are rewarded by persistent study and practice. This book will start you on that path. It will take you on a journey through a variety of music-making activities using Java and the jMusic library. As you work through this book, you will be exposed to a repertoire of well-established compositional and programming patterns and techniques (as well as a few novel ones). You are encouraged to pick and choose from them as contributions to your own music making style and then, hopefully, go on to create new patterns of your own thus continuing the evolution of computer music culture.

This book is for the musician who wishes to learn about Java programming and computer music concepts, and for the programmer who is interested in music and sound design with Java. As such, it assumes little musical or programming experience and introduces topics and issues as they arise. Sections on computer music and programming are interlaced throughout, but kept separate enough so that those with experience in either area can skip ahead as required. You may like to augment this book with specialized books on Java or computer music to follow up topics in more detail than there is space to cover here. For example; *Learning Java* by Patrick Niemeyer & Jonathan Knudsen (2000) or *The Computer Music Tutorial* by Curtis Roads (1996).

The code examples in this book make use of the jMusic library, a music extension to Java co-written by the author. There are instructions for installing jMusic in Appendix C. The jMusic library is a powerful open source toolkit for composing, synthesis, instrument building, analysis, and computer music education. More detailed information about jMusic, including examples and updates, is available from http://jmusic.ci.qut.edu.au and http://sourceforge.net/projects/jmusic/.

The code for examples and their associated files are available for download from http://explodingart.com/mmwj.

Representing Music

Writing music using Java code is simply one way of representing, or encoding, musical information and ideas. Other symbol systems for representing music include common practice notation (CPN), matrix/piano roll displays, guitar tablature and Musical Instrument Digital Interface (MIDI) messages. Each system of representation has strengths and weaknesses, and each enables different methods of music making. Figures i to v show a whole-tone scale represented in different symbols systems.

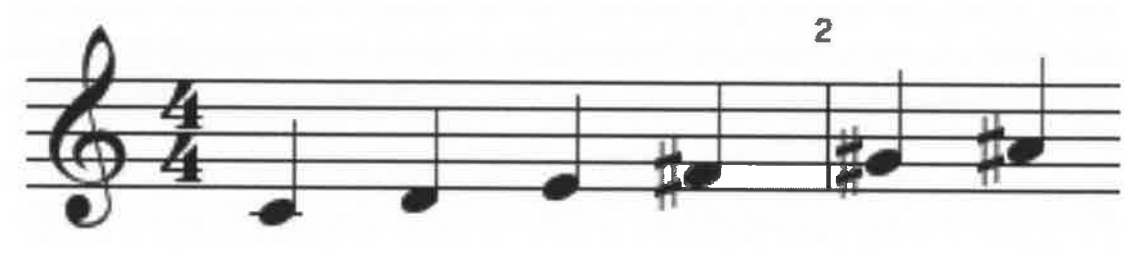


Figure i. A whole-tone scale in CPN.

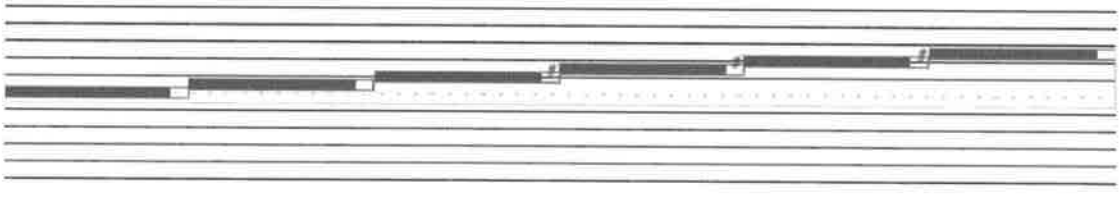


Figure ii. A whole-tone scale in piano-roll view.

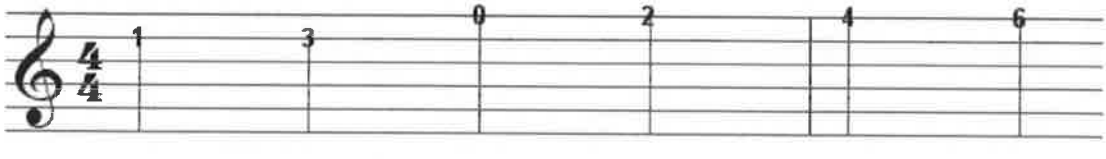


Figure iii. A whole-tone scale in tabalture.

Status	Pitch	Velocity
5	60	100
5	60	0
5	62	100
5	62	0
5	64	100
5	64	0
5	66	100
5	66	0
5	68	100
5	68	0
5	70	100
5	70	0

Figure iv. A whole-tone scale as MIDI data.

```
Phrase phr = new Phrase();
for (int i=0; i<12; i = i+2) {
    phr.addNote(new Note(60 + i, 1.0));
}
```

Figure v. A whole-tone scale as Java code.

Music representations have a number of purposes. They assist musicians by externalizing musical thinking, which allows for further reflection and action. They are a convenient method of storing music, either to assist memory or for archival purposes. And, they enable musical communication in circumstances where sonic communication is not practical or desirable. Normally, musical communication is between humans, and so representation systems such as CPN and guitar tablature are designed to be human readable. When we want to communicate with computers we use systems such as MIDI that are designed to be machine readable, and therefore may look confusing to us. Java, and most computer languages, are designed primarily to be human readable but in many ways reflect the underlying machine structures, thus enabling efficient translation to machine code. Java uses English-like keywords and mathematical symbols that are familiar enough to us, such as `class`, `public`, `void`, `double`, `+` and `/`, but relies heavily on logical structures that un-

derpin computer system design, including loops, jumps between sections, and strict syntactic rules. Java keywords have special meanings and are reserved for those purposes and should not be otherwise used. The jMusic libraries, which extend the Java language, provide additional words and structures based on musical practices to aid in the representation of music in Java. Additional keywords in jMusic include data types such as `Note`, `Phrase`, `Part` and `Score`, and constants such as `QUARTER_NOTE`, `FORTE`, `C4`, and `PENTATONIC_SCALE`. Additional musical structures or processes added to Java by jMusic include `repeat`, `elongate`, `transpose`, `invert`, `accent` and `crescendo`. So, representing music in Java combines elements of our linguistic, mathematical, logical and musical histories, even though at times the meaning of terms in Java may differ from your previous understanding of them. The way in which these elements can be combined to make music is vast but also constrained and so, like music representation systems from the past (and those yet to be invented), Java has its syntax and grammar which need to be understood in order to become musically expressive with it.

Integrating Java into your Musical Workflow

To make the most of your Java-based music making skills it is important to think about how they can be integrated into your existing music making practices and tools. Your music making activities probably already include a number of software components, perhaps a MIDI/Audio sequencer, hard disc recording software, and software samplers. The examples in this book include saving the Java-based music as standard MIDI files and audio files. These can be imported into your other software for further elaboration. This means that you don't have to immediately create an entire musical work in Java, but you can create small sections or components that are combined with other elements into a final whole. In this way your music making with Java adds to and leverages your existing musical skills. This interoperability between software works both ways because jMusic and Java include routines to read MIDI and audio files as well. Therefore you might start working in other software, then import your files into your Java applications for manipulation or enhancement, then export them back out to other software environments for arrangement and mastering. As your Java skills increase you can find an optimal balance between making music with Java and other software or hardware devices at your disposal, using each in the way that maximizes its features and potential.

The Computer as a Musical Instrument

When making music with Java it is useful to approach the computer as a musical instrument. As an instrument the computer becomes a vehicle through which you express musical ideas. Like all music technologies, instruments (including the computer) have tendencies, or pathways, for the musi-

cian to follow. For example, most acoustic instruments are easier to play in some ranges than others, and many orchestral instruments play only monophonic music. However, these are tendencies rather than limitations, the skilled performer can play notes over a wide pitch range and techniques such as double stopping and multiphonics enable homophonic textures from monophonic instruments. The computer has limitations imposed by the hardware and software but, when viewed as a musical instrument, you can look beyond these limitations or considerations of the computer as simply an efficient tool, to see that the computer is a partner in the music making process. Viewing the computer as an instrument recognizes that music making moves beyond simple human-directed activity to become collaboration with the computer—by assisting, the computer has an influence. However, the more responsibility a computer music system has over the music production, the greater the expectations on the composer using the system. This might seem counter-intuitive but put more simply; a greater emphasis on the unintelligent (even perhaps unskilled) computer requires more of the (presumably) intelligent programmer and composer.

As with traditional musical instruments, such as the guitar, intimate engagement with the computer as an instrument increases with your familiarity with the computing medium - as you increase your programming skills. As with any musical instrument, the greatest satisfaction will result when you learn to *play* the computer well. This involves studying the principals involved, regular practice, and an immersion in the computer music culture through listening, reading and discussion.

Musical instruments are normally associated with their use in real-time performance. The computer can be used in this way, however, you can approach the computer as an instrument in non real-time interactions also; such as the algorithmic computation of musical parts which can take from a few seconds to a few minutes depending upon the amount of data to compute. Engagement is maintained through these drawn out processes when there is sufficient interest in the results of the processes. Another indication of your engagement with the computer as an instrument is the effectiveness with which you can express music with it, which requires a good operational ability and fluency with the music representation system. When these are achieved you can operate directly with the computer to create music without the need for deliberate translation from some other musical language or way of thinking. You just play the computer.

This book focuses more on music composition than performance and it may seem that for composers the computer acts more like a tool or medium, while this is true, the idea of the computer as an instrument is still valid because it denotes a special relationship often witnessed between performer and instrument that can still be found between composer and instrument. In this book you will be introduced to the areas of skill and knowledge necessary for you use the computer as a musical tool, as a medium for musical communication and, as well, you are encouraged to approach the computer as a musical instrument.

Acknowledgements

I would like to thank those in the jMusic community who have contributed to the development and robustness of the library and the tutorials that underpin this book. Worth particular thanks in this regard are the efforts of Andrew Sorensen (co-creator of jMusic), René Wooller (who assisted with proof reading), Andrew Troedson, Adam Kirby, and Tim Opie (who contributed substantially to development of the jMusic library).

Concept Quiz

1. What attributes are suggested as helping gain control over Java as a music making tool?
2. What is the relationship between Java and jMusic?
3. What are the suggested purposes of music representations?
4. What is special about Java keywords?
5. What features does a computer share with other a musical instruments?
6. Improvements in what skill increases your familiarity with the computing medium?
7. What is meant by the term 'engagement' in this chapter?

1: Getting Started

This chapter provides a whistle-stop tour of the Java language, the jMusic library, and installing and running Java programs. If you are already familiar with Java programming feel free to skim this chapter and move directly to chapter 2 where music and audio issues are discussed. If you are new to Java programming then this chapter will provide an overview, but may appear somewhat confusing. Take your time, read carefully, and consult a more detailed Java text when more clarification is required.

Installing Java

Java has two main parts that can be installed (often with the one installation process), the Java Virtual Machine (JVM) which runs Java applications and the Java Development Kit (JDK) which includes the tools required to create java programs. When you do a complete Java install using Sun's installers. Both the JVM and the JDK are installed, and possibly an Integrated Development Environment (IDE) as well, depending upon your choice.

The JVM and JDK are specific to each operating system and you need to install the appropriate version for your computer. Sun Microsystems, the inventors of Java, provide installers for Java on many platforms including Windows and Linux. Apple Computer provide Java tools for Mac OS X and a Apple and number of Linux distributions provide these tools as a standard part of the operating system.

For download and installation instructions visit the relevant link:

- Sun Microsystems: http://java.sun.com
- Apple Computer http://apple.com

Installing jMusic

jMusic is a library that extends Java to provide increased music functionality. The jMusic library must be installed separately from Java, and is usually stored separately from other Java classes. Some set up is required after installation to make Java aware of the jMusic library. This involves setting the classpath; the location(s) on the hard drive where Java should look to find classes and class libraries.

See the details on installing and setting up jMusic in Appendix C and visit the jMusic web site, http://jmusic.ci.qut.edu.au, for further information about installing and using jMusic.

Java Basics

Java is a computer programming language developed and owned by Sun Microsystems. In many respects it is quite similar to other programming languages such as C++, C#, and even Fortran. Java has become quite popular for a variety of good reasons, some of which make it useful for music programming and others which are sometimes a hindrance. In this section I will outline a number of these Java features.

Java programs run on the Java Virtual Machine (JVM), a software layer that sits above the operating system. In this respect it is similar to Lisp and C# but different from C++ and Fortran. JVM's are written for each operating system and, as a result, Java programs only have to be created once and can run on any platform with a JVM. This portability of Java programs is particularly useful to programs distributed over the internet, but is also convenient for people working in computing environments with a mix of platforms. As for example in music communities where it is common to find both Windows and Macintosh users in large numbers, and some musicians running Linux or Unix.

Java is an interpreted language. This means that the Java compiler generates byte code that the JVM can translate into instructions for the local operating system. This interpretive step means that Java programs may be slower to run than those written in C++, for example, but in most cases this presents little practical problem and as computers increase in processing power performance problems due to interpretation requirements decrease.

Java is a modern language that employs many leading architectural features. It is object-oriented, dynamic, robust, secure and multithreaded. While there is not space to explain each of these features in detail, the message is that your Java programs will be quick to write by leveraging other people's efforts (such as using the jMusic library), will take good advantage of your computer's capabilities, will be well behaved internet citizens, and should be relatively robust.

There is a great deal of support for Java as a teaching language. Dozens of "how to program" books have been written for Java and many Universities teach computer programming using Java. You should have no trouble finding help with your Java questions as you work through this book, and I encourage you to have good Java reference by your side (or on your computer) to supplement the explanations in this book. You may also supplement that material in this book with the online Java tutorials at http://java.sun.com.

Making it go Bing!

Let's get straight into it and write our first Java music program with jMusic. Type the code below into a text file and save it as *Bing.java*.

```
import jm.music.data.Note;
import jm.util.Play;

public class Bing {
    public static void main(String[] args) {
        Note n = new Note(60, 1.0);
        Play.midi(n);
    }
}
```

Compiling

To compile the program using a terminal (command line) application use the java compiler, `javac`. Open a new terminal window and navigate (using the cd command in Unix/Linux systems) to the directory that contains your `Bing.java` file. Type `javac Bing.java`, and hit return. This should create a file called `Bing.class` in the same directory. If there are errors, then check for typo's in the file or consult Appendix C to ensure Java and jMusic are installed correctly.

Running

To run the *Bing* program from the terminal, use the *java* application. Type `java Bing`, and hit return. Notice that the `.class` suffix is not required. A single note should sound from your computer's speakers. If you are using an IDE such as *NetBeans* or *jBuilder* for your Java programming, consult the manual about how to compile and run you programs.

This may seem like a lot of trouble to produce just one note. As you work through this book it will become clearer how to leverage algorithmic processes to shift the balance of effort to produce vast quantities of music with modest amounts of code. Here is an example to whet your appetite, that uses the simple algorithmic processes of repetition and randomness and the power of Java's `for` loop to generate hundreds (potentially thousands) of times more music than the *Bing* program with just three more lines of code.

```
import jm.music.data.*;
import jm.util.Play;
```

```
public class BingLoop {
    public static void main(String[] args) {
        Phrase phr = new Phrase();
        for (int i=0; i< 200; i++) {
            Note n = new Note(60+(int)(Math.random()*24), 0.125);
            phr.addNote(n);
        }
        Play.midi(phr);
    }
}
```

Programming Practice

Create `BingLoop.java` file from the above code. Compile the java file with the terminal command `javac BingLoop.java` to create a class file. Run the class file with the terminal command `java BingLoop`.

In the remainder of this chapter the details of how these, and other Java programs, work will be unpacked as the basic concepts of Java programming are introduced.

Concept Quiz

1. What is the difference between the JVM and JDK?
2. What does a *classpath* specify?
3. What does it mean to say that Java is an *interpreted* language?
4. What are the URLs of the Java and jMusic web sites?
5. What is the *javac* program?
6. Does the *java* program execute a .java or .class file?

Introducing Object-Oriented Programming

It was mentioned earlier that Java is an object-oriented language. This design feature involves a particular way of structuring a program and has implications for the way Java programs operate. This section will focus on a few object-oriented concepts that are important to get clear from the start, and others will be discussed in more detail later in the book. Thinking about the world in an object-oriented way is based on the notion that things can be categorized and described, that we objectify things in the world. Categories of objects are often hierarchically arranged. Take, for example, automobiles where there is a category of objects we call cars. All cars are part of the

broader category we call vehicles, which can include bikes, trains, and so on. Vehicles are objects that are part of the transportation category, along with airplanes, boats, and the like. This partial object-oriented structure is depicted in figure 1.1.

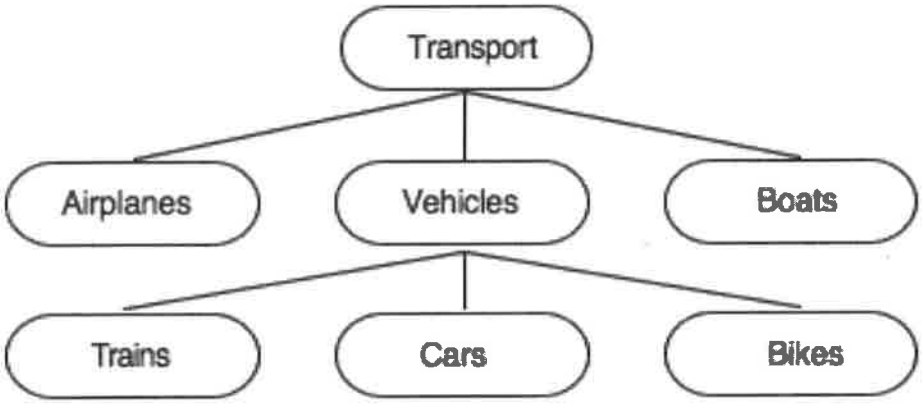


Figure 1.1. An object hierarchy.

At each layer of the hierarchy there are features that describe the objects at that level. For example, all transportation objects share the feature of movement and a mechanism of propulsion; therefore a house does not normally fit in that class of object, whereas a skateboard does. Vehicles, add more precise features, such as the fact that they move on land. The class of cars inherits those transportation and vehicles features and adds more specifics. For example, cars generally have four wheels, are designed for domestic transportation, and so on. Similarly, we can subclass cars into more specific classes of objects, those with four wheel drive, those with six cylinder motors, etc. This object categorization process can be extended as required.

Using object-oriented programming languages a description of the features of an object is called a *class* declaration. A class inherits all the features of the classes that it extends, it describes the types of *data* it will hold, and contains *methods* for manipulating that data. A class description is like a blueprint for objects of a certain type. For example, the four wheel drive car class is a template for producing those vehicles, each of which may have a distinctive set of data values, such as a different brand, model, color, and motor size. These individual objects (four wheel drives) are said to be *instances* of the four wheel drive class, they are specific exemplars or versions of the class.

In Java, we write classes that describe objects. These classes define what the objects will do. The *new* keyword is used to create an instance of a class. A class includes a *constructor* that is run when the object is created, and includes *fields* (variables or constants) that hold data about the object, and includes *methods* which are descriptions of what the class can do to data.

The creation of instances of a class is called *instantiation*. As a musical example, let's look at the `Note` class from jMusic. There is only one `Note` class, and it describes the category of objects we call musical notes. `Notes` have variables including pitch, rhythmValue, and dynamic, and each variable has a data value. `Notes` have methods to act on those data, like `setPitch()`, and methods that act on the entire object, like `copy()`. A piece of music can have hundreds of notes, but these are all instances of the one `Note` class. Each note object (instance) is unique but they all share the same features and behavior. The single `Note` class can be used to create any number of note objects. Now, it's time to see how a Java class is organized.

The Anatomy of a Class

Java programs are made up of one or more classes. As mentioned above, a class is a template or blueprint for creating objects. When programs are run objects are created from these blueprints, these objects are called *instances* of the class and are created using the *new* keyword.

The Java code files are text files designed to be human readable and writable. Each Java file you write is compiled into a class file of the same name. Class files are only machine readable. By convention class names, and their files, start with uppercase letters, use uppercase letters at the start of each new word in multi-word names and do not have any spaces. The source file *MakeMusic.java* produces a *MakeMusic.class* file when compiled. The class files are binary files, designed to be executed by the JVM. You use the source files to write your program and the class files to run your program.

The shell of a class

```
public class MakeMusic {

}
```

Java classes are declared with the `class` keyword, followed by the class name, in this case `MakeMusic`. When the file is saved it must have the same name of the class with a .java suffix, in this case *MakeMusic.java*. The *public* keyword indicates that this class is available for use by any other classes. There are other keywords that can be used instead of public to limit the availability of this class, these include *protected* and *private*. The code within the class is delimited by the opening and closing curly brackets {}. Other sub-sections of the program can also be bounded by these brackets.

Fields

Classes can contain data. Data is stored in fields, usually these are *variables* whose values can change, but they may be *constants* whose values cannot change. The *MakeMusic* class has one variable, a `Note` named `n`. Additionally, the variables within the object `n` can be assigned values when it is created by passing arguments. In the example below, the arguments are a pitch of middle C (C4) and a rhythm value of one beat (QUARTER_NOTE). In summary, `n` is a variable of type `Note` belonging to the *MakeMusic* class, and the note `n` contains variables including a pitch of C4 and a rhythm value equal to a quarter note.

```
public class MakeMusic {
    Note n = new Note(C4, QUARTER_NOTE);
}
```

Methods

A class can also have methods, which are sections of code that describe a process or algorithm. Methods usually act on the data within the class. By convention method names start with lower case letters – this makes them easy to distinguish from class names. In this example a method called `playMusic()` has been added to the class.

```
public class MakeMusic {
    Note n = new Note(C4, QUARTER_NOTE);

    public void playMusic() {
        Play.midi(n);
    }
}
```

Method names are followed by brackets "()" which may contain indications that data (arguments) are required by the method. Methods must also indicate the type of value they will return (*int*, *double*, *String*, *Note*, etc.), or *void* if they return no result, as in this example. (Types will be discussed in more detail later in the book.) This method does not accept or return any values, it is self-contained; it simply plays the note object `n`.

Importing existing classes

One of the advantages of modern computer languages is that rather than having to write all the required code there are many libraries of existing classes and methods you can draw upon. jMusic is one such library, and there are many that are part of the Java language as well.

To access existing classes, or groups of classes called *packages*, you *import* them at the beginning of your program.

```
import jm.util.Play;
import jm.music.data.*;
import jm.JMC;

public class MakeMusic implements JMC{
    Note n = new Note(C4, QUARTER_NOTE);

    public void playMusic() {
        Play.midi(n);
    }
}
```

There are three `import` statements in this example, they point to particular sections of the jMusic libraries that are used by this class. The first imports the `Play` class. It is in the *util* package, inside the *jm* directory. Next, the `jm.music.data` package is imported. The asterix (*) indicates that all classes in the `data` package should be available for use, these include the `Note` class. Lastly, the JMC (jMusic Constants) class is imported. This class has many fields that are abbreviations and synonyms that map standard musical terms to numbers required by jMusic. Being Java constants, the values of these fields cannot be changed. For example, `C4` is a JMC constant equaling the value 60, and `QUARTER_NOTE` is a JMC constant with a value of 1.0. The use of constants makes the code more human readable, and save you having to remember arcane numbers when you would rather think in musical terms.

Additionally, the class declaration *implements* the JMC class. This is required because the JMC is actually an *interface* rather than a normal class. In Java an interface is a kind of abstract class that declares variables and methods but does not implement the methods. When implementing an interface, a class inherits all its fields and should override any abstract methods. In this case we use the JMC interface to inherit many fields with English-like names that can be used in the code to make it more musically meaningful when read, and the JMC interface does not have any method definitions to override. The important thing to understand for now, is that implementing the JMC allows you to use musical terms throughout your code making it much easier for musicians to understand.

Knowing where to start

Every Java program must have one `main()` method. The method can be contained in any one of the program's files. The main method is where the program begins execution and everything follows from that point. In a simple program like the *Bing* example the `main()` method can be the one and only method. In this example we'll see how the `main()` method can be in a different file.

```
public class MyProgram {
    public static void main(String[] args) {
        MakeMusic musicObject = new MakeMusic();
        musicObject.playMusic();
    }
}
```

The **`main()`** method has an additional keyword, *static*. This indicates that there can only be one copy of this method and when instances of this class are created this method is not duplicated. This is the formal way of ensuring that there is only one `main()` method and that a program is not confused about where it will begin.

The `main()` method always accepts one *argument*, a list of words, a string array `String[]`, that can hold a list of alphanumeric characters, called `args` (short for arguments). This allows words to be passed to the program when it beings. These words may be used to provide data for the program to work on or parameter settings that can be specified each time the program runs. We do not use this feature of the main method in our program, but Java requires that the `main()` method accepts this argument. The main method in this example creates a new *MakeMusic* instance then calls it's `playMusic()` method.

The *MyProgram* class creates an instance of the *MakeMusic* class named `musicObject`. It then calls (runs) the `playMusic()` method of that object. How is this 'remote' control of another class achieved?

Accessing methods in other objects

Notice that the syntax for calling the `playMusic()` method uses a 'dot' between the instance name and the method, `musicObject.playMusic();`. This can be read as, call the `playMusic()` method from the `musicObject` instance of the *MakeMusic* class. Or, more simply, call `musicObject's playMusic()` method.

Programming Practice

To run this example program, create two new files. Into the first type the final version of the *MakeMusic* code and save it as `MakeMusic.java`. In the second text file type the *MyProgram* code above and save it as `MyProgram.java`. Compile both Java files and run the *MyProgram* class. You will get the same result as the *Bing* program earlier, but it has been achieved in an object-oriented way. This, somewhat convoluted object oriented program structure will pay dividends as the programs become more sophisticated.

Following the Flow of Control

The way the running of the program unfolds is depicted in figure 1.2. When the *MyProgram* class is run the *main* method is executed first. It creates a new instance of the *MakeMusic* class which automatically results in the creation of a new `Note` object called `n`. The main method then calls the *playMusic* method which creates a single note and plays it using Java's software synthesizer (more on these musical details later).

```
public class MyProgram {
    public static void main(String[] args) {
        MakeMusic musicObject = new MakeMusic();
        musicObject.playMusic();
    }
}
```

```
import jm.util.Play;
import jm.music.data.*;
import jm.JMC;

public class MakeMusic implements JMC{
    Note n = new Note(C4, QUARTER_NOTE);

    public void playMusic() {
        Play.midi(n);
    }
}
```

Figure 1.2. Flow of control between the program's two classes.

Comments

Like most other programming languages Java lets the programmer write remarks into the source code. These sections are called comments, and are ignored by the compiler. There are a couple of ways that comments are indicated.

```
// Single line comments start with two back slashes.
/*
        Multiple line comments start with an astrix and star and
        end with the opposite, an astrix and a slash.
*/
```

JavaDoc

One of the most helpful sources of information about Java is the documentation. Learning to use Java easily relies upon becoming familiar with accessing information from the java documentation. There are too many classes and methods in the Java libraries, let alone adding jMusic libraries, for most people to ever remember, and so a successful strategy is being aware of the general capabilities of the language then looking up the fine details in the documentation as required.

Fortunately, there is a standard format for Java documentation, therefore once this is understood finding information about new libraries and packages will not be difficult. The Java documentation is in html format and should be installed alongside Java, but can also be accessed online or downloaded from http://java.sun.com. Below, in figure 1.3, is an example of the Javadoc index page for Java 2 SE.

On the left are index panels, the top one selects a particular package and the lower one displays all the classes in the selected package. By default 'All Packages' is selected and so all the Java classes are shown in the index in alphabetical order. Selecting a class in the index panel displays it in the larger right panel. There are a large, and increasing, number of classes in Java and, as a result, the documentation can appear overwhelming at first. The trick is to understand the simple organization of the documentation and for the time being ignore the classes you have not yet encountered.

The organization of the class documentation that displays in the right panel is always similar. First is the class name and its package location in the library, after this is a short description of the class and how it can be used. Following the description is a summary of the fields (variables and constants), constructors, and methods. These summaries are one-sentence descriptions with a column to the left that indicates the type of data involved (the type of the field or the return type of a method). The summary may also indicate fields and methods that have been inherited by the class

from any super (parent) classes. Finally in the documentation page are detailed descriptions of the fields, constructors and methods.

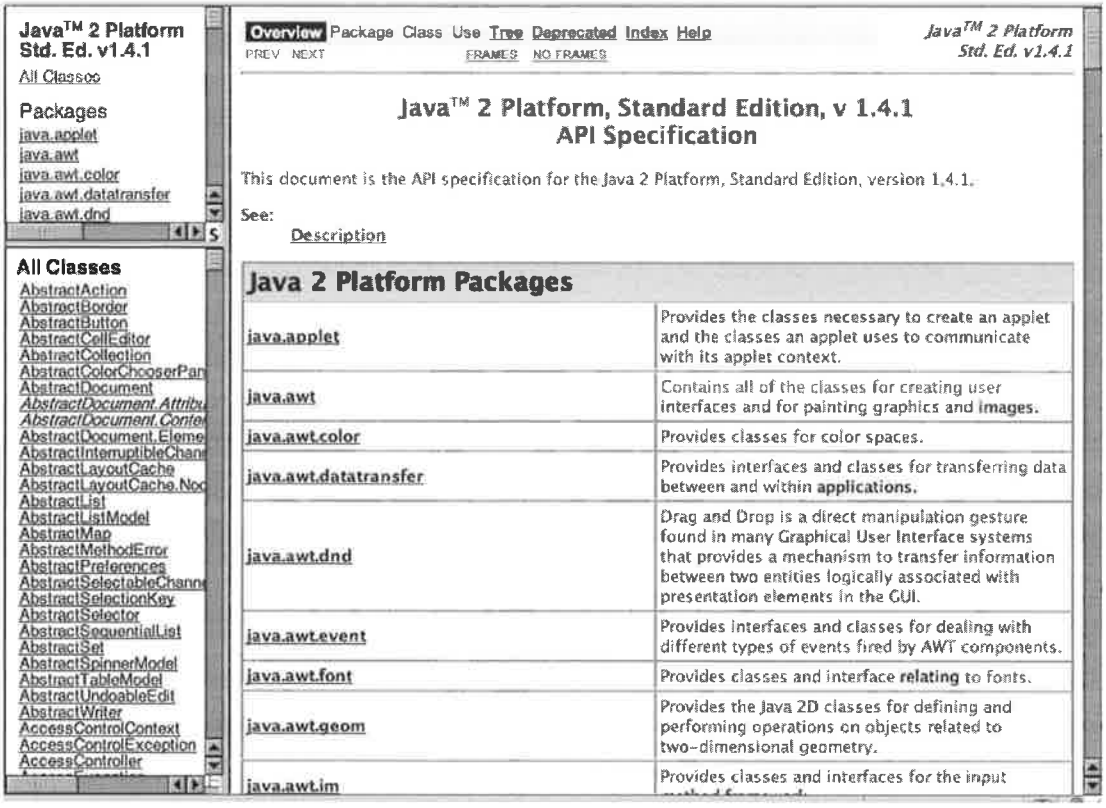


Figure 1.3. The java documentation index page.

It is good programming practice to have a browser with the documentation open at all times while you are working and to access it regularly when you need reminding of the details of a class or method. There is similar documentation for jMusic. It can be downloaded from `http://sourceforge.net/projects/jmusic/` or viewed online from the jMusic web site, `http://jmusic.ci.qut.edu.au/jmDocumentation/index.html`.

Documenting Java Code

A neat feature of the Java documentation is that it is semi-automatically created from comments in the source code. While this is not the place for a detailed discussion of Java documentation procedures, the simple story is that all documentation comments that start with `/**` (rather than the normal `/*` or `//`) and ends with a `*/` are included in the Java documentation. A program called

`javadoc` searches through source files and builds the HTML pages. Further details on Java documentation procedures can be found at http://java.sun.com/.

Concept Quiz

1. What is an object oriented approach?
2. Objects derived from the definition are called an _____ of it.
3. There are two types of fields that hold data within objects, what are they?
4. What are sections of code that act on data called?
5. What does the public keyword indicate?
6. What does the import keyword indicate?
7. When we implement the JMC what does that add to the program?
8. Which method is always the first to be run?
9. What will the file name be for a program that starts, public class Test implements JMC ?
10. Where does all the text for the Java documentation come from?

2: Sound as Digital Audio

While symbolic representations of music, including notes and phrase, assist in compositional tasks and in thinking about large musical structures. In the end the computer produces a sonic outcome from a digital audio stream based on these structures. In this chapter we will examine how sound is represented as digital data and how that data can be directly manipulated to vary the sound. Along the way we will learn about important Java language features and how they can be utilized for digital signal processing. In this and other early chapters of this book the symbolic and sample-based representations of music are being discussed separately, but in later chapters they will be interconnected into a complete computer music system with even more potential for music making than either of the constituent parts.

There are two main sources of digital sounds, recordings and synthesis. Recorded sounds are captured from the acoustic world with microphones and we tend to treat them at a macro level, as entire files or wavetables. Synthesized sounds are generated by an algorithmic process and can be produced (computed) as required. In this book, we will look at how to deal with recorded sound material and how to generate sounds via synthesis. But, whichever way a digital sound is created, at a fundamental level they are all the same kind of data, and so it is useful to briefly describe the basics of digital audio.

Digital Audio

Computers are digital devices, meaning that they deal with number or digits. The reason they work this way is because fundamentally they are built with switching devices, called gates. These gates have one of two states, on or off. As a result, all numbers in the computer have discrete values rather than continuously varying (analogue) values like other media; such as water volume, electrical current or a magnetic field. Digital audio data, like all other computer data, is simply a stream of numbers. When recording a sound, an analogue-to-digital converter (ADC) measures the amplitude (loudness) of the sound input thousands of times every second in a process called sampling, and the list of sample values is the digital audio data. When synthesizing digital sound the sample values are computed rather than measured. In the digital domain there are two considerations with respect to audio quality, sample rate and resolution.

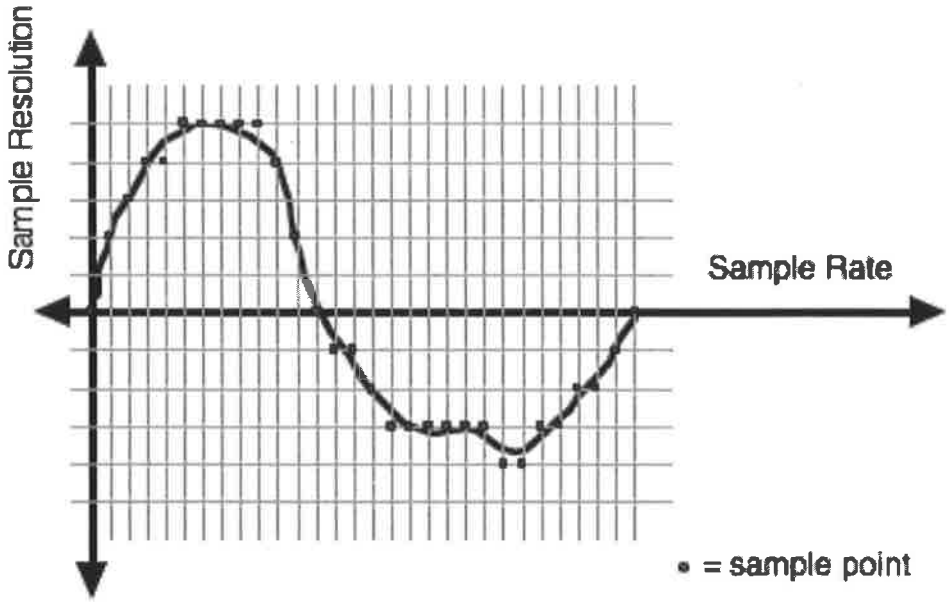


Figure 2.1. Converting an analogue signal to digital values.

The more frequently samples are taken the more accurately the sound is represented, remembering that an analogue/acoustic sound is infinitely variable, and this measurement regularity is called the sample rate. Typical sample rates vary from 11 to 96 thousand samples per second, but the most common sample rate is 44100 samples per channel per second; as used for CDs.

The resolution of a sample refers to the accuracy of measuring its amplitude. The finer the resolution the higher the audio quality and the greater the dynamic range. In computer sound the resolution is generally related the number of bits used to store the sample. For example, in a sound with 8 bit resolution the amplitude scale has 256 steps, in a 16 bit sound this increases to 65,536, and for a 24 bit sound 16,777,216 steps. When the sample rate and resolution are increased the sound quality improves, but the memory required to store the sound data and the time taken to process them also increase. A trade-off between quality and file size is always in play, and the best choice depends upon the importance of the features in each circumstance. For example, voice over IP applications trade quality for low bandwidth data transmissions, while professional music studios trade hard disk storage costs for undistorted audio clarity.

Java audio

Java provides classes for reading and writing from audio files in the AU, WAVE and AIFF formats. jMusic provides some simple read and write utilities that deliver and accept audio data as an array

of floating point numbers. Specifically, the `Read.audio()` and `Write.audio()` methods, in the `jm.util` package. The reading method operates in two ways, one returns an array of floats, the other accepts an existing array and fills that array. Both methods require the name of the file to be read.

Note that in programming terminology, a 'float' means a floating point number; that is a number with decimal places such as 12.345.

```
float[] data = new float[1];
Read.audio(data, "Welcome.au");
```

or

```
float[] data = Read.audio("Welcome.au");
```

The writing method has only one argument format, it requires an array with the audio data in it and the name for the saved file.

```
Write.audio(data, "SavedAudio.aif");
```

Accessing recorded sound

A program to access an audio file simply needs to wrap the `Read.audio()` method with a class structure. Here is one of the simplest jMusic classes that imports and uses the `Read.audio()` method.

```
import jm.util.*;

public final class ReadAudio {
     public static void main(String[] args){
        float[] data = Read.audio("Welcome.au");
    }
}
```

During the reading of the audio file, the **Read** class prints some feedback out to the terminal window. This includes information about the file.

```
-------------------- Reading Audio File ---------------------
File 'Welcome.au' read in. Details:
Channels = 1 Samples per channel = 47492 Sample rate = 44100 Bit depth = 16
-------------------------------------------------------------
```

As a final code example in this section, we'll manipulate the samples before they are written. A simple change is to reverse the order of the samples, thus playing the sound backwards.

```
import jm.util.*;
/*
* Read an audio file into an array and write out
* a reversed version of the array.
*/
public final class WriteReverseAudio {
    public static void main(String[] args){
        float[] data = Read.audio("Welcome.au");
        // reverse the order of samples
        float[] reversed = new float[data.length];
        for (int i=0; i< data.length; i++) {
            reversed[data.length - i - 1] = data[i];
        }
        // write
        Write.audio(reversed, "ReversedAudio.wav");
    }
}
```

Programming practice

Type up each of these short classes in a text editor, save, compile and run them to hear the results. You may also like to experiment with some different processing of the audio data after reading and before writing it. There is no end to the types of signal processing functions you can apply.

Note that the JavaSound library in Java also has classes for reading and writing files, but these are not interoperable with jMusic. The jMusic library was developed before JavaSound existed (in the days of Java 1.0 in the late 1990s) and while it has integrated many of JavaSound's MIDI features jMusic audio processing remains somewhat parallel to JavaSound audio until the final input-output stage.

Digital Audio Details

When audio data comes from recordings, the process of digital recording involves an change from continuous analog signal to a quantized digital one; a process called sampling. The device that does this is called a analogue to digital converter (ADC). Representing waveforms as digital samples is often referred to as pulse code modulation (PCM) data. The sample data stored in this way is often

not high quality). Hence the highest frequency capable of being sampled by a particular sampling rate is half the rate, and is called the *Nyquist* frequency. Given that human hearing ranges up near 20,000 hertz a sampling rate of more than 40,000sps is required. This consideration played a large part in determining the 44,100sps sampling rate used on CDs. The effect of using a sample rate lower than twice the highest frequency in the signal spectrum is *aliasing*. In effect, the frequencies above the Nyquist 'fold over' and re-emerge as low frequency components that are usually non-harmonic. The result is an undesirable change in timbre. Aliasing is usually prevented by the use of low-pass filters in the sampling process to prevent frequencies that are too high from entering the process.

Quantization

When measuring a sample, a fixed number of bits (binary digits) are used to store the value. In an 8 bit sample, for example, 256 values are available to represent the dynamic range. An analogue recording source will have an infinite number of values which are rounded or truncated to the closest available digit. This rounding is *quantization* which introduces some inherent inaccuracy, referred to as quantization error.

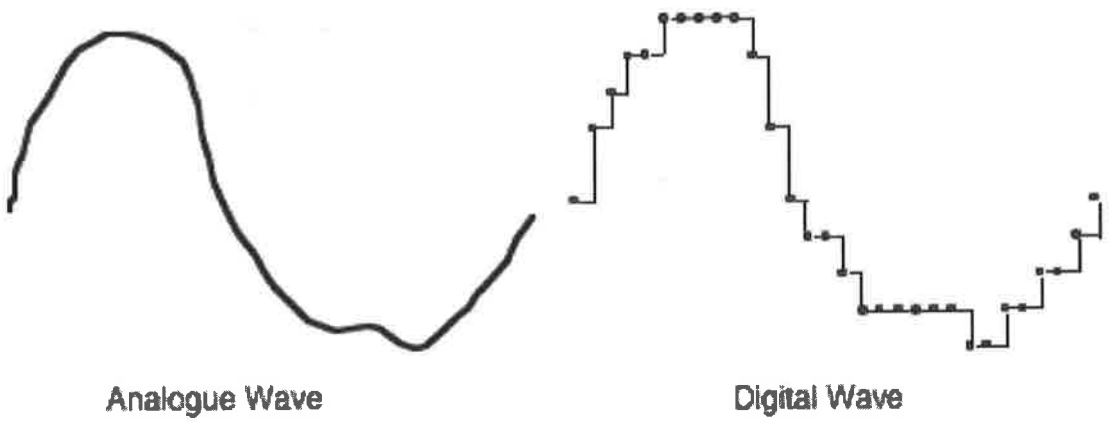


Figure 2.3. Severe quantization of a recorded waveform.

As shown in figure 2.3, the digital representation of the wave is less smooth than the original signal. Just as a square wave has a richer harmonic spectrum than a sine wave, so the more jagged digital signal will introduce some higher harmonics that were not in the original. This quantization noise is usually heard as additional high frequency hiss. The main solution to minimizing quantization is the use of higher bit rates, 16 and above, so that the number of steps in the digital values are so numerous that the rounding is insignificant.

audio files to store values as signed integers, however, it is more convenient to process audio as floating point numbers because the precision is very high and the risk of clipping audio is almost negligible. The JavaSound libraries leave the decision about the audio processing data type to the programmer, while the jMusic audio routines always work with audio data as floating point values. The floating point values are assumed to be normalized between –1.0 and 1.0 and so data read from files will be scaled within that range and, to avoid clipping when saving, all data should be normalized (see below) prior to saving.

Understanding the details of digital audio is important because even though the JavaSound and jMusic packages contains classes for reading and writing the audio file headers, when we create audio streams in Java we must be aware of the rate, resolution and formats supported by the language, the computer hardware and the file formats we are using. To a large extent, both Java and the jMusic libraries take care of many of these digital audio details, but once we move beyond the simplest programs, an understanding of these details will become essential.

Concept Quiz

1. What is the difference between sample rate and resolution?
2. What is the sample rate used by CD audio?
3. Is the bit depth of the file related to sample rate or resolution?
4. Do the jMusic `Read.audio()` and `Write.audio()` methods deal with samples as integers, bytes or floats?
5. What is the audible result of clipping?
6. Would normalizing floating point audio data prevent clipping in the saved audio file?
7. What is the audible result of aliasing in an audio file?
8. Does the sample rate or sample bit depth influence the possibility of quantization noise?
9. How many samples are required to calculate a linear interpolation?
10. In an interleaved stereo audio file how many samples are between the 1st and 3rd sample of the first channel?

Audio Dynamics

At their essence audio samples have only one parameter, loudness. Other musical attributes, such as a pitch and timbre, are a result of combinations of samples over time. Given, that loudness is the basic attribute of samples this section will focus on the way's in which this can be usefully manipulated.

```
        }
        // calulate the change in gain required
        gain = 1.0f / maxGain;
        System.out.println("MaxGain = " + maxGain + " gain = " + gain);
        // change the volume of all samples
        for (int i=0; i< data.length; i++) {
            data[i] *= gain;
        }
        Write.audio(data, "Normalized.aif");
    }
}
```

Remember, that there are negative values in the data and the loudest sample may be a negative one. This is why the `Math.abs()` method is used to find the absolute (positive equivalent) value of each sample.

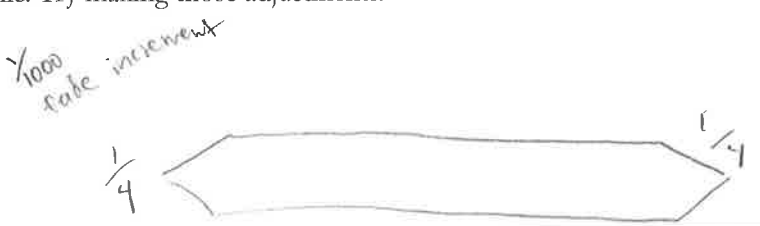
Fading

In this program each sample is changed by a different value in order to achieve a fade out effect. The first sample is at full volume, then each subsequent one is at an ever decreasing level. The ramping effect is linear, and would look like a straight line if plotted on a graph.

```
import jm.util.*;

public final class Fading {
        public static void main(String[] args){
        float[] data = Read.audio("Welcome.au");
        float fadeIncrement = 1.0f / data.length;
        for (int i=0; i< data.length; i++) {
            data[i] *= (data.length - i) * fadeIncrement;
        }
        Write.audio(data, "Fading.aif");
    }
}
```

Only small changes are required to make the audio fade in rather than fade out, or to fade over a shorter period than the whole file. Try making those adjustments.



```
        float threshold = 0.2f;
        float ratio = 2.0f;
        float gain = 2.0f;
        for (int i=0; i< data.length; i++) {
            // positive values
            if (data[i] > threshold) {
                data[i] = threshold +(data[i]-threshold) *
                     (1.0f/ratio);
            }
            // negative values
            if (data[i] < -threshold) {
                data[i] = -threshold + (data[i]+threshold) *
                     (1.0f/ratio);
            }
            // apply the gain to all samples
            data[i] *= gain;
            if (Math.abs(data[i]) > max)
            max = Math.abs(data[i]);
        }
        Write.audio(data, "Compressed.aif");
    }
}
```

Experiment with the ratio, threshold and gain variables to achieve different compression effects. Using a ratio of less than 1:1 will turn the compressor into an expander.

Math Moment

The human ear does not respond to loudness in a linear way. This means that a gain reduction of 0.5 in the `VolumeChange` class will not result in a perceptual halving on the loudness, in fact a much larger change is necessary to produce that. Perceived loudness is measured on the logarithmic decibel scale. All the variations in dynamics would need to substitute different formulae to achieve the appropriate psychoacoustic effect. For example in the compressor, rather than the simple (1.0f/ratio) ratio calculation the math should be `(float)Math.min(1.0, (Math.abs(Math.log(1.0 - 1.0/ratio) * 0.2 )));`.

Java Details

There have been a few new Java concepts introduced in this chapter. In this section we'll examine them in more detail.

```
int[] myIntArray;
```

An array with a particular size is declared as:

```
int[] myIntArray = new Array[23];
```

An array can also be declared and filled in the one operation, in this way:

```
int[] myIntArray = {2, 45, 6, 78, 32, 11};
```

We have seen an array of Strings as an argument to the `main()` method in all our examples so far. This enables the program to accept and list of words and/or numbers as data when the program runs.

Data is accessed in an array by specifying the index in the list you wish to set or retrieve inside the square brackets. Remember that the index starts at zero. For example, retrieving the fourth element from the array:

```
int age = myIntArray[3];
```

Setting the first element of the array;

```
myIntArray[0] = 44;
```

Iteration

Repeating processes is called iteration and is a common and powerful feature of programming languages, including Java. The workhorse iteration operation in Java is the for-loop. Which has the form:

```
for (initial condition; boolean result; iterative update) {
    //code to execute if boolean result is true;
}
```

An example is :

```
for (int i=0; i < data.length; i++) {
    data[i] *= (data.length - i) * fadeIncrement;
}
```

This example starts by setting an integer variable `i` to a value of zero. Each time around the loop `i` is compared to see if it is less than the length of the array named `data`. If so, then the value of `i` is incremented by one and the block of code is executed once. The Boolean (true or false) test is checked and the loop continues until the result is false, at which time the program moves on to the code after the for-loop.

3: Manipulating Digital Audio

The concept of digital audio was introduced in chapter 2, so now we will turn our attention to processes and techniques for working with digital audio. Because the digital representation of sound is a simple list of numerical sample values, all manipulations will be mathematical functions of various types on those numbers. While some of the manipulations, such as reversing a sound, require straight forward manipulations others are more complex or less obvious. We will look at a number of these processes without attempting to teach digital signal processing techniques in detail, which is the topic of other (many other) engineering texts.

Music Moment

The use of audio recordings as a compositional tool was extensively developed in the middle of the 20th century by Pierre Schaeffer and his colleagues at the GRM studios in Paris through a process they called *musique concrète*. The name was given to reflect the fact that they were making music from static recorded sounds assembled into sound constructions. At the time, recordings were made onto analogue tape and these recordings were 'processed' by cutting and splicing tape segments together into sound collages. This practice was revolutionary at the time, but today we take the capture and manipulation of recorded sound for granted, and it has become widely accepted that such manipulation can be considered an artistic practice.

Dividing and Combining Audio

The methods used in creating pieces of musique concrète by cutting and splicing magnetic tape are now much easier with digital audio technologies. In this chapter we will explore some of these techniques and see how they can be used creatively to make new music.

Mixing

Combining two audio files to hear both sounds together—mixing the files—is simply a matter of adding the sample values one by one. `CombinedSample = file1Sample + file2Sample`. In order to prevent clipping if the combine samples values are above the maximum threshold, a volume change multiplier can be used. So the final formula is; `CombinedSample = (file1Sample + file2Sample) * gainReductionAmount`.

```
        // prepare for calling mix method
        float[] mixedData;
        if (data1.length > data2.length) {
            mixedData = mix(data1, data2);
        } else {
            mixedData = mix(data2, data1);
        }

        // output the resulting file
        Write.audio(mixedData, "MixedAudio.aif");
    }

    private  float[] mix(float[] longest, float[] shortest) {
        float[] tempData = new float[longest.length];
        // combine where file length overlaps
        for (int i=0; i<shortest.length; i++) {
            tempData[i] = (longest[i] + shortest[i])  * 0.5f;
        }
        // pad out with remainder of longer file
        for (int i=shortest.length; i<longest.length; i++) {
            tempData[i] = longest[i]  * 0.5f;
        }
        return tempData;
    }
}
```

Dividing audio files

A common requirement when working with audio files is to isolate particular sections of the file. The *Chop* program creates a new file from a specified segment off the original file. The start and end points of the segment are specified in samples. It is possible to modify the program so the start and end locations are specified as time (in seconds) or beats (taking into account a tempo) locations and then performing the appropriate mathematical conversion to samples.

```
import jm.util.*;

public class Chop {
    float[] data = Read.audio("Welcome.au");

    public static void main(String[] args) {
        Chop chop =  new Chop();
```

```
        }
        return tempData;
    }
}
```

Inserting silence

Silence is very quiet audio. To make a section of an audio file contain silence the sample values simply need to be set to zero. It is important to remember that even though it cannot be heard, silent audio sections continue to occupy the same amount of memory and file space as audio we can hear. The number of samples in the file do not change, only their values. The silence program changes the samples values within a specified region of the audio file.

```
import jm.util.*;

public class Silence {
    float[] data = Read.audio("Welcome.au");

    public static void main(String[] args) {
        Silence sil =  new Silence();
        Write.audio(sil.process(5000, 20000),"SilenceAdded.aif");
    }

   private float[] process(int startLocation, int endLocation){
        int length = endLocation - startLocation;
        for (int i=0; i<length; i++) {
            data[startLocation + i] = 0.0f;
        }
        return data;
    }
}
```

Envelopes

At the borders between sounding and silent areas of a wave file (including at the beginning and end of the file) a sudden change of sample value can cause an audible click. To reduce this unwanted artifact a short fade in or out can be applied to the sounding section. The curve which outlines the overall loudness of the wave file is called the amplitude envelope of the sound. By changing the volume at the ends of the sounding segments to minimize 'clicks' we are altering the enve-

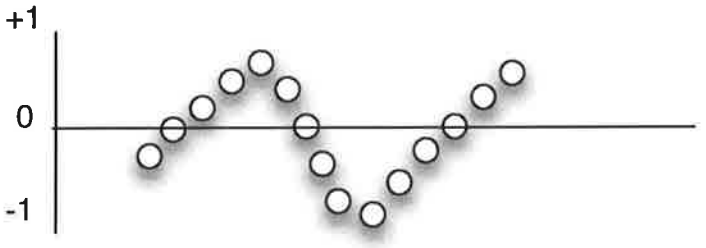


Figure 3.2. A wave section with three zero crossing points.

To search for zero crossing points in your code, simply check the value of each sample until you find one that equals (or is very close too) zero. This code fragment is an elaboration on a section of the *Loop* program that adjusts the loop end point to ensure that it falls on the first zero crossing after the specified loop end location. In practice this difference in loop end time will be so short that it will not effect the tempo of the loop to a noticeable degree. The additional code required is shown in bold in below. It creates an additional condition that keeps the for-loop processing until both the segment length is exceeded and the sample value equals zero.

```
for (int i=0; i<length && data[startLocation + i] != 0.0f; i++) {
    tempData[i] = data[startLocation + i];
}
```

Random recombination

The *Recombine* program chops several segments from a wave file and creates a new file from them. The segments are randomly selected from the wave data, then toped and tailed by changing the segment envelope with a short fade in and out. Because several segments are extracted and processed, the code that does this can be put into a `newSegment()` method which is called three times. Checks are made to the random values to make sure that the ranges for segment size and position are valid.

```
import jm.util.*;

public class Recombine {
    float[] data = Read.audio("Welcome.au");

    public static void main(String[] args) {
        Recombine recomb =  new Recombine();
        Write.audio(recomb.process(), "Recombined.aif");
        View.au("Recombined.aif");
```

```
        return tempData;
    }
}
```

Programming Practice

After typing in and running the Recombine program try changing the `process()` method and adding a for-loop within in to allow any number of very short segments to be recombined. Whenever you see the sort of repetition of code that appears in the `process()` method there is usually an opportunity to consolidate by reducing the repetitive code and adding a for-loop around the single remaining code block.

Audio Data Parameters

The two key attributes of an audio file are its sample rate and bit depth. In this section we will explore how the sound changes as these attributes vary.

Resolution

The resolution of the audio file depends upon the size, or bit depth, of the 'words' used to represent each sample. Data words are made up of a number of bytes, and each byte is made up of 8 bits – binary digits. Data words are, therefore, multiples of a byte such that a sample value can be represented by an 8 bit, 16 bit, 24 bit or 32 bit word, and so on.

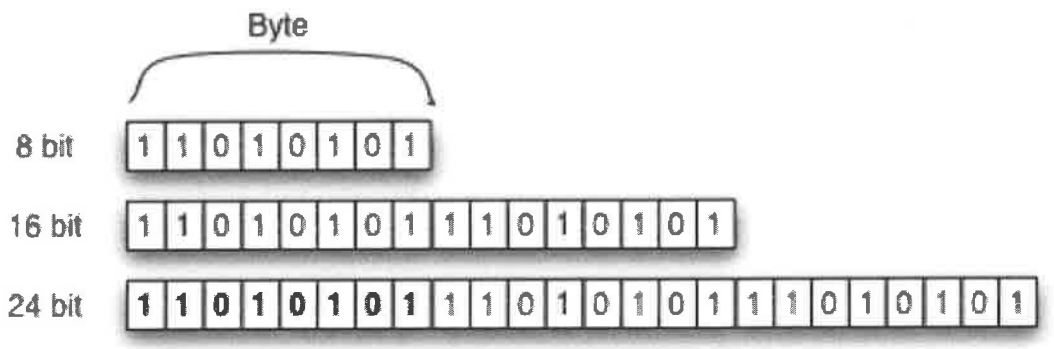


Figure 3.3. Various sample word sizes.

A 16 bit sample has greater resolution and than 8 bit sample but less than a 24 or 32 bit sample.

lowering of pitch, and the higher sample rate results in an increase in pitch. Here is the important line of code in this program.

```
Write.audio(data, "Welcome22.aif", 1, 22050, 16);
```

The arguments to the `Write.audio()` method indicate that the number of channels will be 1 (mono), the sample rate for the new file is 22050 sps and the bit depth of the file is 16 bit. Here is the complete code listing.

```
import jm.util.*;
public final class SampleRateChange {
   public static void main(String[] args) {
      float[] data = Read.audio("Welcome.au");
      Write.audio(data, "Welcome22.aif", 1, 22050, 16);
      Write.audio(data, "Welcome48.aif", 1, 48000, 16);
   }
}
```

This resampling process changes the duration of the sound as well, lower sounds are longer, while higher sounds are faster. This is quite understandable when you consider that the same number of samples are in each file, but they are being played back more or less frequently, depending on the sample rate.

Changing the number of channels

Another way of changing the pitch and duration of the waveform is to alter the number of samples. In this example we will divide the samples into two channels, thus making it a stereo file. This process will put every second sample in the first channel and the others into the second. Picking alternate channels for each sample halves the playback time, because both channels play back simultaneously. Because the sound file is being stepped through twice as fast the pitch will also increase, even though the sample rate of the file is the same. Figure 3.4 shows how the sample data is organized in mono and stereo files.

```
        double newPitch = 261.0;
        double resamplingRatio = newPitch/originalPitch;
        float[] newData = new float[(int)(data.length/resamplingRatio)+1];

        int counter = 0;
        for (double i=0.0; i<data.length; i += resamplingRatio) {
           newData[counter++] = data[(int)i];
        }
        Write.audio(newData, "ShiftedPitch.aif");
    }
}
```

Let's examine the code step by step.

First, we make an assumption about the original pitch, or fundamental frequency, of the sound as a reference point. In this case we arbitrarily define the original pitch as 440.0 cps, A4. The target frequency is specified, `newPitch = 261.0`, which is approximately C4. The resampling ratio is the difference between these two frequencies. It will be used to determine how regularly to interrogate the original data and calculate a new sample value. If this ratio is 1.0 each new sample will correspond with an existing one and the result will be identical to the original. If the resampling ratio is less than 1.0, then the new data will have more sample than the original data, and the resulting pitch will be lower. If the resampling ratio is above 1.0 then the pitch shift is upward, and the new data will skip over some of the samples in the original data.

As shown in figure 3.5. The sample ratio increments are `double` values because `int` values would not provide enough granularity to produce all the pitch values we might want. As a result, the new sample positions will not fall evenly on the existing values, so in-between values will need to be created. In the *SimplePitchShift* program new in-between samples are created by duplicating a nearby value.

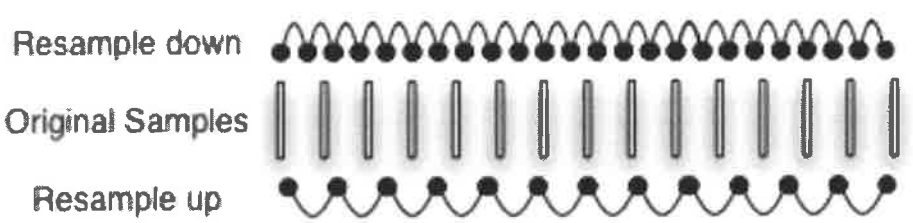


Figure 3.5. Sample data points during interpolation.

Pitch shift through interpolation

The use of interpolation results in a smoother amplitude curve and, therefore, less quantization noise than in the previous simple pitch shifting algorithm. Weather or not this is a desirable result is a musical decision. The most common form of interpolation is linear interpolation, where the value is calculated as a point along the straight line drawn between those existing sample values.

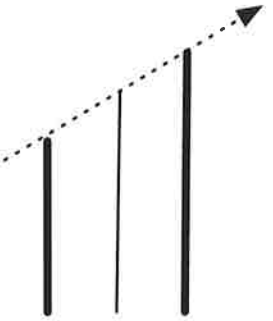


Figure 3.6. Linear interpolation used to calculate the centre sample value.

The bold lines of code in the *Interpolation* program below perform the linear interpolation between the two samples closest to the new sample location. In other respects this example is the same as the *SimplePitchShift* program.

```
import jm.util.*;

public final class Interpolation {
    public static void main(String[] args){
        float[] data = Read.audio("Welcome.au");
        double originalPitch = 440.0;
        double newPitch = 261.0;
        double resamplingRatio = newPitch/originalPitch;
       float[] newData = new float[(int)(data.length/resamplingRatio+1];

        int counter = 0;
        for (double i=0.0; i<data.length - 1; i += resamplingRatio) {

            float lowerSample = data[(int)i];
            float upperSample = data[(int)i + 1];
            float difference = upperSample - lowerSample;
            float newSampleValue = lowerSample + difference *
                (float)(i%1.0);
            newData[counter++] = newSampleValue;
```

when they are used. Some of the jMusic classes work this way too, including the `Read`, `Write`, `Play`, `View` and `Mod` classes. This use of static methods is not considered a very elegant design from the perspective of object-oriented programming, but it is a convenient way to access these functionalities. This table summarizes a number of the most commonly used methods in the math class. Consult the Java documentation for a complete list.

- `Math.abs(x)` Int, long, float, double Absolute (positive) value
- `Math.cos(x)` Double Cosine value
- `Math.sin(x)` Double Sine value
- `Math.log(x)` Double Natural logarithm of x
- `Math.max(x, y)` Int, long, float, double Maximum of the two
- `Math.min(x, y)` Int, long, float, double Minimum of the two
- `Math.pow(x, y)` Double X to the power y
- `Math.random()` None Random value, 0.0 to 1.0
- `Math.round(x)` Float, double Closest whole number to x
- `Math.sqrt(x)` Double Square root of x

A method's return type

Method declarations must always specify a return type. Return types can be any data type. When the method does not return (produce) any value the type `void` is specified. The return type is positioned just prior to the method name in the declaration as in this example where the `getCounter()` method returns an `int`.

```
int getCounter() {}
private int getCounter() {}
```

Here are some more examples;

```
float getAmplitude() {
    ...
}
private boolean makeDecision() {
    ...
}
```

4: Music as Digital Data

Musical ideas are abstract, and therefore in order to communicate them to another person or to a computer we need to represent these ideas in some way. Languages are formal systems for representing our ideas and Java is a language designed to help us communicate with the computer. Music notation systems are languages (more correctly symbol systems) for representing music. The most well established Western music representation system is Common Practice Notation (CPN), the system of dots and squiggles on five-line staves. Computer music systems usually require a much more accurate and flexible system than CPN because the computer cannot be relied upon to interpret the notation with very much cultural understanding. jMusic's music representation scheme is based on CPN to make it easy to learn, but can be much more elaborate and detailed than CPN when this is required. In this chapter, we introduce Java objects and the way in which they are structured for music making. In particular we look at some of the jMusic music data objects and how they represent musical information.

Musical Components and Structure

Music is made up of structured sounds and silence. But music, as sound or ideas, is not a concrete thing that we can see, grasp or capture. In order to do so we create musical representations. Typically these representations stand for particular musical events and describe the characteristics of these events. But what are music representations for? They serve a number of functions. First, they make music concrete and stable so that it can be stored and manipulated. We can capture music as scores, data structures and files and we can process and modify the representation. Secondly, it allows music to be communicated in ways other than aurally. We can transfer files to other software packages, send the musical representations over the internet, and print them out using music publishing software. Thirdly, a representation is a conception of music, a way of understanding it, a way to think about it. The way we represent music says a lot about what we conceive music to be, and it influences the types of musical transformations that will be evident and available to us. Importantly, the music representation is *not* the music, and so the way a representation is interpreted by software, hardware or humans can vary the outcome significantly by amplifying any distortion present in the representation system.

Over time there have been many schemes for representing music, and in Western cultures the most prevalent has been the use of the notated score. The data structure in jMusic takes this tradition as

- `Pitch` - integer (0 – 127, lowest to highest frequency of the note in chromatic steps)
- `RhythmValue` - double (>=0.0, shortest to longest length of the note in beats)
- `Dynamic` - integer (0 – 127, softest to loudest intensity of the note)

Creating a note is done using the `new` keyword, and passing the desired attribute values. The most common `Note` constructor takes arguments for pitch, rhythm, and dynamic. This line of code creates a loud, middle C, quarter note that is assigned to the variable n.

```
Note n = new Note(60, 1.0. 85);
```

There are helpful constant values for these attributes to make the code easier to read and write. The line above can be rewritten using constants inherited from the JMC interface. A full list of JMC constants can be found in Appendix A.

```
Note n = new Note(C4, QUARTER_NOTE, FORTE);
```

or with constants that are abbreviations,

```
Note n = new Note(C4, QN, F);
```

A special type of `Note` object is a `Rest`. A rest is a silent note, used to create space in a phrase. A `Rest` has a `rhythmValue`, but no `pitch` or `dynamic` (of course).

```
Rest r = new Rest(QN);
```

There is another way to create a rest, as a special type of `Note`. This is useful in algorithms where notes and rests are being created together.

```
Note r = new Note(REST, QN);
```

This code example creates a phrase from a few notes and rests and plays it back.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;

public class Beet implements JMC {
    public static void main(String[] args) {
        Phrase phr = new Phrase(0.0);
        for (int i=0; i<3; i++) {
            Note e = new Note(E4, EIGHTH_NOTE_TRIPLET);
            phr.addNote(e);
        }
```

beats, so that note lengths are relative to each other, and so that timing and playback length change as the tempo of the score is altered. We will discuss setting and changing tempi later, when describing `Score` objects.

Creating a new phrase is usually done using just one argument, the start time. In this case the phrase is set to start at beat 0.0, the beginning of the music. This code creates a new `Phrase` object assigned to the variable `phr`.

```
Phrase phr = new Phrase(0.0);
```

Once a phrase has been created, notes can be added to it using the `addNote()` method.

```
phr.addNote(n);
```

Importantly, the list of notes in a phrase is monophonic, that is, notes are stored one after another in the order that they are added to the phrase. This seems obvious for melodies and riffs, but it means that several phrases are required to represent a chord or a drum kit rhythm where more than one note may sound together. Monophonic phrases are not a limitation, but a design choice of jMusic, which aims to keep simple things easy to do while ensuring that complex things are possible. Chords and polyphonic textures are adequately catered for in other ways using multiple phrases or the `CPhrase` class, as we will see later.

Here is an extension of the *Bing* program that creates a phrase, adds three notes to it, then saves the phrase as a MIDI file and plays it back.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.Play;
import jm.util.Write;

public final class BingPhrase implements JMC {
    public static void main(String[] args) {
        Phrase phr = new Phrase(0.0);
        Note n = new Note(G4, EIGHTH_NOTE);
        phr.addNote(n);
        Note n2 = new Note(D4, QUARTER_NOTE);
        phr.addNote(n2);
        Note n3 = new Note(E4, HALF_NOTE);
        phr.addNote(n3);
        Write.midi(phr, "BingPhrase.mid");
        Play.midi(phr);
    }
```

```
Part part1 = new Part("Flute", 73, 0);
```

There are constants in the JMC for the General MIDI set of instruments. A list of the General MIDI instrument numbers and constants can be found in Appendix A. Notice that by convention in Java, constants are written using upper case letters.

```
Part part1 = new Part("Flute", FLUTE, 0);
```

Adding a phrase to a part is done using the part's `addPhrase()` method.

```
part1.addPhrase(phr);
```

To recap about the music data structure so far; we have `Notes` that are the elemental music objects, many notes can be stored in series within a `Phrase` which has a start location in the music. A `Phrase` can be added to a `Part` which can have an instrument (sound) associated with it. Each layer of this structure adds additional features to the musical whole.

The MIDI Standard

It is likely that you will want to save your music as a file for storage, playback, or use in another application. The most common file format for symbolic digital music is the standard Musical Instrument Digital Interface (MIDI) file. Files saved in this format have, by convention, a *.mid* suffix appended to their name. Some attributes of the `Note` class follow the conventions of the MIDI specification. These include the pitch values which are consecutively number semitones from 0 to 127, with middle C being 60, the C sharp above it is 61, then D is 62 and so on. The jMusic dynamic values that range from 0 to 127, softest to loudest, also mirror the MIDI specification. There has been much debate about the limitations imposed by MIDI (Moore 1991) however the jMusic specification goes beyond the MIDI specification in may areas, as will become clear later in this book. The following of these MIDI conventions assists in the translation between jMusic and MIDI data for playback, makes note values easy to specify for those familiar with MIDI, and enables fluent translation to and from MIDI files.

Here is the *Bing* program, from chapter 1, modified to save its output as a MIDI file. This file can be read by other music software. Compare the *Bing* and *BingFile* programs and notice that two lines are different.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.Write;
```

Concept Quiz

1. What are the three functions that music representations serve?
2. What does it mean than the music representation is not the music?
3. Name the four levels of the jMusic's data structure.
4. Why do many of the code examples implement the JMC?
5. What is MIDI an acronym for?
6. jMusic `Phrases` have a `startTime` field, what does its value indicate?
7. Is a jMusic `Phrase` monophonic or polyphonic?
8. What is the method used to add a `Phrase` to a `Part`?

Playing many parts

Below is an example program that elaborates on the *Bing* class to play several notes in two different parts.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;

public final class BingPart implements JMC{
    public static void main(String[] args){
        Part myPart = new Part(TRUMPET, 0);
        Phrase topPhrase = new Phrase(2.0);
        Note n = new Note(G4, HALF_NOTE);
        topPhrase.addNote(n);
        myPart.addPhrase(topPhrase);
        Phrase midPhrase = new Phrase(1.0);
        Note n2 = new Note(E4, DOTTED_HALF_NOTE);
        midPhrase.addNote(n2);
        myPart.addPhrase(midPhrase);
        Phrase bottomPhrase = new Phrase(0.0);
        Note n3 = new Note(C4, WHOLE_NOTE);
        bottomPhrase.addNote(n3);
        myPart.addPhrase(bottomPhrase);

        Write.midi(myPart, "BingPart.mid");
        Play.midi(myPart);
    }
}
```

```
        Phrase phrase1 = new Phrase(0.0);
        for (int i=0; i<16; i++) {
            for (int j=0; j<12; j++) {
                Note n = new Note(C4+j*2, SIXTEENTH_NOTE);
                phrase1.addNote(n);
            }
        }
        Phrase phrase2 = phrase1.copy();
        phrase1.setTempo(130.0);
        phrase2.setTempo(131.0);

        Part guitarPart = new Part(GUITAR, 0);
        guitarPart.addPhrase(phrase1);
        guitarPart.addPhrase(phrase2);

        Play.midi(guitarPart);
    }
}
```

Scores

If we consider music of the complexity of a jazz trio, with its multiple parts, these parts need to be collected together into a score. The `Score` class represents the top level of the jMusic data structure and contains (you guessed it) a `vector` of parts. A score object also has other attributes including a title and a tempo.

The tempo, or speed, is described in beats per minute (BPM) and is set to 60bpm by default, which is quite a slow tempo but has the advantage that each beat equals one second of time, which is useful for music that is time-based rather than beat-based.

A common way to create a new score is with the constructor that takes a title and tempo as the arguments.

```
Score myMusic = new Score("Opus 1", 135.0);
```

Insert parts into a score with the `addPart()` method.

```
MyMusic.addPart(part1);
```

```
import jm.music.tools.*;
import jm.util.*;

public final class Phasing implements JMC{
    public static void main(String[] args){
        Phrase melody = new Phrase(0.0);
        CPhrase chords = new CPhrase(0.0);
        int[] melPitches = {C5,D5,E5,A5,G5,D6,C6};
        double[] melRhythm = {DQN,EN,QN,QN,QN,EN,DQN};
        melody.addNoteList(melPitches, melRhythm);

        int[] notePitches1 = {C3, E3, G3};
        int[] notePitches2 = {D3, F3, A2};
        int[] notePitches3 = {B2, D3, G3};

        chords.addChord(notePitches1, HALF_NOTE);
        chords.addChord(notePitches2, HALF_NOTE);
        chords.addChord(notePitches3, HALF_NOTE);

        Part kalimba = new Part("Thumb Piano", KALIMBA, 0);
        kalimba.addPhrase(melody);
        Part rhodes = new Part("Electric Piano", EPIANO,1);
        rhodes.addCPhrase(chords);

        Mod.repeat(kalimba, 6);
        Mod.repeat(rhodes, 7);

        Score comp = new Score("Phasing Example", 90);
        comp.addPart(kalimba);
        comp.addPart(rhodes);

        Play.midi(comp);
    }
}
```

Another way of achieving phasing is to independently vary the playback speed of two identical looped phrases. This is a subtle effect and, if the speed variation is only minor, adds timbral variety through phase cancellation to the more obvious rhythmic evolution. This technique, used by Steve Reich in many of his earlier works, is the basis for the next example program.

```
        score.addPart(part1);
        Mod.repeat(phras41);
        part2.addPhrase(phrase2);
        score.addPart(part2);
        part1.setTempo(120.0);
        part2.setTempo(120.5);

        Play.midi(score);
    }
}
```

Musical data structure

You should have a good overview of the jMusic data structure at this stage. Notes describe musical events, notes are arranged in as a list in phrases. Phrases have a start time which positions them within a part. Parts can contain many phrases and also have an instrument assigned to them. All notes within a part are played by that instrument. The parts are added to a score which also sets the tempo of music within it.

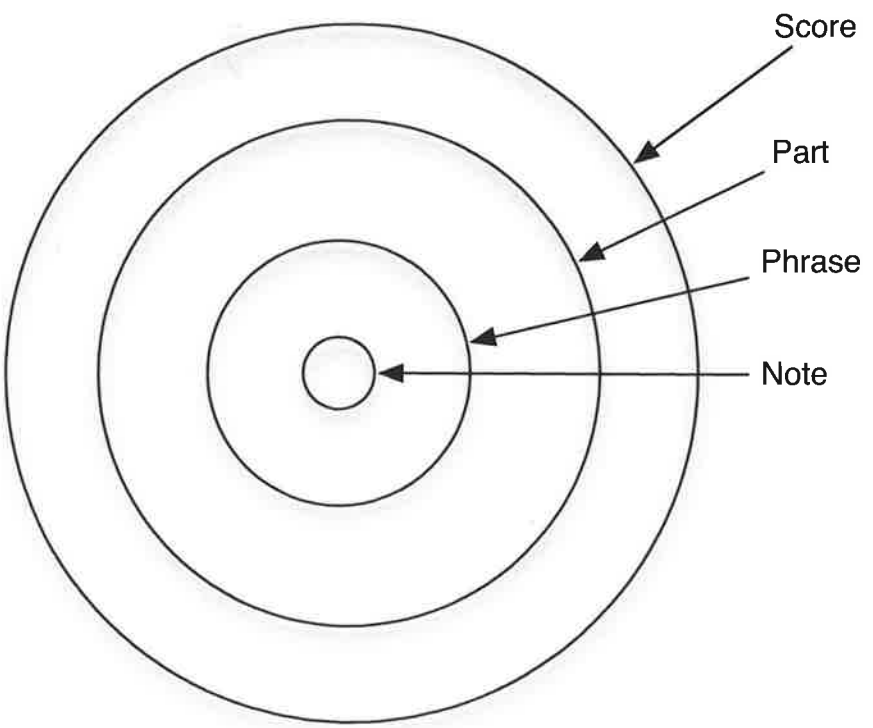


Figure 4.2. The jMusic Data Structure.

```
public class MultiPhraseDrums implements JMC {
    // declare instance variables
    Score music = new Score(125.0);
    Part drums = new Part(25, 9);
    Phrase kickPhrase = new Phrase(1.0);
    Phrase snarePhrase = new Phrase(1.0);
    Phrase hatsPhrase = new Phrase(1.0);

    public static void main(String[] args) {
        MultiPhraseDrums kit = new MultiPhraseDrums();
        kit.createKickPhrase();
        kit.createSnarePhrase();
        kit.createHatsPhrase();
        kit.output();
    }

    // the constructor
    public MultiPhraseDrums() {
        drums.addPhrase(kickPhrase);
        drums.addPhrase(snarePhrase);
        drums.addPhrase(hatsPhrase);
        music.addPart(drums);
    }

    private void createKickPhrase() {
        for (int i=0; i<8; i++) {
            kickPhrase.addNote(new Note(C2, QUARTER_NOTE));
            kickPhrase.addRest(new Rest(QUARTER_NOTE));
        }
    }

    private void createSnarePhrase() {
        for (int i=0; i<8; i++) {
            snarePhrase.addRest(new Rest(QUARTER_NOTE));
            snarePhrase.addNote(new Note(D2, QUARTER_NOTE));
        }
    }

    private void createHatsPhrase() {
        for (int i=0; i<16; i++) {
```

tory. The JMC class that is imported in all the examples in this book is located within the `jm` directory - the top level directory of the jMusic library. External libraries, such as jMusic, always need to be imported. In this example the jMusic constants (JMC) interface is imported.

```
import jm.JMC;
```

Constructors

A constructor is a special method that is run when a new instance of a class is created. Up to this point only one example program required the writing of a constructor. Whenever the `new` keyword is invoked to create an object, that class's constructor is run as the object is instantiated. A constructor is a method with a couple of distinctive syntactic features. Constructors must have the *same name* as the class they are in and, when declared, they *don't* require a return type. Here is an empty constructor added to the *MakeMusic* class.

```
// class declaration
public class MakeMusic {
    // constructor
    public MakeMusic() {
        ...
    }
}
```

The constructor is run only once in the life of the object, when it is created. The brackets "( )" following the constructor name can, as with methods, contain information about data that may be passed to the constructor when it is called. Such data are called *arguments*. In the example above, the constructor does not take any arguments. Even if no arguments are required (there is no data to be passed), the empty brackets are still required.

Instance variables

Variables, as we've seen, can be used to hold objects or primitive data types. For example the variable `n` may stand for a `Note` object, and the variable `myHeight` may contain the number 170. The variable `n` in the code below is an instance variable standing for a `Note`.

```
public class MakeMusic implements JMC{
    Note n = new Note(C4, QUARTER_NOTE);

    public void playMusic() {
        Play.midi(n);
    }
```

```
        ... you can do something here with A or B

        for (int C; ; ) {// local variables within the for-loop
            int D;
            ... you can do something here with A or B or C or D
        }
        ... you can do something here with A or B
    }
    ... you can do something here with A
}
```

IT is important to remember that if a variable is declared within a sub block of code, such as an if statement, it is only visible within that block of code. Remember, that a block of code is often bounded by open { and close } curly brackets but that in the case of the for-loop the `for` statement and conditions are visible within the code bounded by curly brackets that follows, because they are part of the same expression. In this incomplete class below the variable `scaleLength` and `i` are local, however, because of where they are declared, `scaleLength` is visible anywhere within the method but i is only visible within the for-loop block. The phrase `myPhrase` is an instance variable and is visible anywhere in the class.

```
public class BeatMover {
    Phrase myPhrase =  new Phrase();

    private int chromaticScale () {
        int scaleLength = (int)(Math.random() * 24);
        for (int i=0; i < scaleLength ; i++) {
            myPhrase.addNote(new Note(60 + i, 1.0));
        }
        return scaleLength;
    }
    more code...
}
```

Concept Quiz

1. How many jMusic Notes can be in a Phrase?
2. Which jMusic class specifies the instrument for playback of the notes?
3. What do we mean by the term musical phasing?
4. Why are `import` statements required?
5. What is the connection between the new keyword and a constructor method?
6. What is a variable?

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;

public class TheGrid1 implements JMC {
    private Phrase hats;
    private Part drums;

    public static void main(String[] args) {
        TheGrid1 grid = new TheGrid1();
      grid.compose();
    }

    public TheGrid1() {
        hats = new Phrase(0.0);
        drums = new Part("Drums", 25, 9);
        drums.addPhrase(hats);
    }

    public void compose() {
        int[] hatsHits = {FS2, REST, FS2, REST, FS2, FS2, FS2,
              REST, FS2, REST, FS2, REST, FS2, FS2, FS2, FS2};
        hats.addNoteList(hatsHits, SIXTEENTH_NOTE);
      Write.midi(drums, "TheGrid1.mid");
        Play.midi(drums);
    }
}
```

Programming Practice

The classes start execution at the `main()` method, which calls the constructor to create a new instance of the class which, in turn, calls other methods as necessary. Try compiling, running then editing this code. You can change the rhythm by changing the order of notes and rests in the hatsHits array. Run the program by typing `java TheGrid1` and press return.

There are three methods in this class, `main()`, `TheGrid1()` (the constructor), and `compose()`. In addition there are two instance variables, `hats` and `drums` that are used by several of the methods. The main method creates a new instance of the class which calls the constructor. The constructor instantiates (creates and sets up) an empty phrase named hats and an empty part

Music Moment

A number of ways of generating notes have been used in the examples so far in this book. They include adding notes individually, at random, selecting from an array of note attribute values, and using the `addNotelist()` method. Here is an example of each technique.

Individual Notes

```
Note n = new Note(A4, QUARTER_NOTE);
phrase.addNote(n);
```

Random Notes

```
for (int i=0; i< 8 ; i++) {
    Note n = new Note((int)(Math.random() * 12 +60), QUARTER_NOTE);
    phrase.addNote(n);
}
```

Selecting from a pitch set

```
int[] pitches = {C4, E4, G4, D4, A3, B3, C4};
for (int i=0; i< 8 ; i++) {
    Note n = new Note(pitches[(int)(Math.random() *pitches.length)],
          QUARTER_NOTE);
    Phrase.addNote(n);
}
```

Adding notes using an attribute list

```
int[] pitches = {C4, E4, G4, D4, A3, B3, C4};
phrase.addNotelist(pitches, QUARTER_NOTE);
```

5: Playing with Polyphony

Creating music of significant complexity involves coordinating several musical lines at one time. For example, coordinating a melody, a bass part, and a percussive accompaniment. Music that includes several independent strands at one time is said to be polyphonic. Typical examples include multi-part vocal chorales or string quartets. However, such multi-part coordination is required by music of almost any style. The jMusic part and score classes are provided to assist with this structuring.

Managing Musical Parts

Many experiments have been done by psychologists to determine how we hear musical lines. It should be noted that the tendencies in our imaginative perception are not simply choices about how things appear to us, but are the result of how our brain attempts to group events and always seeks to find patterns in what we hear. In musical terms the patterns that appear in musical lines are the basis for the texture and the form of music. These structural elements are designed by the composer and then described in a medium, for example in Java code. This compositional process requires an imagining of the musical structures and then the describing of them. Interestingly this is the reverse of the listeners experience, where they hear the music unfold over time, then retrospectively construct the form from their memory of the passing events, or more particularly from the lines, shapes and textures they perceived as the events unfolded.

There is a connection between music's structure and its meaning and enjoyment. Understanding some of the psychology of perception that underpins these sound gesture metaphors assists the composer to better create a desired musical experience. The roots of psychological concerns with how people perceived patterns in a holistic way can be traced back to the Gestalt movement that originated in Europe in the early decades of the 20th century. Their experiments were particularly focused on the visual perception of movement, but regular correspondence to audible perception of music were made. The Gestalt psychologies examined a range of phenomenal qualities and derived a series of rules, or laws about bout ways in which humans organize their experiences. These included the law of proximity, where objects that are closer (in space, time, pitch, and so on) are grouped together; the law of symmetry, where things that create a trend or pattern are perceived as a closed region, and; the law of good continuation, where foreground and background are differentiated by degrees of interruption, change or inconsistency. The Gestalt psychologists also tended

seems then that in our most basic apprehension of music there lies a complex system of metaphor, which is a true description of no material fact" (Scruton 1983:106).

This metaphorical aspect of musical structures goes beyond the gestural nature of musical shapes in time and space to include the ways in which these combine to indicate musical styles and to trigger emotional states. Particular musical gestures can be characteristic of particular musical styles, or even of individual composers. David Cope called such frequently found gestures stylistic signatures (Cope 2001). Musical metaphors are even more direct when depicting emotional states, for example the slow tempo and wide vibrato that imitate the despondent movement and quivering lip of a very sad person, or the violent staccato bowing of Bernard Herman's score for the stabbing featured in the 'shower' scene from Alfred Hitchcock's film *Psycho.*

There are many types of musical form and structure that are commonly used and the jMusic data structure has been designed to allow for them. In this chapter we will examine one of the oldest musical forms, the cannon.

Making a cannon

A musical cannon, also called a round, is a musical structure where one melody is played against a delayed version of itself. The same phrase is played by several parts whose entries and exits are staggered. We will use as an example the well known round *Row, Row, Row Your Boat.* For a more elaborate example of a cannon you can listen to the first movement of Béla Bartók's *Sonata for Two Pianos and Percussion.*

There are two ways in which to write polyphonic music in the jMusic data structure, and these reflect compositional practices in acoustic music. The first is to allocate multiple voices (phrases) to a single part. This is analogous to a harpsichord fugue where a composer may write two or more simultaneous parts for each hand to play. The second is to allocate a single phrase to each of two or more parts, as you might for a woodwind quintet where each voice is played by a different instrument. Obviously, a combination of both techniques can be used.

For now, we will write our cannon for multiple parts each containing a single phrase. The *RowYourBoat* program is a somewhat longer program so we will deal with it one section at a time. Here is the complete code.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;
import jm.music.tools.*;
```

```
        phrase2.setStartTime(4.0);
        phrase3 = phrase1.copy();
        phrase3.setStartTime(8.0);

        Mod.transpose(phrase1, 12);
        Mod.transpose(phrase3, -12);
    }

    private void playAndSave() {
        Write.midi(score, "Rowboat.mid");
        Play.midi(score);
    }
}
```

In essence, the steps for creating the cannon are; 1) create a melodic phrase, 2) make copies of that phrase, 3) change the start times so each phrase is offset in time, and 4) add each phrase to parts with different instruments.

Java Moment

When typing up a program you will inevitably make some mistakes, that produce errors when compiled. Interpreting Java compiler error messages is an important skill in working productively in the language. The error messages are the way that the compiler communicates with you about things it does not expect in the code. The errors are not always easy to interpret and, occasionally, they are misleading, but they are the key to assist you to debug your code. Here's an example:

```
AudioPlayer.java:220: cannot resolve symbol
symbol  : class Getopt
location: class AudioPlayer
Getopt  g = new Getopt("AudioPlayer", args, "hlufM:e:i:E:S:D");
^
```

The error messages, like the one above, start by indicating the class and line in which the error occurs. Then the error message, in this case 'cannot resolve symbol.' This message indicates that the class `Getopt` is not known to the compiler. Possibly it needs to be imported at the start of the code. Lastly, the offending line is displayed and a caret (^) points to the section in the line where the error was detected. All in all, this is pretty helpful, especially when compared to error messages in other languages.

Copies of the first phrase and made using the `Phrase.copy()` method and assigned to `phrase2` and `phrase3`.

```
phrase2 = phrase1.copy();
phrase3 = phrase1.copy();
```

The `copy()` method returns a duplicate of the Phrase object which we assign to the new phrase named `phrase2`. At this stage phrase 2 and 3 are identical to phrase 1, which includes that they have the same start time. However, a cannon needs each phrase to have a different start time. This can be achieved with the `Phrase.setStartTime(double newStartTime)` method which accepts an argument indicating the new start time. The start time is in beats (and fractions of a beat) and remember that the first beat of the music is zero (0.0) not one. This follows the general computer programming convention where counting begins at zero, so if you are not used to it you soon will be.

```
phrase2.setStartTime(4.0);
phrase3.setStartTime(8.0);
```

This code sets `phrase2` to start at the second measure (bar) and `phrase3` to start at the third measure (bar).

Transposing

In order that we can hear the parts a little more clearly we are going to set them to different octave ranges by transposing them. Transposition is a pitch shift up or down. A call to the `Mod.transpose(Phrase phrase, int pitchShift)` method will do this.

```
Mod.transpose(phrase1, 12);
Mod.transpose(phrase3, -12);
```

`Phrase1` is transposed up one octave (twelve semitones) and `phrase3` transposed down one octave

Structuring the music data

At this stage the score, parts and phrases exist independently and are not related to each other. This disconnected state is shown in figure 5.1.

```
score.addPart(flute);
score.addPart(trumpet);
score.addPart(clarinet);
```

Saving and playing the cannon

Finally, the score is written to a MIDI file and played back using static methods in the `Write` and `Play` classes.

```
Write.midi(score, "Rowboat.mid");
Play.midi(score);
```

Programming Practice

Type up the *RowYourBoat* program, compile and run. To make sure you understand the flow of control between methods, print out the source code and draw arrows on the page where one method calls another. Extend the program by adding a fourth part that starts one bar after the third.

The jMusic Mod Class

The `Mod` class contains many methods for modifying, or varying, phrases, parts and scores. The `Mod` class is in the `jm.music.tools` package, and you need to import this package in order to access these methods.

```
import jm.music.tools.*;
```

The `Mod` class uses static methods, so you can call the methods directly from the class and don't need to instantiate a new `Mod` object. It is used in a similar way to the `Math` class, such as `x = Math.random();`. However, somewhat unusually for Java the `Mod` methods are declared as void and do not return any value but modify the data passed to them. As an example the `Mod` class palindrome method is called with the line, `Mod.palindrome(phrase);`. While the arguments for each method vary slightly, the general structure of calls to `Mod` methods are:

```
Mod.methodName(data, [argument1], [argument2], ...);
```

Phrase manipulations are widely used in western composition, and this is reflected in the jMusic `Mod` class having more methods than other data classes. In most cases the method names are self explanatory, but that will not always be the case. Here are some of the `Mod` methods that apply to a `Phrase`:

`repeat();` - play the phrase a number of times through

Most of the method names in the Mod class are *overloaded*. This is programming term meaning that several methods have the same name but different arguments. For example the Mod class contains several append methods.

```
Mod.append(note1, note2);
Mod.append(part1, part2);
Mod.append(score1, score2);
```

This framework can be used with all the methods of the Mod class. The full list of Mod methods, including comments about their use and specifics about their parameters, are listed in appendix B.

Modifications effect existing jMusic objects. So remember to create a Phrase, CPhrase, Part, Score before modifying it.

```
Phrase phr = new Phrase();
        . . . add notes . . .
Mod.retrograde(phr); // play backwards

Phrase phr = new Phrase();
        . . . add notes . . .
Mod.repeat(phr, 3); // repeat three times
```

Concept Quiz

1. What do we call music that includes several concurrent lines?
2. What does our brain attempt to find in music we hear?
3. What does the copy() method do to a phrase?
4. What effect does changing a phrases start time have?
5. Name three of the methods in the jMusic Mod class.
6. What does the void return type indicate about a method?

Arpeggiator

Arpeggiators have been used in electronic music for many decades. An arpeggiator creates a phrase from a set of notes (a chord) by playing them in a series one after another. In essence they generate a musical order from a group of pitches; providing note-level structure. An arpeggiation is often associated with note sets that are chords, for example C, E, G, and B flat (a C7 chord). Traditionally the arpeggiated phrase is repeated many times to provide both a rhythmic and harmonic layer in the music. A variety of arpeggiation patterns are common; these patterns include playing them in ascending pitch order, in descending order, up then down, or in a random order. Here is a simple Java arpeggiator program called *Arp1*.

Java Details

In this chapter, and previously, a number of Java features have been introduced, and some have been used without much explanation. This section elaborates on a number of these Java features.

JavaSound playback

The playback of jMusic data types with the `Play.midi()` method, uses the JavaSound libraries that have been a part of the standard libraries in Java since version 1.3. When Java is installed on your computer a library of instrument sounds (a soundbank) conforming to the General MIDI instrument standard is included. It is this sample soundbank that is being used for playback. Importantly this soundbank is independent of any computer sound card or external synthesizer that you may have. This provides convenient playback, but with little control over the quality of sound. There is an option to download and install alternate soundbanks from Sun, at http://java.sun.com. Later in this book we will see how you can use your own samples and synthesized instruments to play, and render, music. This will provide much greater control over how the music sounds.

When the `Play.midi()` method is called the jMusic data is converted into a MIDI format behind the scenes before playback. This explains why, after running the program, there may be a short wait before playback begins.

Keywords

Identifying which words in the code are Java keywords, which are jMusic keywords, and which are variable names is important to understanding the code. If you use a text editor with syntax highlighting, then many of the Java keywords will be in a different color to other text. Java keywords include `public`, `class`, `int`, `double`, `static` and so on.

Keywords in java have special meaning and are reserved so you are not allowed to use them for other purposes. This means you can't create your own variable called 'class', for example. The jMusic libraries define various another words to have special meaning, including `Note`, `Phrase`, `Mod`, and `Play` which are class names, and `C4`, `FORTE`, `QUARTER_NOTE`, and `PIANO` which are constants. These words are not reserved, you can redefine their meaning, but when working with jMusic we suggest you don't. Some text editors allow you to add your own words for syntax highlighting, and it is useful with these to make a list of the common jMusic class names, method names, and constants so they are easily identified in the code.

```
int val = addEmUp(12, 4);
```

When passing primitive types as arguments, such as numbers and `Boolean` values, the arguments become copies of the original variables. Therefore the variable `x` within the `addEmUp()` method is distinct from the number that was passed. Consider this code fragment.

```
public int muckEmUp(int x, int y) {
    x = x * 2;
    y = y + 5;
    return x + y;
}

int a = 12;
int b = 4;
int val = muckEmUp(a, b);
```

The `muckEmUp()` method changes the value of `x` and `y` but the values of `a` and `b` are not effected. This is because primitive types in Java are passed by copy. This means that the value is cloned into two distinct values before one copy is passed to the method.

All Java objects (including arrays and strings) that are passed as arguments to methods are passed by reference. This means that the variable inside the method that refers to the argument is a reference to the original object passed to it. So any changes made to the argument effect the original object passed. Consider this example.

```
public void effectEm (Note n, Phrase phr) {
    n.setPitch(34);
    phr.addNote(n);
}

Note myNote = new Note(60, 1.5);
Phrase myPhrase = new Phrase(0.0);
effectEm(myNote, myPhrase);
```

The note `myNote` and phrase `myPhrase` are declared outside the `effectEm()` method and passed to it as arguments. Because they are non-primitive types, `n` and `phr` become references to `myNote` and `myPhrase` respectively. This means that both the name `n` and the name `myNote` point to the same `Note` object as depicted in figure 5.2. Therefore any changes made to `n` is reflected as a change in `myNote`, similarly with `phr` and `myPhrase`.

Accessor methods

A safe and convenient way to access the variables within a class from outside of it, is to provide methods that get (retrieve) and set (assign) its value. For example, in the `Note` class there are methods for accessing the note's pitch variable, `Note.getPitch()` and `Note.setPitch(int newValue)`.

Get and set methods are required when an instance variable is declared private because, as a result, access from outside the class is prohibited. They are optional, but it is still good practice to define them, when the instance variables are declared public. It is good practice to declare variables private and enable access to them with accessor methods because the get and set methods can do additional tasks that may be required when the values are accessed. Such tasks may include checking that the new value is within range, doing a conversion from an internal data type to an externally required one, updating other values in the class that depend on the variable, or as a convenience method that saves the programmer some time by doing a couple of steps in one call.

For example, in jMusic the `Note.setLength()` method is a convenience method that changes both the Note's rhythmValue and duration in concert. By default in jMusic when a note is created, the duration is set to 90% of the rhythmValue. To achieve different articulation—staccato or legato performance—the duration can be changed independently from the rhythmValue. It is a common oversight to change the rhythmValue of the Note without changing its duration. The `setLength()` method does both, and so using it avoids this common error. For a Note n, `n.setLength(2.0)` is the same as `n.setRhythmValue(2.0)` and `n.setDuration(2.0 * 0.9)`.

Method overloading

It is often the case that you want the same process to be applied to different types, for example, it is likely that you want to transpose a phrase or a part or a score. Java allows a class to contain methods with the same name if they each have a unique set of arguments. So, these transpose methods in jMusic's `Mod` class are each unique because they accept different arguments.

```
Mod.transpose(Phrase phr, amount) {}
Mod.transpose(Part p, amount) {}
Mod.transpose(Score s, amount) {}
```

This ability to have several methods of the same name in a class, is called method *overloading*.

```
Play.midi(drums);
```

Lastly, the music is played back using Java's internal synthesizer. Here is the complete code.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;
public class TheGrid2 implements JMC {
    private Phrase hats, snare, kick;
    private Part drums;
    public static void main(String[] args) {
        TheGrid2 grid = new TheGrid2();
          grid.compose();
          grid.output();
    }
    public TheGrid2() {
        hats = new Phrase(0.0);
        snare = new Phrase(0.0);
        kick = new Phrase(0.0);
        drums = new Part("Drums", 25, 9);
        drums.addPhrase(hats);
        drums.addPhrase(snare);
        drums.addPhrase(kick);
    }
    public void compose() {
         int[] hatsHits = {REST, REST, FS2, REST, REST, REST, FS2, REST,
             REST, REST, FS2, REST, REST, REST, FS2, REST,};
        hats.addNoteList(hatsHits, SIXTEENTH_NOTE);
        int[] snareHits = {REST, REST, REST, REST, D2, REST, REST, REST,
            REST, REST, REST, REST, D2, REST, D2, REST};
        snare.addNoteList(snareHits, SIXTEENTH_NOTE);
        int[] kickHits = {C2, REST, REST, REST, REST, REST, REST, C2, C2,
            REST, REST, REST, REST, REST, REST, REST};
        kick.addNoteList(kickHits, SIXTEENTH_NOTE);
    }
    public void output() {
        Write.midi(drums, "DrumPattern2.mid");
        Play.midi(drums);
    }
}
```

6: Design and Layout

Creating music is not just about making a single phrase, nor is writing Java programs just about defining one method. Sophisticated compositions and programs are made of many parts that fit together in a design that creates a working whole. In this chapter we begin to look at issues and techniques of arrangement and design of both music and code.

Viewing and Arranging Music

The arrangement of a musical work concerns its structure and texture, the organization of and relationships between musical events. When considering the arrangement of musical elements it is useful to consider their organization over time and space.

Musical structure over time is the most obvious aspect of arrangement. When considering this we think about when musical events start and stop, when they might reappear, how we might consider groups of events as a coherent block (such as phrase, sections, verses, choruses, and movements). In this chapter, we will explore techniques for placing musical sections in time.

Musical space is multi-dimensional, and we will explore techniques that control some of the dimensions now, and others in later chapters. A good place to start is to consider sounds in physical space. Their location can be organized from left to right in the stereo spectrum controlled by panning, and from front to back in an imaginary depth perspective controlled, largely, by loudness. Another consideration is texture; the number of concurrent parts and their interaction. As parts are added to the music the texture becomes thicker, but it need not be cluttered if attention is played to each parts timbre (sound quality), pitch range, and independence (or synchrony), to name some important considerations. As we proceed through this chapter techniques allowing the control of these features in Java will be introduced, as well as some tools to assist to judge the monitor the musical arrangement by seeing it, hearing it, and manipulating it.

We will see with the jMusic `HelperGUI` interface how the computer can be a useful tool for speeding up the processes of testing out compositional ideas. As we move through the rest of this chapter we will see how the computer can also start to act like a compositional *partner*. It will do this by taking on some of the responsibility for musical decision making.

As the *HelperTest* example shows, a class that extends the `HelperGUI` class must have a more sophisticated structure than simply having one main method. It will include a short `main` method that simply calls the constructor.

```
public static void main(String[] args){
    new HelperTest();
}
```

The constructor makes a call to the `HelperGUI` class; its *super* class, the one it extends, as part of the class instantiation process. This makes sure the `HelperGUI` class does its necessary set up procedures, and does any initializing the class requires. In this example, a variable (used for the initial pitch) is set in the constructor.

```
public HelperTest() {
    super();
    setVariableA(60);
}
```

Importantly, your class needs to have a `compose()` method, which will be run each time the compose button on the interface is clicked. It is in this method that all of your musical logic will go.

```
public Score compose() { …
```

Notice that the `compose()` method declaration indicates that it will return a `Score` object. This means that within the method you must declare a score, fill it with whatever data you want, then return the score as a last line in the method using the code,

```
return score;
```

As well as providing button to trigger composition of your score, the `HelperGUI` interface has buttons for starting and stopping playback of the score using JavaSound's General MIDI soundbank, so you can audition the composition. If you like what you hear, the score can be saved as a standard MIDI file using the `Write.midi()` button, with any name you specify in the filename field. The interface has buttons for viewing the score, which we'll explore later in this chapter, and there is an option to save as an audio file of the composition, which we will see more about later in this book.

Finally, the interface has five sliders labeled `variableA` – `E`. These can use used to set variables within your composition to change values of any parameter. Once values are changed on the interface, the compose button can be clicked to recalculate your composition with the new values, then you can audition and re-view the result. In our example program *HelperTest*, we have used `vari-`

for each note a pitch is randomly selected from that array. A call to the random method of Java's static Math class, `Math.random()`, results in a `double` value between 0.0 and 1.0. However, in this program we require an `int` between 0 and 4 so the random number needs to be scaled. We multiply by the number of pitches in the array (the array length) and cast to an integer so it is the correct type for the pitch value of the note constructor. Each time the piece is 'composed' a different melody is likely to be generated.

```
pitches[(int)(Math.random() * pitches.length)
```

This line is quite compact and could be rewritten with each step more explicit as,

```
int numberOfPitches = pitches.length;
double randomNumber = Math.random();
int pitchNumberToUse = (int)(randomNumber * numberOfPitches);
pitches[pitchNumberToUse];
```

Only the compose method of *RandomNotes* is printed here because otherwise the program is similar to the *HelperTest* program above.

```
public Score compose() {
    Score score = new Score("Random Notes", 140);
    Part xylophonePart1 = new Part("Xylophone",XYLOPHONE, 0);
    Phrase melody1 = new Phrase(0.0);
    int[] pitches = {C4,D4,E4,F4,G4};
    double[] rhythm = {EN,EN,SN,EN,EN,EN,EN,SN,EN};

    for (int i=0; i<rhythm.length; i++) {
        Note n = new Note(pitches[(int)(Math.random()*
            pitches.length)], rhythm[i]);
        melody1.addNote(n);
    }

    Mod.repeat(melody1, 8);
    xylophonePart1.addPhrase(melody1);
    score.addPart(xylophonePart1);
    return score;
}
```

Start times

Phrases can be positioned anywhere in the score by setting their start time variable, either when declaring the phrase,

```
        xylophonePart1.addPhrase(melody1);
        xylophonePart2.addPhrase(melody2);
        score.addPart(xylophonePart1);
        score.addPart(xylophonePart2);
        return score;
    }
```

Also the line below is added to the constructor to set `variableA` to 4.

```
setVariableA(4);
```

Variables can also have an associated label that appears on the `HelperGUI` interface to remind you what that variable does.

```
setVariableA(4, "Phrase 2 start");
```

Automatically appending phrases

In the same way that notes in a phrase follow one another in the order added, it is common that phrases follow one another without a break. Rather than having to calculate the new start times for each phrase in these situations, phrases can be declared without any start time at all. When such a phrase is added to a part it will automatically be placed at the end of the part. In the code fragment below, eight phrases will be added to the part end on end.

```
Part part1 = new Part("Melody", FLUTE, 0);
for (int i=0; i<8; i++) {
    Phrase phr1 = new Phrase();
    //… add notes to the phrase …
    part1.addPhrase(phr1);
}
```

Panning

The position of notes across the stereo field can be specified by their pan value. A pan value of 0.0 will have the sound out the left speaker only, a value of 1.0 will have the sound out the right speaker only. While a value of 0.5 (the default) will sound in the centre, equal volume from left and right speakers.

The *StartTimes* program can be modified to have the `melody1` phrase play to the left and the `melody2` phrase play to the right by using the phrase's `setPan()` method. This will effect all notes in the phrase at the time the method is called.

The volume of a note in jMusic is controlled by its dynamic value, an `int` with a value between 1 (softest) and 127 (loudest). For convenience, there are methods to change the dynamic of all notes in a phrase, part or score in one step. To change the dynamic values within a part so that all notes are at a specific level use the `phrase.setDynamic()` method and pass an integer value. Each note can have a separate dynamic value which can add richness and life to the music. If we wish to adjust the overall dynamic level of a part that has notes with varying dynamic levels, we an use the `Mod.increaseDynamic()` method, and pass the object to be modified (the part) and an `int` value for the amount of increase (a negative value for a decrease) to apply to each note. Use the `Mod.shake()` method to randomly adjusts all Notes' dynamic value to create uneven loudness. An `int` that specifies the degree of randomization is one argument to the `shake()` method.

The *MixingParts* program adds dynamic control to the *PanningPhrases* program. Variables `D` and `E` are used to control the volume of the two parts. These lines are added to the constructor to set the initial values.

```
setVariableD(100);
setVariableE(50);
```

These lines are added towards the end of the compose method to set the dynamic level of each part.

```
xylophonePart1.setDynamic(variableD);
xylophonePart2.setDynamic(variableE);
```

Viewing as a piano roll

Earlier we saw how phrases can be viewed as CPN. Another useful view of music is the piano roll view, named after the paper scrolls used on old pianolas; player pianos that played automatically powered by the energy of pumped air. In these scrolls the music was represented as holes in the paper, whose position on one axis was the pitch and whose length on the other was the note duration. As air was let through the holes the notes were played on the pianola. These piano roll displays are now common on many MIDI sequencing programs.

values in the score, displaying each part, phrase and note in turn. You can see this display of any music data object by passing it to the `View.print()` method. The display will be similar to figure 6.3.

```
******* SCORE: Scatter Piece. Contains 8 parts. *******
Tempo = 140.0 bpm
------- PART: Part 0. Contains 2 phrases. -------
Channel = 0
Instrument = 6
---------- PHRASE: Untitled Phrase. Contains 30 notes.  Start time: 88.9580309758615 ----------
Note: [Pitch = 81][RhythmValue = 0.5][Dynamic = 90][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 84][RhythmValue = 0.5][Dynamic = 54][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 86][RhythmValue = 0.5][Dynamic = 93][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 88][RhythmValue = 0.5][Dynamic = 34][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 86][RhythmValue = 0.5][Dynamic = 95][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 91][RhythmValue = 0.5][Dynamic = 35][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 81][RhythmValue = 0.5][Dynamic = 34][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 81][RhythmValue = 0.5][Dynamic = 34][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 79][RhythmValue = 0.5][Dynamic = 58][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 76][RhythmValue = 0.5][Dynamic = 68][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 76][RhythmValue = 0.5][Dynamic = 89][Pan = 0.8039950782602798][Duration = 0.45]
Note: [Pitch = 76][RhythmValue = 1.0][Dynamic = 71][Pan = 0.8039950782602798][Duration = 0.9]
```

Figure 6.3. Print view.

The `HelperGUI` interface has a button to provide a print list display of the current score.

Viewing as music notation

In addition to hearing your music played it can be useful to see a visual representation of it as well. jMusic supports a number of visualization displays which are accessed with the static `View` class. A frequently used music representation is common practice notation (CPN). The `View` class has a limited CPN facility that currently supports viewing a single phrase at a time. The code for calling this method is, `View.notate(phraseName);`.

To see the notation of the phrase used as a motif in the *PianoPhase* program, add a call to the `View` class after the phrase is created, as shown in this code sample.

```
Phrase phrase1 = new Phrase("Piano Phase Motif", 0.0);
int[] pitchArray = {E4,FS4,B4,CS5,D5,FS4,E4,CS5,B4,FS4,D5,CS5};
phrase1.addNoteList(pitchArray, SN);
View.notate(phrase1);
```

The result will appear similar to figure 6.4.

`lean`, true or false, result about weather a particular note is a member of the specified scale. Here is a simplified version of the method showing the workings of the random walk process.

```
private Phrase makePhrase() {
    Phrase phr = new Phrase();
    int pitch = 60;
    double[] rhythm = {EN,EN,SN,EN,EN,EN,EN,SN,EN};

    for (int i = 0; i < rhythm.length; i++) {
        Note note;
        do {
            pitch += (int)(Math.random()*10-5);
            note = new Note(pitch, rhythm[i]);
        } while (!note.isScale(PENTATONIC_SCALE));
        phr.addNote(note);
    }
    return phr;
}
```

The *Scatter* example uses all five `HelperGUI` variables to allow the exploration of different tempo and textual densities. The variables are often scaled to an appropriate amount. For example, `variableA` controls the tempo - but the tempo in beats per minute (BPM) is twice the variable readout. For example, a `variableA` value of 70 equals a tempo of 140 bpm. Below is the complete code for the *Scatter* example.

```
import jm.JMC;
import jm.util.*;
import jm.music.data.*;
import jm.music.tools.*;
import jm.gui.helper.HelperGUI;

public final class Scatter extends HelperGUI implements JMC{
    public static void main(String[] args){
        new Scatter();
    }

    public Scatter() {
        super();
        setVariableA(70, “Tempo”);
        setVariableB(60, “# of parts”);
        setVariableC(2, “Phrases per part”);
```

Math Moment

Algorithms and Heuristics: An algorithm, to be precise, is a process that is guaranteed to produce a specified outcome. Many times, particularly where aesthetic judgments are involved, an answer which is mostly true or close to optimal will be fine. A rule of thumb that almost always gives a 'resonable' answer is called a heuristic. In many cases it is quite acceptable to use a heuristic, rather than an algorithm, and sometimes it may make no practical difference. Because algorithms are guaranteed to give the correct answer they may be quite slow to execute as they search through all possible answers to make sure they provide the best one. For example, in computer chess programs, an algorithm to find the best move would need to search every possible end game of every possible combination of moves available. Because this is a ridiculously large number and would (at best) make the game very slow and boring, chess programs use heuristics which may provide the best answer on occasion but provide a very good one in most cases. Many of the 'limitations' of computers turn out to be the limitations of algorithms and, consequently, our ability to adequately describe a process.

Structure

The importance of structure can be heard in the next example program, *MelodicStructure*. The form of this piece is a simple nursery rhythm structure with four sections A, B, C and D where the A and C sections are reused (copied) to add a sense of familiarity through repetition. Each section is four beats long except section D which is eight beats long. The form is A, B, A, C, D, A, C. This example uses a random walk melody in the `makePhrase()` method very similar to the *Scatter* program, the argument passed is the length of the generated phase, in beats. The strength of the example is to hear how this simple structure can add cohesion to a random series of notes. The format of the *MelodicStructure* program is similar to the others in this chapter, therefore only the critical compose method is listed below.

```
public Score compose() {
    Phrase a = makePhrase(4.0);
    Phrase b = makePhrase(4.0);
    Phrase c = makePhrase(4.0);
    Phrase d = makePhrase(8.0);
    Phrase a1 = a.copy();
    Phrase a2 = a.copy();
    Phrase c1 = c.copy();
    Part part = new Part();
    Score score = new Score (130);

    part.addPhrase(a);
```

```
    public static void main(String[] args) {
        new Journey();
    }
```

The first few lines import the required jMusic packages and classes. After a comment to describe the piece, the class is declared. First, a `Score` object is created at the class level. This means that this object will be 'visible' to all methods in this class. Because it is `private` it can't be seen outside this class.

The first method is `main()` in which all we do is create a new instance of the `Journey` class. In effect the `main()` method simply calls the constructor of this class. The `new` keyword is important in these lines of code. It is used to create an instance of the `Score` class, then used to create an instance of the `Journey` class.

```
public Journey() {
    for (int i=0; i<8;i++) {
        new NPart(newPart());
    }
    View.print(score);
    Play.midi(score);
}
```

The code above shows the constructor method for the `Journey` class. Remember that the constructor method has the same name as the class (`Journey` in this case) and does not have a return type specified in the first line like all other methods do.

This method provides an overview of our compositional process. First, eight new parts are created in a loop. Next, the `View` class is used to print out the music as text so we can see what was generated visually. Thirdly, the `Play` class is called upon to playback the composition.

The for-loop iterates over one line. This line uses the `new` keyword to create instances of the `NPart` class (we'll see more of it below). The `NPart` class takes a `Part` as an argument. This is created by the `newPart()` method because we want to set some values of the part before giving it over to the `NPart` class.

```
    private Part newPart() {
        Part p = new Part();
        p.setChannel(score.size());
        score.addPart(p);
        return p;
}
```

which makes the piece more interesting. The variable `nPartStartTime` contains the time (in beats) when the music of this part should begin. The variable `panPosition` holds the information about where in the stereo spectrum notes from this part should emerge.

Java Moment

The breaking up of this program into classes and methods makes the code easier to read and edit, and multiple method calls save repetitive typing. The division of the program into more than one class file also keeps each file manageable in size, and provides a physical division where there is a logical (compositional) one. In this case the operations on parts is kept separate from working on the score structure of those parts.

Let's examine the NPart class.

```
public NPart(Part p) {
    this.part = p;
    p.setInstrument((int)(Math.random()*60));
    nPartStartTime = (int)(Math.random() * 200) * SN;
    p.addPhrase(makePhrase());
}
```

There are two methods in this class, the first (above) is the constructor. Remembering that constructor methods have the same name as the class. The constructor is run (called) when a new instance of the class is made (constructed). Within the constructor the part passed from the `Journey` class is assigned to the local variable part. Next, a General MIDI instrument number for this `Part` is selected at random between 0 and 59. Then a start time for this `Part` is calculated using a random value somewhere within the first 200 semiquaver (sixteenth- note) steps of the piece.

Each part in this piece has just one `Phrase` and the last line calls the `makePhrase()` method to make it and adds it to the part, all in one fell swoop.

```
private Phrase makePhrase() {
    Phrase phr = new Phrase( nPartStartTime);
    phr.addNoteList(pitches, rhythms);
    phr.setPan(Math.random());
    Mod.shuffle(phr);
    Mod.cycle(phr, partLength);
    Mod.fadeIn(phr, fadeInLength);
    Mod.fadeOut(phr, fadeOutLength);
    return phr;
}
```

Java GUI Classes

One of the advantages of programming applications in Java is that they can be cross platform. The same program can run on the Macintosh, Windows and Linux operating systems and beyond. This is facilitated for end-user software by Java packages that describe its own graphical user interface classes for drawing windows, buttons, and more and for the handling mouse and keyboard interactions and the like. The disadvantage of providing platform-independent interface libraries is that Java applications often look like, well, Java applications that may not always look the same as native applications. In many cases this is a small price to pay for the universality, and if you write your own interface components (like many music applications have a tendency to do) them this "unofficial" appearance is of no concern. In this section we will introduce some of Java's user interface classes, but a comprehensive coverage is well beyond the current scope and the interested reader is directed to seek one some of many good reference sources on Java GUI programing.

Introducing the AWT package

The abstract windowing toolkit (AWT) is Java's original graphical user interface library. It handles many of the basic interface attributes for Java and many aspects of it are still widely used. The AWT package has elements relating to four aspects of interaction:

1. Graphics - colors, drawing, images and so on.
2. Components - windows, buttons, sliders and so on.
3. Layout Managers - component positioning on screen
4. Event Handlers - tracking and handling mouse, keyboard and other interaction events

To access these features a program imports the `java.awt.*` classes.

Listeners

One of the important features of the AWT package are `ActionListeners` that receive `ObjecEvents` generated by components or interaction with components. Typically, this includes the monitoring of and responding to mouse clicks, slider movements, button clicks, and so on.

Interactions with components can generate events. For example a `Button` object, when clicked, generates a ActionEvent. All events have a corresponding event handling method call a listener. For example, `ActionEvents` are sent to an `actionPerformed()` method. There are other event-listener pairings provided by a range of AWT interfaces. A commonly used one is the `MouseListener` interface that responds to `MouseEvents` and has a number of handler methods

Components

Many user interface classes are "components" like buttons, sliders, scrollbars and so on. The `JComponent` class is fundamental to Swing and the other component classes extend `JComponent`. The `JComponent` has attributes and behaviors such as setting its size, noticing mouse clicks on itself, setting its color, making itself visible or invisible, and many others.

Frames

Graphical interfaces need a main window and this is provided by the `JWindow` or `JFrame` classes. A `JFrame` is a `JWindow` with a title bar, resize tab and other niceties, so it's the one we tend to use for applications. The `JFrame` constructor accepts, as an argument, a name to display in the title bar. Here is a simple application that displays an empty window.

```
import javax.swing.*;

public class DemoWindow {
    public static void main(String[] args) {
        String name = "My App Name";
        JFrame f = new JFrame(name);
        f.setSize(600, 400);
        f.setVisible(true);
    }
}
```

Containers such as `JFrames` implement a layout manager that controls the organization of other components added to it. We will not discuss layout managers in detail here but be aware that the default layout manager is called `FlowLayout` and simply arranges components one after the other side by side within the container.

Buttons

`JButton` objects create a GUI button with name and/or icon that when clicked by the user generates an `ActionEvent`. As mentioned above, `ActionEvents` are handled by the `actionPerformed()` method which is part of the `ActionListener` interface. Let's add a button to out demo application that prints to the console when clicked.

```
import javax.swing.*;
import java.awt.event.*;

public class DemoWindow2 implements ActionListener {
```

```
        new DemoWindow3();
    }
    public DemoWindow3() {
        f = new JFrame(name);
        JPanel panel = new JPanel();
        f.add(panel);
        box1 = new JCheckBox("Mute");
        box1.addActionListener(this);
        panel.add(box1);
        f.setSize(600, 400);
        f.setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == box1) {
            if (box1.isSelected()) {
               System.out.println("Playback muted");
            } else System.out.println("Playbck unmuted");
        }
    }
}
```

Sliders

Sliders allow users to control variables within the program. The `JSlider` objects are typically established with an orientation, minimum, maximum, and starting values. Slider movements generate `ChangeEvents` that can be handled by the `stateChanged()` method of the `ChangeListener` interface. The `ChangeListener` interface was introduced as part of swing so you need to import the `javax.swing.event.*` classes to access it. Below is code that implements a volume slider.

```
import javax.swing.*;
import javax.swing.event.*;

public class DemoWindow4 implements ChangeListener {
    JSlider slider1;

    public static void main(String[] args) {
        new DemoWindow4();
    }
    public DemoWindow4() {
        String name = "My App name";
```

TheGrid3

This program will add a display showing hi-hat rhythm as a series of check boxes, where crosses or ticks represent sounding notes. The result will appear similar to figure 6.5.

Figure 6.5: TheGrid3 appearance.

Structurally there are three new methods in this version of TheGrid. The GUI code is in the `makeInterface()` method, and we've separated the MIDI file writing and playback each to their own methods. The constructor method will call each of these methods in turn. Here is a code fragment from the constructor method where this occurs.

```
compose();
makeInterface();
saveFile();
play();
```

The compose method is unchanged from *TheGrid2* and the `saveFile()` and `play()` methods simply have the one line of code required to do their job. Therefore, we will concentrate on the new code in the `makeInterface()` method.

Here is the complete method.

```
private void makeInterface() {
    JFrame window = new JFrame("Drum pattern");
    JPanel grid = new JPanel(new GridLayout(1,16));
    for (int i=0; i<16; i++) {
        JCheckBox tick = new JCheckBox();
        if (hats.getNote(i).getPitch() != REST) {
            tick.setSelected(true);
        }
        grid.add(tick);
    }
    window.getContentPane().add(grid);
    window.pack();
    window.setVisible(true);
}
```

in the `GridLayout` in the order they are added, from the top down. The really new information in this version of TheGrid is the buttons.

```
playBtn = new JButton("Play");
btnPanel.add(playBtn);
saveBtn = new JButton("Save MIDI file");
btnPanel.add(saveBtn);
window.getContentPane().add(btnPanel);
```

The code fragment above shows the instantiation and addition of the buttons. A `JButton` constructor takes the button name to be displayed as an argument. Each button is added to a (previously created) panel called `btnPanel`. The `btnPanel` uses the default `FlowLayout` so each component is displayed aligned to the center and from left to right and top to bottom as space allows. Finally, the `btnPanel` is added to the frame.

Below is the complete source code for *TheGrid4* application.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;
import java.awt.*;
import javax.swing.*;

public class TheGrid4 implements JMC {
    private Phrase hats, snare, kick;
    private Part drums;
    private JCheckBox[] hatTicks, snareTicks, kickTicks;
    private JButton playBtn, saveBtn;

    public static void main(String[] args) {
        new TheGrid4();
    }

    public TheGrid4() {
        hats = new Phrase(0.0);
        snare = new Phrase(0.0);
        kick = new Phrase(0.0);
        drums = new Part("Drums", 25, 9);
        drums.addPhrase(hats);
        drums.addPhrase(snare);
        drums.addPhrase(kick);
```

```
            grid.add(tick);
            hatTicks[i] = tick;
        }
        window.getContentPane().add(grid);
        // snare
        JPanel grid2 = new JPanel(new GridLayout(1,16));
        for (int i=0; i<16; i++) {
            JCheckBox tick = new JCheckBox();
            if (snare.getNote(i).getPitch() != REST) tick.setSelected(true);
            grid2.add(tick);
            snareTicks[i] = tick;
        }
        window.getContentPane().add(grid2);
        // kick
        JPanel grid3 = new JPanel(new GridLayout(1,16));
        for (int i=0; i<16; i++) {
            JCheckBox tick = new JCheckBox();
            if (kick.getNote(i).getPitch() != REST) tick.setSelected(true);
            grid3.add(tick);
            kickTicks[i] = tick;
        }
        window.getContentPane().add(grid3);
        JPanel btnPanel = new JPanel();
        //  buttons
        playBtn = new JButton("Play");
        btnPanel.add(playBtn);
        saveBtn = new JButton("Save MIDI file");
        btnPanel.add(saveBtn);
        window.getContentPane().add(btnPanel);

        window.pack();
        window.setVisible(true);
    }
    private void saveFile() {
        Write.midi(drums, "DrumPattern4.mid");
    }
    private void play() {
        Play.midi(drums, false);
    }
}
```

7: Building Interfaces

Having in previous chapters created classes using the HelperGUI and making graphical user interfaces for *TheGrid* application, and with your experience using many other software applications, you will be well aware of how much a graphical interface can assist the use of software. In this chapter we will walk you through the process of creating your own music applications with a GUI.

Graphical User Interfaces

Let's have a close look at the code first.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import jm.JMC;
import jm.music.data.*;
import jm.util.*;

public class SimpleGUI extends JFrame implements ActionListener, JMC {
    private JButton composeBtn;

    public static void main(String[] args) {
        SimpleGUI gui = new SimpleGUI();
    }

    public SimpleGUI() {
        composeBtn = new JButton("Compose");
        composeBtn.addActionListener(this);
        this.getContentPane().add(composeBtn);
        this.pack();
        this.setVisible(true);
    }

    public void actionPerformed(ActionEvent ae){
        if (ae.getSource() == composeBtn) compose();
```

ate button object by getting the 'source' of the `actionEvent`. Then, if the comparison is true, the `compose()` method is called and a short statement printed to the terminal.

Adding GUI details

The *SimpleGUI* application, was a minimal implementation and you may have noticed some behavior inconsistencies with other applications. For example, when the frame was resized, by dragging it from the bottom right corner, the button continued to expand to fill the entire window. Also, when the close or exit button was clicked the window may have closed, but the application was still running. In this section we'll fix these details. All the changes take place in the constructor. Assuming you copy the previous class and rename it *SimpleGUI2*, here is a our new constructor.

```
public SimpleGUI2() {
    composeBtn = new JButton("Compose");
    composeBtn.addActionListener(this);
    JPanel panel = new JPanel();
    panel.add(composeBtn);
    this.getContentPane().add(panel);
    this.setSize(200, 100);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setVisible(true);
}
```

In order to prevent the button expanding when we resize the frame we will first add the button to a `JPanel` object, then insert the panel into the frame. One line creates a new `JPanel` called panel, and a second adds the `composeBtn` object to the panel. The panel is then added to the frame.

The previous program's appearance was somewhat crowded because we used the `pack()` method to keep everything as small as possible. This time we'll specify a dimension for our frame using the `setSize()` method, that takes pixel values for width and height as arguments.

In order to ensure that the program exits when we close the window we use this self explanatory method that takes a constant from the `JFrame` class as an argument.

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

The results should appear similar to figure 7.2.

The `saveAndPlay()` method saves the music as a MIDI file and plays it back. So that multiple versions of the music can be saved one after another, each MIDI file is give a unique name using the counter variable that increments each time.

```
public void saveAndPlay() {
    Write.midi(arpPart, "RandomArp" + counter++ + ".mid");
    Play.midi(arpPart, false);
}
```

This class can save multiple MIDI files of the score, and each is numbered using the counter variable. The counter variable is used to keep track of how many MIDI files have been created and it increments each time a file is written. The maintenance of the counter value is possible because it is an instance that persists as long as the class lives. The `Play.midi()` method takes a second `Boolean` argument, false, to indicate that the program should not exit after playing back the music.

Changing variables via the interface

In this final elaboration of the *SimpleGUI* applications, we'll add a slider to the interface that will control the length of the arpeggio. The *SimpleGUI4* program will look similar to figure 7.3 when run.

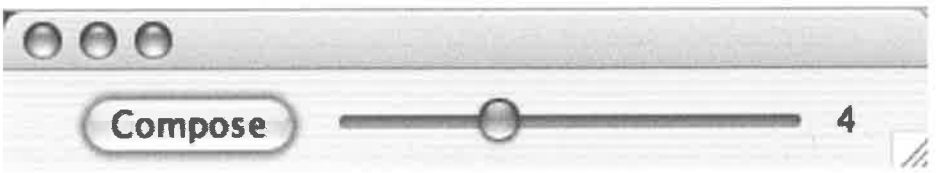


Figure 7.3. Interface of the SimpleGUI4 program.

The implementation of GUI slider components was outlined in Chapter 6, and a review of the process is outlined below. Here is the complete code for *SimpleGUI4*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import jm.JMC;
import jm.music.data.*;
import jm.util.*;
import jm.music.tools.Mod;

public class SimpleGUI4 extends JFrame
```

```
    }

    public void compose() {
        Phrase phr = new Phrase();
        arpPart = new Part("Apreggio", SYNTH_BASS, 0);
        for (int i=0; i<arpLength; i++ ) {
            phr.addNote(new Note(rootNote +
            pattern[(int)(Math.random() * pattern.length)],
                EIGHTH_NOTE));
        }
        Mod.repeat(phr, 8);
        arpPart.addPhrase(phr);
        arpPart.setTempo(130.0);
        saveAndPlay();
    }

    public void saveAndPlay() {
        Write.midi(arpPart, "RandomArp" + counter++ + ".mid");
        Play.midi(arpPart, false);
    }
}
```

Adding the slider involves a few steps. The addition of the slider component itself, a label component to display the slider value, and implementation of the `ChangeListener` interface to detect slider movements.

In order to access the `ChangeListener` interface the `javax.swing.event.*` package is imported. The class declaration includes the implementation of the `ChangeListener` interface that will detect slider movements. The `JSlider` and `Jlabel` classes from the Swing library are used. The slider and label instance variables are declared near the top of the file and instantiated in the constructor. These are each added to the panel object and the size of the frame is altered to allow the three components to fit. In order that movements of the slider are monitored, the slider object is registered with the `ChangeListener` using the `addChangeListener()` method. This method (like all listener registering methods) takes as an argument the class that will handle the slider changes, in our case it is this class itself; 'this'. The `ChangeListener` implementation requires the `stateChanged` method to be overridden. This method is called when the slider is moved. Inside that method the new slider value is assigned to the `arpLength` variable (controlling the length of the arpeggio pattern created by the compose method), and the label

The 'static' keyword

Variables and methods within a class that are declared static are shared by each instance of the class. This contrasts with instance variables and methods where each instance of the class has a unique copy of the variable or method. Static variables should to be used sparingly because they 'break' data encapsulation which is a strong aspect of object-oriented design, but they are useful because they can hold data that can be shared amongst all instances of the class, they only take up memory in the class and not in each instance of the class, and static variables and methods may be accessed by other classes even when no instance of the class has been created.

Static variables

The data in a static variable resides with the class and is accessible from all instances of the class. Use the static keyword to declare a variable as static.

```
public static tempo = 120.0;
```

If this tempo variable were in a class named `Voice`, then each instance of the class would share the same tempo.

```
Voice soprano = new Voice();
Voice alto = new Voice();
Voice tenor = new Voice();
Voice baritone = new Voice();
```

When a change to the tempo was made, `Voice.tempo = 85.0;` then all voice instances would be effected.

Static methods

A method is declared static using the static modifier.

```
public static myMethod()
```

Static methods are used quite frequently in jMusic. For example the `midi()` method in the `Play` class is static, as are most methods in the `Mod` class such as `repeat()`. They are used less frequently in Java itself, however, a notable example is the `Math` class which has many static methods including `Math.random()`.

The most common use of a static method is Java's `main()` method which must be declared static. This is because it would be very confusing for Java to decide where to start executing a program if there could be more than one `main()` method.

Java Moment

In Java, a `listener` is a class or interface that monitors events in your program, for example when the user clicks a button, and sends a notification when the event occurs to all classes that have registered with the listener. All listeners extend the `java.util.EventListener` class which sets up a thread in the background to regularly check of events, a process called polling. Most `listeners` in the Java libraries are `interfaces`, so a class that want to use a listener must implement the interface. It must also register the objects that it wants the listener to monitor and pass the class to notify as an argument – usually the class to be notified is the current class, so the this keyword is used to mean 'this class'.

Listeners

This code fragment shows how a particular area (panel) within the GUI can respond to mouse clicks within it. By implementing the `MouseListener` interface and registering a panel with the listener.

```
public class MyInterfaceClass implements MouseListener {
    // other class code here.
    Panel displayArea = new Panel();
    displayArea.addMouseListener(this);
    // other class code here..
}
```

When an event is 'heard' by the listener it 'tells' the registered classes by calling a specified method that all registering classes must implements. In the case of a mouse clicking on an button the `MouseListener` it calls a `mouseClicked()` method in each of the registered classes. Each listener responds to specific types of events. The `MouseListener` responds to `MouseEvents`. The methods that are called by the listener when an event occurs as passed the event. So the mouse`Clicked()` method takes one argument with an is `MouseEvent` object, often named `e` by convention. Below is a simple implementation of the `mouseClicked()` method that might be used when monitoring a GUI button named `startBtn`.

```
public void mouseClicked(MouseEvent e) {
    if (e.getSource() == startBtn) startMusic();
    ...
}
```

There are a number of listeners in the Java `AWT` and `Swing` libraries. The two required for this application are the `ItemListener`, which can handle check box changes, and the `ActionListener`, which handles button clicks.

Our application is now interactive! You can switch on or off each drum on any of the 16 subdivisions, and play back or save the drum pattern whenever you like. Below is the full code for *TheGrid5*.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class TheGrid5 implements JMC, ItemListener, ActionListener {
    private Phrase hats, snare, kick;
    private Part drums;
    private JCheckBox[] hatTicks, snareTicks, kickTicks;
    private JButton saveBtn, playBtn;

    public static void main(String[] args) {
        new TheGrid5();
    }
    /**
    * Constructor. Sets up the parameters.
    */
    public TheGrid5() {
        hats = new Phrase(0.0);
        snare = new Phrase(0.0);
        kick = new Phrase(0.0);
        drums = new Part("Drums", 25, 9);
        drums.addPhrase(hats);
        drums.addPhrase(snare);
        drums.addPhrase(kick);
        // set look and feel
        String local = javax.swing.UIManager.getSystemLookAndFeelClassName();
        String metal =
             javax.swing.UIManager.getCrossPlatformLookAndFeelClassName();
        try {
            UIManager.setLookAndFeel(local); // use local or metal
        } catch (Exception e) {}
        hatTicks = new JCheckBox[16];
        snareTicks = new JCheckBox[16];
        kickTicks = new JCheckBox[16];
```

```
        window.getContentPane().add(grid);
        // snare
        JPanel grid2 = new JPanel(new GridLayout(1,16));
        for (int i=0; i<16; i++) {
            JCheckBox tick = new JCheckBox();
            tick.addItemListener(this);
            tick.setHorizontalAlignment(SwingConstants.CENTER);
            if (snare.getNote(i).getPitch() != REST) tick.setSelected(true);
            grid2.add(tick);
            snareTicks[i] = tick;
        }
        window.getContentPane().add(grid2);
        // kick
        JPanel grid3 = new JPanel(new GridLayout(1,16));
        for (int i=0; i<16; i++) {
            JCheckBox tick = new JCheckBox();
            tick.addItemListener(this);
            tick.setHorizontalAlignment(SwingConstants.CENTER);
            if (kick.getNote(i).getPitch() != REST) tick.setSelected(true);
            grid3.add(tick);
            kickTicks[i] = tick;
        }
        window.getContentPane().add(grid3);
        JPanel btnPanel = new JPanel();
        //  buttons
        playBtn = new JButton("Play");
        playBtn.addActionListener(this);
        btnPanel.add(playBtn);
        saveBtn = new JButton("Save MIDI file");
        saveBtn.addActionListener(this);
        btnPanel.add(saveBtn);
        window.getContentPane().add(btnPanel);
        window.pack();
        window.setVisible(true);
    }
    /**
    * Handle the checkbox clicks
    */
    public void itemStateChanged(ItemEvent e) {
        for (int i=0; i<16; i++) {
```

TheGrid6

In *TheGrid6* application labels are added to each of the drums and controllers are added to control tempo and repetition.

Labels

So that it is clear which row of checkboxes is for hi hats and which is for snare or kick, we'll add a label to the left of each grid.

```
JPanel grid = new JPanel(new GridLayout(1, 17));
grid.add(new JLabel("Hats"));
```

The grid width is increased to 17, to allow for the label and new `JLabel` is created and added to the first (left most) cell in the grid. This process in repeated for the snare and kick grids also.

JComboBoxes

Up till now, TheGrid application has played only one bar of music and always at the same speed. In *TheGrid6* we remedy this and add a new component type, the `JComboBox`. A `JComboBox` is a control that provides a pop-up menu to select from. This application adds one for selecting the tempo and another for selecting the number of repetitions.

```
private JComboBox repeatMenu, tempoMenu;
```

The new components are declared at the top of the file as private instance variables so they are available to all methods within instances of this class.

```
repeatMenu = new JComboBox();
repeatMenu.addActionListener(this);
repeatMenu.addItem("1");
repeatMenu.addItem("2");
repeatMenu.addItem("4");
repeatMenu.addItem("8");
repeatMenu.addItem("16");
```

This code fragment shows the instantiation of the `repeatMenu` object as a `JComboBox`, its registration with the `ActionListener`, and the adding of items to be displayed in the pop-up menu.

```
btnPanel.add(new JLabel("Repeats = "));
btnPanel.add(repeatMenu);
```

```
    String metal =
         javax.swing.UIManager.getCrossPlatformLookAndFeelClassName();
    try {
        UIManager.setLookAndFeel(local); // use local or metal
    } catch (Exception e) {}
    // set graphic data
    hatTicks = new JCheckBox[16];
    snareTicks = new JCheckBox[16];
    kickTicks = new JCheckBox[16];
    // call other methods
    compose();
    makeInterface();
}
/**
* Do the real musical work
*/
public void compose() {
     int[] hatsHits = {REST, REST, FS2, REST, REST, REST, FS2, REST, REST,
         REST, FS2, REST, REST, REST, FS2, REST,};
    hats.addNoteList(hatsHits, SIXTEENTH_NOTE);
    int[] snareHits = {REST, REST, REST, REST, D2, REST, REST, REST, REST,
         REST, REST, REST, D2, REST, D2, REST};
    snare.addNoteList(snareHits, SIXTEENTH_NOTE);
    int[] kickHits = {C2, REST, REST, REST, REST, REST, REST, C2, C2,
         REST, REST, REST, REST, REST, REST, REST};
    kick.addNoteList(kickHits, SIXTEENTH_NOTE);
}
/**
* GUI code
*/
private void makeInterface() {
    JFrame window = new JFrame("Drum pattern");
    window.getContentPane().setLayout(new GridLayout(5,1));
    // numbers
    JPanel numbGrid = new JPanel(new GridLayout(1,17));
    numbGrid.setPreferredSize(new Dimension(400, 20));
    numbGrid.add(new JLabel(""));
    for (int i=1; i<17; i++) {
        numbGrid.add(new JLabel(""+i, SwingConstants.CENTER));
    }
```

```
        playBtn = new JButton("Play");
        playBtn.addActionListener(this);
        btnPanel.add(playBtn);
        saveBtn = new JButton("Save as MIDI file");
        saveBtn.addActionListener(this);
        btnPanel.add(saveBtn);
        // repeats
        repeatMenu = new JComboBox();
        repeatMenu.addItem("1");
        repeatMenu.addItem("2");
        repeatMenu.addItem("4");
        repeatMenu.addItem("8");
        repeatMenu.addItem("16");
        btnPanel.add(new JLabel("Repeats = "));
        btnPanel.add(repeatMenu);
        // tempo
        tempoMenu = new JComboBox();
        tempoMenu.addActionListener(this);
        tempoMenu.addItem("60");
        tempoMenu.addItem("80");
        tempoMenu.addItem("100");
        tempoMenu.addItem("120");
        tempoMenu.addItem("140");
        tempoMenu.addItem("180");
        btnPanel.add(new JLabel("Tempo = "));
        btnPanel.add(tempoMenu);
        btnPanel.add(new JLabel("bpm"));
        window.getContentPane().add(btnPanel);
        window.pack();
        window.setVisible(true);
    }
    /**
    * Handle the checkbox clicks
    */
    public void itemStateChanged(ItemEvent e) {
        for (int i=0; i<16; i++) {
            if (e.getSource() == hatTicks[i]) {
                if (hatTicks[i].isSelected()) hats.getNote(i).setPitch(FS2);
                else hats.getNote(i).setPitch(REST);
                return;
```

8: Combination and Recombination

Digital audio processes can produce large scale works, and automation can allow us to create enormous amounts of sound with just a little code. However, matching this ability to scale is the considerable size of digital audio tasks, where each and every digital audio outcome must be specified sample by sample. The ability to control musical structures at all levels of scale is a powerful attribute of computer music. In this chapter we'll examine some processes of structuring music in large and small chunks and examine how, at a micro level, we can even control the sound of our music at the sample-by-sample level, and how, at a macro level, we can organize musical form. This ability to explore music at various levels of granularity is made accessible because we can utilize Java classes to operate on data from sample to score.

Musical Chunks

The jMusic data structure provides levels of granularity with the `Note`, `Phrase`, `Part` and `Score` objects. It is possible to algorithmically control the structure of your music at any of these levels. We've seen often in this book how this is achieved at the note level, but larger chunks – especially phrases – can be usefully arranged as well. The texture of the music can be altered by turning on or off particular parts, and larger forms (such as rondo or verse-chorus forms) can be modeled by using whole scores as musical sections.

Chords

A chord occurs when several notes sound together. In our exploration thus far, we've concentrated on monophonic and polyphonic music where the musical thinking focuses on horizontal musical lines. Chordal, or homophonic, musical thinking is concerned with the vertical dimension. Notes which play together, like when the six strings of a guitar are strummed, form a chord. jMusic provides a way to add notes as chords to parts using the `CPhrase` class.

CPhrases

The `CPhrase` (chord phrase) class allows the composer to construct homophonic musical structures easily. That is, provide a convenient way for jMusic to manage chords. A jMusic chord is a group of notes that share the same onset time and duration but have different pitches. The

Recombination

The algorithmic arranging of existing musical chunks in this way is often referred to as recombination or, more formally, musical combinatorics which dates back as far as the ars combinatoria published by the German philosopher and mathematician Gottfried Wilhelm Leibnitz in 1666 (Selfridge-Field 2001). A famous example of recombination is Musikalisches Würfelspiel, a dice game attributed to Mozart where the results of throws choose amongst possible prewritten bars of music.

Musical Moment

Mozart's Dice Music: Mozart's Musikalisches Würfelspiel combines fragments of music into a short two-part waltz. Depending upon the numbers thrown, different, but similar, musical works can result. At each of sixteen rolls of the die there are eleven musical fragments to choose from. At each stage the fragments are interchangeable because they share an underlying harmonic progression and structural function. You can read more about the workings of Musikalisches Würfelspiel in "The Computer Music Tutorial" (Roads 1996:823), and the full score for it can be found in the Appendix of the book "Machine Models of Music" (Schwanauer and Levitt 1993).

The *MozartDiceGame* application is an implementation in Java of this game, and the process is repeated twice to create a thirty-two measure composition, in two distinct halves. The program is in two files. The `MozartDiceGame.java` file has the main method and generates the random numbers and assembles the score. The `MozartPhraseList.java` file has the data, it sets up a matrix (two dimension array) of one-measure-long phrases.

You may like to look at an implementation of the Mozart Dice Game at the tutorials page of the jMusic web site. http://jmusic.ci.qut.edu.au/

Assembling rhythms

In this section we will examine a simple recombinational program that will create rhythmic phrases from individual one-beat rhythm patterns.

All recombinatorial processes have three stages. First, the collection or creation of the musical data available for use. Second, the selection and organization of that data into a whole and, thirdly, the rendering or performing of the completed score.

```
import jm.JMC;
import jm.music.data.*;
```

musical form or structure. In jMusic, notes added to phrases are sequenced one after another, similarly, when phrases are added to parts are similarly added end on end. This behavior can be overridden by specifying a specific start time for a phrase. The program *MelodicStructure* shows how significant phrase-level structure can be by piecing together random notes into a coherent work. The final structure will be in the form A B A C D A C.

The *MelodicStructure* application creates phrases with its `makePhrase()` method. This method chooses pitches as a random walk in C minor and uses recombined rhythmic cells. See the source code below for the full implementation. For now we will examine some relevant code fragments.

```
Phrase a = makePhrase(4.0);
Phrase b = makePhrase(4.0);
Phrase c = makePhrase(4.0);
Phrase d = makePhrase(8.0);
```

Four phrases are constructed by calling this method. `Phrases` a, b, c and d. Each passes an argument to the `makePhrase()` method indicating how long it should be, either 4 or 8 beats.

```
Phrase a1 = a.alias();
Phrase a2 = a.alias();
Phrase c1 = c.alias();
```

Because some phrases a reused in the form, aliases of them are created. Aliases of phrases in jMusic are very similar to copies; they are new objects that are identical to the original but can have their own start time, tempo and other attributes. The difference is that an alias refers to the same note objects as the original, such that any changes made to the notes in the original will be reflected in the alias, while a copy of a phrase makes new objects for all notes therefore changes to one notes in one do not effect the copied notes.

```
Part p = new Part();

p.addPhrase(a);
p.addPhrase(b);
p.addPhrase(a1);
p.addPhrase(c);
p.addPhrase(d);
p.addPhrase(a2);
p.addPhrase(c1);

View.show(p);
```

```
        score.addPart(part);
        return score;
    }
    // Generate a phrase based on a random walk
    private Phrase makePhrase(double beats) {
        Phrase phr = new Phrase();
        int pitch = (int)(Math.random()* 24 + 48);
        for (int i = 0; i < beats * 2; i++) {
            Note note;
            do {
               pitch += (int)(Math.random() * 10 - 5);
               if (pitch < 0) pitch = 0;
               if (pitch > 127) pitch = 127;
               note = new Note(pitch, EN, (int)(Math.random()* 70 + 30));
            } while (!note.isScale(PENTATONIC_SCALE));
            phr.addNote(note);
        }
        Mod.accents(phr, 2.0);
        return phr;
    }
}
```

Macrostructure & microstructure

Creating musical forms at the phrase level, as we did in the previous section, is useful and common place. In this section we'll see how larger data chunks (parts and score) and smaller chunks (notes and sample data) can be manipulated to create musical structures. Larger chunks allow us to manipulate the macrostructure of the work, and organizing the music at the sample level allows us to change the microstructure.

Appending sections

As we have already seen, notes, phrases, parts and scores can be used to structure our work. The *AppendRhythms* program will demonstrate how large sections, even whole scores, can be combined one after another into large-scale musical forms.

Phrases are generated by a `makePattern()` method that returns a rhythmic phrase pitched to a random percussion instrument. These phrases form the basis of larger data constructs. The `AppendRhythms` class applies the `Mod.append()` method at three levels, phrase, part and score.

```
Phrase phr = makePattern();
```

`StartTime` parameter. Below is the full code for the *Spray* example that includes extensive comments. Make sure you substitute your own audio file for processing in place of the "Welcome.au" file name in the code.

```
import java.io.*;
import jm.JMC;
import jm.music.data.*;
import jm.util.*;
import jm.audio.*;
import jm.audio.io.*;
import jm.audio.synth.*;

public class Spray implements JMC {

	public static void main(String[] args) {
		new Spray();
	}

	public Spray() {
		Score score = new Score();
		Part p = new Part("Spray",0,0);
		Phrase phr = new Phrase();
		String fileName = "Welcome.au";
		// get file size
		int numb = (int)(Math.random()*10)+1;
		File f = new File(fileName);
		double fileSize = (double)(f.length() - 32)/44100/2/2;
		// make instrument stuff
		Instrument[] ensemble = new Instrument[1];
		ensemble[0] = new SimpleSampleInst(fileName, FRQ[C4]);
		// set start time in file
		double st = 0.0;
		// the length of each note (audio segment)
		double dur = 0.1;
		// the start panning position (left)
		double pan = 0.0;
		// sets the number of steps between panning changes
		int panSpeed = 8;
		// iterate through many notes
		for (int i=0; i<100; i++) {
			// check about updating panning position
```

TheGrid7

Functionally, *TheGrid7* is identical to *TheGrid6*. The code has been divided into two classes. The main method and GUI code is in `TheGrid7.java` and the musical logic is in a new class called `GridMusic7.java`.

```
private GridMusic7 musicObject;
...
musicObject = new GridMusic7();
```

The program starts by running *`TheGrid7`* which uses the line above to declare an instance of `GridMusic7` which it calls `musicObject`. In the constructor of *`TheGrid7`*, `musicObject` is instantiated. When musical changes are required, for example from clicks on the `JCheckBoxes`, *`TheGrid7`* makes appropriate calls to `musicObject`.

```
public void setHatsPitch(int arrayIndex, int pitch) {
        hats.getNote(arrayIndex).setPitch(pitch);
    }
```

To facilitate communication, *GridMusic7* provides a number of assessor methods for `TheGrid7` to call, such as the `setHatsPitch()` method above.

By now we know that to make changes or additions to the visual appearance of the application we work in the `TheGrid7.java` file, and to make changes to the music aspects of the application we work in the `GridMusic7.java` file.

TheGrid with audio

As we have been focusing on audio playback in this chapter, it is natural that our next version of the *TheGrid* application will add features for audio playback and saving of the music as an audio file. Also, in the chapter, each drum part will get it's own volume control so we can set the mix levels. There are a few other minor changes to the application, including setting the default tempo to 100 bpm.

TheGrid8

In *TheGrid8* we add two new buttons, one for playing the music with synthesized instruments, the other for rendering that output to an audio file.

```
private JButton saveBtn, playBtn, audioSavebtn, audioPlayBtn;
```

any instrument allocated to a part. You may notice that the drums part is still set to instrument 26, but this is now redundant.

```
public void audioPlay() {
    Instrument hats = new TR808HatsInst();
    hats.setOutput(Instrument.REALTIME);
    Instrument snare = new TR808SnareInst();
    snare.setOutput(Instrument.REALTIME);
    Instrument kick = new TR808KickInst();
    kick.setOutput(Instrument.REALTIME);
    Instrument[] kit = {hats, snare, kick};
    Part temp = drums.copy();
    Mod.repeat(temp, this.repeats);
    Score score = new Score(temp.getTempo());
    score.addPart(temp);
    Play.audio(score, kit);
}
```

The real time playback of the music is handled by the `audioPlay()` method. Instruments are declared for each drum and set to operate as real time instruments. By default, jMusic Instruments are usually set up for rendering. The `TR808HatsInst` and those for snare and kick have been specially created for this example. In later chapters we'll examine how to create your own jMusic instruments. The instruments use white noise and sine wave oscillators to create synthetic drum sounds in the style of the famous Roland *TR808* drum machine. The instruments are added to an instrument array called kit. The order of their addition is important, because it corresponds to the instrument numbers assigned to each phrase above.

Just as for MIDI playback, we first create a copy of the drums part so as not to effect the original with any changes, and apply the required number of repeats to the pattern with a method from the `Mod` class. A Score, called score, is created and assigned the tempo from the part – as chosen by the user. A copy of the drums part, called `temp`, which holds all the phrases and notes is added to the score. Finally, the score and instrument array are passed as arguments to jMusic's `Play.audio()` method, which plays the music.

```
public void saveAudioFile(String fileName) {
    Instrument hats = new TR808HatsInst(44100);
    Instrument snare = new TR808SnareInst(44100);
    Instrument kick = new TR808KickInst(44100);
    Instrument[] kit = {hats, snare, kick};
    Part temp = drums.copy();
```

```
        drums.addPhrase(hats);
        drums.addPhrase(snare);
        drums.addPhrase(kick);
    }
    /**
    * Create the initial state of the music by filling arrays
    * with notes and rests.
    */
    public void compose() {
         int[] hatsHits =
            {REST, REST, FS2, REST, REST, REST, FS2, REST, REST, REST, FS2,
                REST, REST, REST, FS2, REST,};
        hats.addNoteList(hatsHits, SIXTEENTH_NOTE, 127);
        int[] snareHits =
            {REST, REST, REST, REST, D2, REST, REST, REST, REST, REST, REST,
                 REST, D2, REST, D2, REST};
        snare.addNoteList(snareHits, SIXTEENTH_NOTE, 127);
        int[] kickHits =
            {C2, REST, REST, REST, REST, REST, REST, C2, C2, REST, REST, REST,
                 REST, REST, REST, REST};
        kick.addNoteList(kickHits, SIXTEENTH_NOTE, 127);
    }
```

A series of accessor methods are declared. Many of these specify and interrogate the pitch of notes within the arrays, particularly to allow or check for changes between sounding notes and rests (notes with a pitch of -1 in jMusic). The last couple of methods set the number of repeats and tempo.

```
    public void setHatsPitch(int arrayIndex, int pitch) {
        hats.getNote(arrayIndex).setPitch(pitch);
    }
    public void setSnarePitch(int arrayIndex, int pitch) {
        snare.getNote(arrayIndex).setPitch(pitch);
    }
    public void setKickPitch(int arrayIndex, int pitch) {
        kick.getNote(arrayIndex).setPitch(pitch);
    }
    public boolean getHatsState(int arrayIndex) {
        return (!hats.getNote(arrayIndex).isRest());
    }
    public boolean getSnareState(int arrayIndex) {
```

```
        snare.setOutput(Instrument.REALTIME);
        Instrument kick = new SampleInst("Kick.au");
        kick.setOutput(Instrument.REALTIME);
        Instrument[] kit = {hats, snare, kick};
        Part temp = drums.copy();
        Mod.repeat(temp, this.repeats);
        Play.audio(temp, kit);
    }
```

Finally, there are two methods that support the rendering of a music as audio files. These are very similar to the playback methods but use the `Write.au()` methods to send the audio to a file rather than the audio output. One method is supplied to support synthesized sounds and the other to support sample-based sounds.

```
    public void saveAudioFile(String fileName) {
        Instrument hats = new TR808HatsInst(44100);
        Instrument snare = new TR808SnareInst(44100);
        Instrument kick = new TR808KickInst(44100);
        Instrument[] kit = {hats, snare, kick};
        Part temp = drums.copy();
        Mod.repeat(temp, this.repeats);
        Write.au(temp, fileName, kit);
    }
    public void saveSampleFile(String fileName) {
        Instrument hats = new SampleInst("Hats.au");
        Instrument snare = new SampleInst("Snare.au");
        Instrument kick = new SampleInst("Kick.au");
        Instrument[] kit = {hats, snare, kick};
        Part temp = drums.copy();
        Mod.repeat(temp, this.repeats);
        Write.au(temp, fileName, kit);
    }
```

The *SampleInst* class takes an audio file as input and processes it in response to note events by extracting the pitch and dynamic information from the note and applying them to the sample. As well, a simple envelope is provided to avoid clicks at the start and end of sample playback. Below is the full *SampleInst* code.

```
import jm.audio.io.*;
import jm.audio.Instrument;
import jm.audio.synth.*;
```

9: Following Rules

Control over musical processes at the note-attribute level has been common for many centuries but was taken to extremes in the serialist compositions of Schoenberg, Webern, Boulez and others in the first half of the 20th century. A more accessible form of serialism developed is the later 20th century, notably in the music of Arvo Pärt, where less extreme ranges of music elements were used, such as diatonic harmonies replacing atonal. Rule-based music is a natural candidate for computer-based musical practices given that all music algorithms are codification of rules of some sort. Composers such as Iannis Xenakis and Gottfried Michael Koenig were pioneers in algorithmic computer music, that the electroacoustic music community developed and continues to explore. In this chapter we will investigate some common and historically relevant rule-based music processes.

Music From Text

A simple rule to apply is for the musical structure to follow some existing data. Because langauges have some inherent structures, well described by Noam Chomsky and others, it is reasonable to expect that music based on text might reflect some of this structure. The sonification of text can use simple or complex mappings between the text and sound.

In the *RuleMusic* example we explore a simple mapping from the numerical values of text characters in the computer to MIDI note numbers. So the rule here is that each letter in the text triggers a note of a particular pitch. To make this rule clear in this example, all other variables including rhythmValue, dynamic, instrument, and so on, are kept constant.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;

public class RuleMusic implements JMC {
    public static void main(String[] args) {
        new RuleMusic(args[0]);
    }

    public RuleMusic(String sentence) {
```

When running the program, a number of words are added as arguments to the `java` command. These words are used as the input text (words) on which the music is based. For example, using a command line you would type something like:

```
java GuidoWordMusic "this is some text to make music out of"
```

Here is the code for the *GuidoWordMusic* program.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;

public class GuidoWordMusic implements JMC {
    Phrase phr = new Phrase();
    int[] pitchesA = {b3,g4,e5};
    int[] pitchesE = {c4,a4,f5};
    int[] pitchesI = {d4,b4,g5};
    int[] pitchesO = {e4,c5,a5};
    int[] pitchesU = {f4,d5,c6};
    int consonantCount = 1;
    int prevPitch = 72;

    public static void main(String[] args) {
        new GuidoWordMusic(args);
    }

    public GuidoWordMusic(String[] sentence) {
        for (int i = 0; i<sentence.length; i++) {
            for (int j=0; j < sentence[i].length(); j++) {
                System.out.println(sentence[i].charAt(j));
                if (sentence[i].charAt(j) == 'a') {
                    System.out.println("** A");
                    chooseNote(pitchesA);
                }
                if (sentence[i].charAt(j) == 'e') {
                    System.out.println("** E");
                    chooseNote(pitchesE);
                }
                if (sentence[i].charAt(j) == 'i') {
                    System.out.println("** I");
                    chooseNote(pitchesI);
```

brieves), however this was simplified in our implementation as the resulting phrases tended to be somewhat disjointed. The algorithm takes some account of voice leading, and could even be developed to produce polyphonic phrases. However, even in its current form, the melody generation program does provide a good example of the possibilities of algorithmically generating music - even with only the simplest of rules.

Serialism

Serialist music is music composed with strict adherence to rules. In the early 20th century serialism was closely tied to atonal music, that is, music that plays all 12 chromic pitches with equal frequency so as not to create the sense that there is any tonal centre. To achieve this the method developed was to create note rows that included all 12 chromatic pitches then to compose music that used these rows. To provide some variety a number of rules for modifying the rows were developed, these have been included in jMusic's `Mod` class. The `Mod` (modification) methods apply rule-based processes to jMusic data classes. For example, the `Mod.rotate()` method takes a phrase and moves the first note in the phrase to the end.

Arnold Schoenberg and other Atonal serialists composed music by fairly strict adherence to rules of note order. Some of the modifications they regularly employed to vary the tone row (the phrase with all 12 notes of the octave in it) that was the raw material for their works, included playing the row backwards (retrograde) flipping the pitches relative to their intervallic distance from first note in the row (inversion) and the combination of these techniques (retrograde inversion). There are `Mod` methods for each of these processes with the obvious method names, as follows.

```
Mod.retrograde()
Mod.inversion()
Mod.retrogradeInversion()
```

Programming Practice

Write a program that plays a melody backwards and inverted using the `Mod` class. Use this program as a the basis of another that composes a multi-part composition using a tone row as its basic material and including the simple variations on the row described here.

Music Moment

The composer Arvo Pärt was trained in the atonal serialist tradition and wrote for many years in this style. In the 1970s he developed a new tonal serial style he called Tintinnabuli, a process that featured small intervallic

The main method calls the constructor which, in turn, calls each of the private methods. The first of these creates the melody and the next few create accompanying parts, while the final method collates, writes and plays the music.

```
public static void main(String[] args){
    new Arvoish();
}

public Arvoish() {
    createMelody();
    createAccompaniment1();
    createAccompaniment2();
    createAccompaniment3();
    saveAndPlay();
}
```

The melody is created as a random walk, starting on pitch 72 and wandering up to one octave from that starting point. Pitches are restricted to an Aeolian mode and rhythm values are randomized but triadic notes are reinforced by having longer note lengths.

```
private void createMelody() {
    int phraseLength = 47;
    //start on the C above middle C
    int pitch = 72;
    int dynamic = 100;
    //Create the melody as a random walk
    int offset = 0;
    int previousOffset = 0;
    for (int i=0;i<phraseLength;){
        // next note within a couple of octaves and
        // don't allow three repeated notes.
        if (pitch < 84 && pitch > 60) {
            offset = (int)(Math.random() * 8 - 4);
        } else {
            if (pitch <= 60) offset =(int)(Math.random() * 6);
            if (pitch >= 84) offset =(int)(Math.random()*6-6);
        }
        //check that it is a note in the mode
        Note note = new Note(pitch + offset,
            rhythms[(int)(Math.random()*rhythms.length)], dynamic);
        if (note.isScale(mode) && (offset != 0 || previousOffset != 0)) {
```

```
        }
        voice3.addNote(note);
    }
}
```

The final accompaniment section plays some tonic notes when long tonic or dominant notes occur in the melody.

```
    private void createAccompaniment3() {
        for (int i=0;i<voice1.size(); i++) {
            Note nn = voice1.getNote(i);
            if ((nn.getPitch()%12 == 9 || nn.getPitch()%12 == 4)
                    && nn.getRhythmValue() >= 4.0) {
                Rest r = new Rest(nn.getRhythmValue() * 0.5);
                voice4.addRest(r);
                Note note = new Note(A2, nn.getRhythmValue() * 0.5,
                    (int)(nn.getDynamic() * 0.5));
                voice4.addNote(note);
            } else voice4.addRest(new Rest(nn.getRhythmValue()));
        }
    }
```

Finally the score is assembled, displayed, saved and played.

```
    private void saveAndPlay() {
        //add a rest to allow reverb to end
        Rest end = new Rest(8.0);
        voice1.addRest(new Rest(8.0));
        // add the phrase to an instrument and that to a score
        melody.addPhrase(voice1);
        theWork.addPart(melody);
        harmony.addPhrase(voice2);
        theWork.addPart(harmony);
        bass.addPhrase(voice3);
        theWork.addPart(bass);
        sub.addPhrase(voice4);
        theWork.addPart(sub);
        // create a MIDI file of the score
        Write.midi(theWork, "Arvoish.mid");
        // display score
        View.show(theWork);
        // playback
```

The difference between these is that do always executes the statement at least once before the conditional check, whereas while checks the condition first and so may never execute the statement. Here's an example from one of the book's recent programs.

```
while (!note.isScale(triadic) && note.getPitch()%12 != nn.getPitch()%12) {
    note.setPitch(note.getPitch() - 1);
}
```

switch statements

Switch statements are useful when there is a condition check that may have several outcomes that require different action taken in each case. For example, when checking someone's age you may want to provide different responses if they are a baby, a child, a teenager, a young adult, middle-aged person, or older person. The switch statement takes the form:

```
switch ( check ) {
    case outcome1:
        statements
        break;
    case outcome2:
        statements
        break;
    default:
        statements
        break;
```

TheGrid: Controlling Parameters

This version of TheGrid application adds a volume slider and other visual appearance improvements.

TheGrid10

In *TheGrid10*, we'll add some functionality by way of a volume slider for each drum part. Sliders are a new component type for this application so there are a few details about their implementation that we need to review.

```
import javax.swing.event.*;

public class TheGrid10 implements JMC, ItemListener, ActionListener,
                                  ChangeListener {
```

between the side of the frame and the components, this whole `windowPanel` is placed in the "Center" of the frame and empty `JPanels` are added to the "North", "South", "East" and "West' sides of frame's layout.

```
public void stateChanged(ChangeEvent e) {
    if (e.getSource() == hatVolumeSlider)
        musicObject.setHatVolume(hatVolumeSlider.getValue());
    if (e.getSource() == snareVolumeSlider)
        musicObject.setSnareVolume(snareVolumeSlider.getValue());
    if (e.getSource() == kickVolumeSlider)
        musicObject.setKickVolume(kickVolumeSlider.getValue());
}
```

The `ChangeListener` interface requires the implementation the `stateChanged()` method which is called whenever a registered slider is moved. Changes in each slider result in setting of the volume in the *GridMusic9* class, called `musicObject`. The current value of the slider is retrieved using `hatVolumeSlider.getValue()`, or similar, and passed as an argument to the `setHatVolume()`, or similar, method.

```
public void setHatVolume(int val) {
    hats.setVolume(val);
}
```

In the `GridMusic10` class the `setHatVolume()` method is implemented along with those for snare and kick drums. These set methods are simple accessor methods that pass the new volume value to the `setVolume()` method of the phrase. The volume changes are applied just prior to playback, along with the other GUI settings for repeat and tempo in the `updatedPart()` method.

```
private Part updatedPart() {
    Part temp = drums.copy();
    Mod.repeat(temp, this.repeats);
    for (int i=0; i<temp.size();i++) {
        Phrase phr = temp.getPhrase(i);
        phr.setDynamic(phr.getVolume());
    }
    return temp;
}
```

The code for these changes is consolidated in the `updatedPart()` method so that it is only written once and used many times. It is used to create a modified drum part by each of the four

```
public static void main(String[] args) {
    new TheGrid10();
}
/**
* Constructor. Sets up the parameters.
*/
public TheGrid10() {
    // set look and feel
    String local = javax.swing.UIManager.getSystemLookAndFeelClassName();
    String metal = javax.swing.UIManager.
          getCrossPlatformLookAndFeelClassName();
    try {
        UIManager.setLookAndFeel(local); // use local or metal for OS X
    } catch (Exception e) {}
    // set graphic data
    hatTicks = new JCheckBox[16];
    snareTicks = new JCheckBox[16];
    kickTicks = new JCheckBox[16];
    hatVolumeSlider = new JSlider(JSlider.VERTICAL, 0, 127, 85);
    snareVolumeSlider = new JSlider(JSlider.VERTICAL, 0, 127, 85);
    kickVolumeSlider = new JSlider(JSlider.VERTICAL, 0, 127, 85);
    // call other methods
    musicObject = new GridMusic10();
    musicObject.compose();
    makeInterface();
}
/**
* GUI code
*/
private void makeInterface() {
    window = new JFrame("Drum Pattern");
    window.getContentPane().setLayout(new BorderLayout());
    JPanel windowPanel = new JPanel(new BorderLayout());
    // numbers
    JPanel numbGrid = new JPanel(new GridLayout(1,18));
    numbGrid.add(new JLabel(""")); // fill label
    numbGrid.add(new JLabel("Vol", SwingConstants.CENTER));
    for (int i=1; i<17; i++) {
        numbGrid.add(new JLabel(""+i, SwingConstants.CENTER));
    }
```

```
}
// Makes the GUI button and pop controller area
private JPanel addControlPanel() {
     JPanel btnPanel = new JPanel(); //new GridLayout(2,1)
    //  buttons
    JPanel midiPanel = new JPanel(new GridLayout(2,1));
    playBtn = new JButton("Play MIDI");
    playBtn.addActionListener(this);
    midiPanel.add(playBtn);
    saveBtn = new JButton("Save MIDI");
    saveBtn.addActionListener(this);
    midiPanel.add(saveBtn);
    btnPanel.add(midiPanel);
    JPanel audioPanel = new JPanel(new GridLayout(2,1));
    audioPlayBtn = new JButton("Play Audio");
    audioPlayBtn.addActionListener(this);
    audioPanel.add(audioPlayBtn);
    audioSavebtn = new JButton("Save Audio");
    audioSavebtn.addActionListener(this);
    audioPanel.add(audioSavebtn);
    btnPanel.add(audioPanel);
    JPanel samplePanel = new JPanel(new GridLayout(2,1));
    samplePlayBtn = new JButton("Play Samples");
    samplePlayBtn.addActionListener(this);
    samplePanel.add(samplePlayBtn);
    sampleSaveBtn = new JButton("Save Samples");
    sampleSaveBtn.addActionListener(this);
    samplePanel.add(sampleSaveBtn);
    btnPanel.add(samplePanel);
    // repeats
    repeatMenu = new JComboBox();
    repeatMenu.addActionListener(this);
    repeatMenu.addItem("1");
    repeatMenu.addItem("2");
    repeatMenu.addItem("4");
    repeatMenu.addItem("8");
    repeatMenu.addItem("16");
    btnPanel.add(new JLabel("Repeats = "));
    btnPanel.add(repeatMenu);
    // tempo
```

```
/**
* Handle the button clicks
*/
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == playBtn) {
        musicObject.play();
    }
    if (e.getSource() == audioPlayBtn) {
        musicObject.audioPlay();
    }
    if (e.getSource() == samplePlayBtn) {
        musicObject.samplePlay();
    }
    if (e.getSource() == saveBtn) {
        musicObject.saveFile("DrumPattern10.mid");
    }
    if (e.getSource() == audioSavebtn) {
        musicObject.saveAudioFile("DrumPattern10.au");
    }
    if (e.getSource() == sampleSaveBtn) {
        musicObject.saveSampleFile("DrumSamplePattern10.au");
    }
    if (e.getSource() == tempoMenu) {
        musicObject.setTempo(new Double((String)tempoMenu.
              getSelectedItem()).doubleValue());
    }
    if (e.getSource() == repeatMenu) {
        musicObject.repeatMusic(Integer.parseInt(
             (String)repeatMenu.getSelectedItem()));
    }
}
/**
* Handle slider movements
*/
public void stateChanged(ChangeEvent e) {
    if (e.getSource() == hatVolumeSlider)
         musicObject.setHatVolume(hatVolumeSlider.getValue());
    if (e.getSource() == snareVolumeSlider)
         musicObject.setSnareVolume(snareVolumeSlider.getValue());
    if (e.getSource() == kickVolumeSlider)
```

```
                REST, D2, REST, D2, REST};
    snare.addNoteList(snareHits, SIXTEENTH_NOTE);
    int[] kickHits =
        {C2, REST, REST, REST, REST, REST, REST, C2, C2, REST, REST, REST,
                REST, REST, REST, REST};
    kick.addNoteList(kickHits, SIXTEENTH_NOTE);
}
/**
* Specify the pitch (including as a REST) of a note in the hats array.
* @param arrayIndex The note to be changed.
* @param pitch The value to change it's pitch to.
*/
public void setHatsPitch(int arrayIndex, int pitch) {
    hats.getNote(arrayIndex).setPitch(pitch);
}
/**
* Specify the pitch (including as a REST) of a note in the snare array.
* @param arrayIndex The note to be changed.
* @param pitch The value to change it's pitch to.
*/
public void setSnarePitch(int arrayIndex, int pitch) {
    snare.getNote(arrayIndex).setPitch(pitch);
}
/**
* Specify the pitch (including as a REST) of a note in the kick array.
* @param arrayIndex The note to be changed.
* @param pitch The value to change it's pitch to.
*/
public void setKickPitch(int arrayIndex, int pitch) {
    kick.getNote(arrayIndex).setPitch(pitch);
}
/**
* Pass back weather or not the specified hat note is a rest or not.
* @return boolean True if it is a pitched note, false if it is a rest.
*/
public boolean getHatsState(int arrayIndex) {
    return (!hats.getNote(arrayIndex).isRest());
}
/**
* Pass back weather or not the specified snare note is a rest or not.
```

```
public void setSnareVolume(int val) {
    snare.setVolume(val);
}
 /**
* Specify the new volume for the kick drum
*/
public void setKickVolume(int val) {
    kick.setVolume(val);
}
/**
* Return a part with the necessary modification ready for playback.
 */
private Part updatedPart() {
    Part temp = drums.copy();
    Mod.repeat(temp, this.repeats);
    // apply volume to note dynamic for each phrase
    for (int i=0; i<temp.size();i++) {
        Phrase phr = temp.getPhrase(i);
        phr.setDynamic(phr.getVolume());
    }
    return temp;
}
/**
 * Playback the music using jMusic audio.
 */
public void audioPlay() {
    // set up instruments
    Instrument hats = new TR808HatsInst();
    hats.setOutput(Instrument.REALTIME);
    Instrument snare = new TR808SnareInst();
    snare.setOutput(Instrument.REALTIME);
    Instrument kick = new TR808KickInst();
    kick.setOutput(Instrument.REALTIME);
    Instrument[] kit = {hats, snare, kick};
    Play.audio(updatedPart(), kit);
}
 /**
 * Save the music as an au file.
 * @param fileName The name of the  file, usually ending in .au
 */
```

10: Uncertainty

An important consideration when composing music is achieving an appropriate balance between predictability and surprise. Too much predictability leads to boredom, while too much uncertainly leads to confusion. When writing music with a computer, we can pass some of the responsibility for musical choices to the computer. The computer might fill in detail, leaving us free to think about higher level structure, or the computer might be given choices between equally acceptable alternatives. But, how can a computer make musical choices? Essentially it cannot, however, the computer can select between options based on weighted probability, in the simplest case as pseudo-randomness. The result is that the outcome is uncertain to some degree, and it is up to the composer/programmer to decide the nature of the uncertainties and their boundaries.

Aleatoric and Stochastic Music

Music that relies on chance is referred to as aleatoric music. Composers have been interested in the use of chance for some time. A famous example is Mozart's musical dice game Musikalisches Würfelspiel, mentioned previously, where numbered fragments of music are recombined according to the outcome of rolling the dice. But, more specifically, aleatoric music relies on chance in the execution, rather than composition; although at times these are one and the same. John Cage was a twentieth century American composer noted for promoting aleatoric music. In his works chance elements included selecting cards from a shuffled deck, calling phone numbers at random, rolling dice, and relying on the chance actions of an unsuspecting audience.

Music that uses mathematical probability is often referred to as stochastic music. There are historical examples of stochastic music where the calculations were done manually, but since the availability of the computer in the middle of last century the capacity to create stochastic music has increased dramatically. Iannis Xenakis was a composer very interested in stochastic music. He wrote pieces with stochastic elements both using manual calculations and with computers. These works were for both acoustic instruments and for computer generated output, and focused on probabilistic control of almost every aspect of a musical work, from the macro structure to the micro structure. Aspects determined this way included the overall duration and structure of a work, individual note attributes, and sound generation and spatial positioning.

`Math.random()`, produces a linear distribution of its possible outcomes, any double value from 0.0 to less than 1.0. So we will use it when we want to simulate the roll of a dice, spin of the roulette wheel, or unpredictable musical accent.

The *CloudX* program uses randomness in the selection of note pitch, rhythmValue, dynamic, panning position and beat location. In jMusic, position in time is an attribute of Phrases, so in order to position each note at a random location they are each added to a separate phrase. The note lengths are very short and the sound specified is a breathy chiff, so the result is a cluster of short sound bursts that create a cloud-like texture. This program is inspired by one of Xenakis' stochastic compositions, *Concert PH*, which has similar sonic properties.

```
double cloudDensity = 0.1;
Phrase phr = new Phrase()
for (int i = 0; i < 1500; i++) {
    int pitch = (int)(Math.random()*127);
    double rv = Math.random() * cloudDensity;
    int dyn = (int)(Math.random()*127);
    Note n = new Note(pitch, rv, dyn);
    n.setPan(Math.random());
    phr.addNote(n);
}
Mod.fadeOut(phr, 20);
Part p = new Part("Cloud", BREATHNOISE, 0);
p.addPhrase(phr);
Score s = new Score(p);
View.show(s);
```

The `Math.random()` method provides a linear distribution of values. That is, each result has the same probability. This can be seen, and heard, in the *CloudX* program where the notes are evenly distributed across the entire range of pitch values. In many musical situations a linear distribution is not always what we want. Rather, we often want some events to occur more often than others, for example, the occurrence of tonic and dominant notes in a diatonic melody. We will now explore processes that create uneven, or weighted, probabilities.

Gaussian distribution

The Gaussian distribution provides an uneven randomness where values are more likely around a centre value and become less likely to occur the further away from that value. A diagram showing a linear distribution is a flat line, whereas the Gaussian distribution looks like a hill or a hump, which is why it is often called the 'bell curve'. The Gaussian distribution is named after the mathematician

```
        if (pitch < 0) pitch = 0;
        if (pitch > 127) pitch= 127;
    }
```

The similar *GuassianWalk* program uses the `Prob.gaussianPitch()` method in place of the `Math.random()` method of the *RandomWalk* program.

```
pitch = Prob.gaussianPitch(pitch, deviation);
```

The melodies that result tend to have a greater number of small intervals but also feature occasional large leaps greater than an octave. Comparing the melodies from the *RandomWalk* and *GuassianWalk* programs provides a good comparison of the linear and Gaussian distributions in a music context.

There are numerous probability distributions that can be applied to music, each has its own characteristics. Some common ones include binomial, poisson, and Cauchy, however, in most musical situations linear and Gaussian options should provide a sufficient degree of control. For a detailed exploration of the musical applications of these probability distributions see the book, *Formalized Music: Thought and mathematics in Music*, by Iannis Xenakis (1971).

Markov process

Andrei Andreevich Markov was a Russian mathematician who discovered a technique for recording the dependencies between items in series. This technique has been named the Markov process in honor of its inventor. At the heart of the Markov process is the creation of a Markov Chain, a series of values that follow one another with a certain degree of probability. Markov Chains have a long and successful history in computer music research, and for good reason. Markov Chains provide an effective mechanism for creating and using stochastic matrices in musically satisfying ways.

Markov Chains are created by analyzing the chance of any given value going directly to any other value. For example, within a melody, the probability that a note of pitch C4 will be followed by a note of pitch G4. Because C4 might also be followed by D4 with a different likelihood, a Markov Chain is derived by creating first a Markov Matrix of all possibilities. We can calculate the probabilities for the matrix by checking a particular pitch every time it occurs, seeing which notes do follow it, and counting how many times it is followed by each one. If we do this for every pitch value we build a stochastic matrix which resembles the composition that it is based on.

Let's take a look at an example of a melodic series (a melody).

```
{64,62,60,62,64,64,64,62,62,62,64,67,67,64,62,60,62,64,64,64,64,62,64,62,60}
```

on, to create a melody of any length. This new melody will have a similar character to the original melody used to create the matrix.

The *BasicMarkov* program implements the Markov process we've been examining. You can run it several times to hear how the musical results are different each time, but maintain a stylistic consistency.

This matrix defines links between one note and the next. When constructing the matrix we looked a single step ahead in the melody, so we say the matrix is a single state matrix and that it has a depth of one. If Markov matrices were limited to a single state per seed then their musical usefulness would be limited. Luckily, there are no such limit and seeds can in fact be any length you like up to the length of the original melody. Typically, a depth of more that four is unnecessary.

To produce a Markov matrix of a depth of more than one we use seeds of more than one element. For example, we analyze the original melody to see when C4 is followed by G4 how many times the next note is A4? We calculate a matrix with a depth of 2 in the same way we calculate a matrix with a depth of 1. The only difference is that all our calculations are based upon any two numbers being followed by another number. Only pairs of pitches that are in the original melody are included in the matrix. Although our seed is now two pitches in a row, the output is always one pitch. This is true regardless of the Markov depth. Our primary matrix now looks like this.

	60	62	64	67
64,62	0.6	0.4	0.0	0.0
62.60	0.0	1.0	0.0	0.0
60,62	0.0	0.0	1.0	0.0
62,64	0.0	0.25	0.5	0.25
64,64	0.0	0.4	0.6	0.0
62,62	0.0	0.3333	0.6666	0.0
67,67	0.0	0.0	1.0	0.0
67,64	0.0	1.0	0.0	0.0
64,67	0.0	0.0	0.0	1.0

Changes from the previous matrix include that the far left hand column now contains two numbers (prior states) as the seed rather than one, there are more rows than there were previously, and there are more 1.0 (100%) weightings than there were before. The last two points seem obvious, but they point to the most important feature of Markov chains; the ability to maintain local context and provide for more discriminatory, therefore musically coherent, output. Where there is a 1.0 probability it is a forced move for the melody, there is no other choice. This the resulting three

```
*/

public class MarkovTest extends HelperGUI implements JMC{

        public static void main(String[] args){
            new MarkovTest();
        }

        public MarkovTest() {
            setVariableA(2, "Seed depth");
        }

        public Score compose() {
            int[] pitches = {64,62,60,62,64,64,64,62,62,62,64,67,67,64,62,
                60,62,64,64,64,64,62,62,64,62,60};
            double[] rhythms = {QN,QN,QN,QN,QN,QN,HN,QN,QN,HN,QN,QN,HN,QN,
                QN,QN,QN,QN,QN,QN,QN,QN,QN,QN,QN,HN};
            int[] dynamics = {80,70,60,70,80,80,80,70,70,70,80,100,100,80,
                70,60,70,80,80,80,80,70,70,80,70,60};
            Phrase marysLamb = new Phrase();
            marysLamb.addNoteList(pitches, rhythms, dynamics);
            int markovDepth = variableA;
            PhraseMatrix matrix = new PhraseMatrix(marysLamb, markovDepth);
            Phrase myLamb = matrix.generate(true, false, false, 20);
            Score scr = new Score("MarysLamb", 120);
            Part part = new Part();
            part.addPhrase(myLamb);
            scr.addPart(part);
            return scr;
        }
}
```

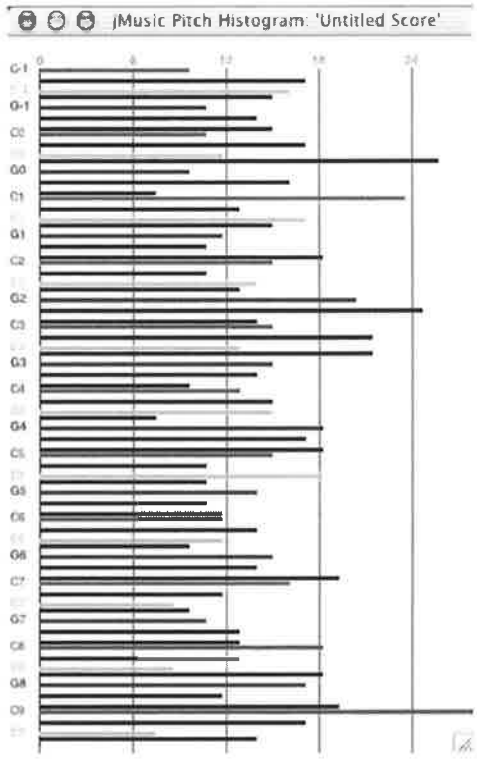


Figure 10.1. Histogram Viewer

To instantiate a histogram viewer use the static method of the jMusic `View` class `View.histogram(Score s);` passing a music data object, typically a score, as an argument. The display shows all the possible values down the left hand side, with bars from left to right indicating the number of each value present in the score, for example the number of each note pitch from 0 to 127. The color-coding is to aid readability. All similar values are the same color; for example, all C pitches are yellow. The histogram menu allows you to s switch between viewing the pitch, rhythm, dynamic and pan values in the histogram. There are options to open a MIDI or jMusic XML file for viewing, and so to save the analysis data as a tab delimited text file, for further viewing or treatment in another program such as a spread sheet, word processor, or data base.

A demonstration of the histogram can be seen with the *HistogramAnalysis* program. This program creates a score of notes with random pitch, rhythm, duration and pan values. Histograms of each are opened, then lastly a dialog box appears enabling you to select a MIDI file for analysis and display.

The `HelperGUI` interface has a `View.histogram()` button which brings up the histogram display showing an analysis of the current composition. Below is the code the HistogramAnalysis program.

from cycling through a wavetable undermines the unpredictability of pure noise. Therefore, the noise audio objects continually calculate random sample values as required.

The *WhiteNoiseNote* program creates an audio file of white noise from a single note score, and displays it in a wave viewer window. It uses the `NoiseInst` instrument that defaults to a monophonic white noise output. The `NoiseInst` instrument adds a percussive envelope to the sound which is clearly audible and visible in the wave viewer.

```
Instrument wave = new NoiseInst(44100);
```

The *AllNoiseTypes* program is similar but creates an audio file in each of the noise types supported by jMusic. It allows you to compare the sound and appearance of each of the noise types. It uses the `NoiseInst` constructor that accepts three arguments, the sample rate, the number of channels, and the noise type. The noise type argument is an integer but, for convenience, there are constants in the `jm.audio.synth.Noise` class for each of them.

```
WHITE_NOISE = 0
STEP_NOISE = 1
SMOOTH_NOISE = 2
BROWN_NOISE = 3
FRACTAL_NOISE = 4
GAUSSIAN_NOISE = 5
WALK_NOISE = 6
GENDYN_NOISE = 7;
```

Below is the instantiation of the *NoiseInst* instrument in the *AllNoiseTypes* program.

```
Instrument wave = new NoiseInst(44100, 1, noiseType);
```

The differences between each noise type derive from variations in the amplitude distribution and granularity of change – some noise type change value at each sample, others less frequently. To clearly see the differences in the wave viewer zoom in to the wave form to a 1:1 viewing ratio so that each sample can be seen. Below is the source code for the *AllNoiseTypes* example.

```
import jm.music.data.*;
import jm.JMC;
import jm.audio.*;
import jm.util.*;

/*
* This class shows each jMusic noise type
*/
```

range without having to scale and cast. The class includes a `nextGaussian()` method so the one instance can create linear and gaussian distributions as required.

The `Random` class only returns floats from the `nextGaussian()` method, so jMusic provides the `jm.music.tools.Prob` class as a wrapper for getting Gaussian values from the `Random` class to make it easier to get jMusic pitch, frequency, rhythmValue, and panning data in the appropriate ranges and types.

TheGrid: Random Velocities

Human performers play each note slightly differently giving a subtly and variation to the performance that we appreciate. Often computer performances can be overly mechanical which does not suite all musical styles. In this version of *TheGrid* application we will add randomness to the loudness of notes that introduces some human quality to the playback.

TheGrid11

One useful application of randomness is to vary the dynamic of each note to add 'feel' to the music. This is sometimes referred to as 'humanizing' the performance because it is impossible for a human performer to play each note exactly the same. To achieve a similar effect for computer playback a small deviation of the existing dynamic value is added to each note.

The code for *TheGrid11* will be identical to *TheGrid10*, and GridMusic11 will require a small change to *GridMusic10* where the following code is substituted for the `updatePart()` method.

```
private Part updatedPart() {
    Part temp = drums.copy();
    Mod.repeat(temp, this.repeats);
    // apply randomness to note dynamic for each phrase
    for (int i=0; i<temp.size();i++) {
        Phrase phr = temp.getPhrase(i);
        for (int j=0; j<phr.size(); j++) {
            Note n = phr.getNote(j);
            n.setDynamic(phr.getVolume() + (int)(Math.random() * 40) - 20);
        }
    }
    return temp;
}
```

11: Emergence

While all algorithmic compositional processes result in music of some kind or another, there can be difference between processes that directly specify a musical outcome and those that are more indirect. Emergent process achieve their results indirectly, the outcomes arise from the interaction between elements within the process and often the outcome is quite unexpected or difficult to predict precisely. The surprising outcomes from emergent processes are somewhat different to stochastic processes, even though chance elements are often used in emergent algorithms. The classic case of emergence is evolution. In nature, the rules of natural selection give rise to the myriad of life forms that inhabit the earth. The rules of evolution don't specify what those life forms should be like but they do describe the process by which they will evolve. The life forms emerge from the evolutionary process.

Other examples of emergent systems include the behavior of an ant colony, where the rules that govern the actions of an individual ant in its foraging and building seem quite unlike the well organized community of ants that somehow manage to build coherent social structures and survive repeated disruption by predators and natural disasters. The existence of human consciousness is often cited as an emergent entity. The rules of life encoded in human beings do not directly describe consciousness, rather they describe how the human body and in particular the human brain should be built and operate. Consciousness emerges as a result of the brain functioning according to this design. A final example of an emergent system is a human constructed one, traffic flow in busy cities. Viewed on a large scale the behavior of traffic is particularly complex, especially when it is disrupted by accidents, traffic lights, and vehicles traveling at different speeds. Traffic flow patterns as a system are difficult to predict based on the 'rules' that control individual vehicles. The larger-scale traffic patterns emerge from the interactions between vehicles.

There seems to be a lot of potential for interesting musical structures from emergent systems. One difficulty is that the emergent behavior is often difficult to predict based upon the rules of individual elements within the system, and therefore considerable trial and error, and at times some luck, are often required to produce successful emergent systems for music. In this chapter we will look at some musical applications of well understood emergent processes.

Let's examine the source code for the *Swarm* program:

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;
import jm.music.tools.Mod;

public class Swarm implements JMC {
    private Note[] notes = {new Note(60, 0.5), new Note(60, 0.5),
         new Note(60, 0.5), new Note(60, 0.5)};
    private Phrase[] phrases = new Phrase[notes.length];
    private Part pianoPart = new Part(PIANO, 0);

    public static void main(String[] args) {
        Swarm noteCluster = new Swarm();
    }

    private Swarm() {
        for (int i=0; i<phrases.length; i++ ) {
            phrases[i] = new Phrase(0.0);
            phrases[i].addNote(notes[i]);
        }
        for (int i=0; i<16; i++) {
            step();
            System.out.println("Completed step " + i);
        }
        for (int i=0; i<phrases.length; i++) {
            pianoPart.addPhrase(phrases[i]);
        }
        Mod.shake(pianoPart, 40);
        View.show(pianoPart);
        Play.midi(pianoPart);
    }
```

The constructor method (shown above) outlines the general structure of the *Swarm* program. First, an array of notes is created, each with a pitch of middle C (MIDI pitch 60). This array will be used to store and update notes for the phrases. Four phrases are instantiated, they will be filled with notes as the program proceeds. Next, the `step()` method is called sixteen times (but it could be called any number of times). At each iteration this method adds a note to each of the phrases with the pitches varying according to the flocking rules. Once created these phases are added to the `pianoPart` object. Finally, the part is parsed by jMusic's `Mod.shake()` method that adds

```
            if (i== noteIndex) continue;
            choiceScope = 2;
            while (inChord(currentPitch, i) || notInScale(currentPitch)) {
                if (currentPitch > averagePitch)
                    currentPitch -= (int)(Math.random() * choiceScope++);
                else currentPitch += (int)(Math.random() * choiceScope++);
            }
        }
        return currentPitch;
    }
```

The `nextStep()` method implements the cohesion and separation rules. The starting point for the next pitch value is a random walk step from the previous pitch. This starting point may be a jump above or below the previous pitch. The first while-loop reels in the pitch. It executes until the pitch is as close to the average as possible while still being a pitch in the C pentatonic scale. The next while-loop moves the pitch around if required so that it is not the same pitch as any other note sounding at the same time. The `choiceScope` variable controls the amount of pitch variation on each iteration of the while loop. It is incremented on each loop so that an ever increasing pitch range is explored for a solution. Without this increasing search space it is possible that the options were so limited that no solution would be found or that it was so wide that the cohesion rule would be negated. The `nextStep()` method is supported by these two private methods.

```
    // check if the pitch is the same as any other pitch this beat
    private boolean inChord(int currentPitch, int counter) {
        boolean result = false;
        for (int i=0; i<counter; i++) {
            if (currentPitch == notes[i].getPitch()) result = true;
        }
        return result;
    }

    // check if the pitch is not within the scale
    private boolean notInScale(int currentPitch) {
        int[] scale = JMC.PENTATONIC_SCALE;
        boolean result = true;
        for (int i=0; i<scale.length; i++) {
            if (currentPitch%12 == scale[i]) result = false;
        }
        return result;
    }
```

```
public class BassMutate1 implements JMC {
    public static void main(String[] args) {
        new BassMutate1();
    }

    public BassMutate1() {
        Part bassPart = new Part();
        Phrase bassLine = new Phrase(0.0);
        for (int i = 0; i<16; i++) {
            bassLine.addNote(new Note(48, 0.25));
        }
        bassPart.addPhrase(bassLine.copy());

        for (int i=1; i<8; i++){
            Note n = bassLine.getNote((int)(Math.random() *
                 bassLine.size()));
            n.setPitch((int)(Math.random() * 12 + 48));
            bassLine.setStartTime(4.0 * i);
            bassPart.addPhrase(bassLine.copy());
        }

        View.print(bassPart);
        Play.midi(bassPart);
    }
}
```

Below is a version of *BassMutate* that iterates through generations of phrase pairs using each pair as parents for the next generation. Both crossover and mutation techniques are applied to create new child phrases.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;

public class BassMutate2 implements JMC {
    Phrase bassLine1 = new Phrase(0.0);
    Phrase bassLine2 = new Phrase(0.0);
    int phraseSize = 16;
    int[] availablePitches = {36, 36, 40, 43, 45, 48, 48, 52, 55,
        REST, REST, REST};
```

```
        View.show(bassPart);
        Play.midi(bassPart);
    }
```

Once the iterations are complete, the bass part is displayed on screen and played back via the in-built JavaSound synthesizer.

```
    private Phrase createChild() {
        Phrase tempPhrase = new Phrase();
        // crossover
        int crossOverPoint = (int)(Math.random() * phraseSize);
        for (int j=0; j<crossOverPoint; j++) {
            tempPhrase.addNote(bassLine1.getNote(j).copy());
        }
        for (int j=crossOverPoint; j<phraseSize; j++) {
            tempPhrase.addNote(bassLine2.getNote(j).copy());
        }
        // mutate
        Note n = tempPhrase.getNote((int)(Math.random() * phraseSize));
        int pitch = availablePitches[(int)(Math.random()
            * availablePitches.length)];
        n.setPitch(pitch);

        return tempPhrase;
    }
}
```

This `createChild()` method creates a child phrase by crossover between the parent phrases and a simple mutation of one of the note pitches.

Programming Practice

Assign variables in the *BassMutation* program to the `HelperGUI` variables. For example, replace the tempo value in the Score declaration with `variableA * 2`, and the number of iterations with `variableB`. Try varying the pitch set to see what different results you get.

Cellular Automata

Cellular Automata (CA) is a process where cells in a grid change properties, typically their color, based on the properties of their neighboring cells. The way in which they change can be specified

```
import jm.util.*;
```

The program imports the jMusic constants, the data types, and the `Util` package that contains the `Write` class.

```
public class RhythmicAutomata implements JMC{
    int barLength = 16;
    int[] bar = new int[barLength];
    int[] tempBar = new int[barLength];
    Part part1 = new Part("Hats", 0, 9);
    Part part2 = new Part("Rim Shot", 0, 9);
    Part part3 = new Part("Claps", 0, 9);
    Score score = new Score("AlgoRhythm", 84);
```

The class is declared and so are a number of class variables. Notice that an array of `int`'s called `bar` is declared. This holds the once dimensional cell matrix. The `tempBar` array is used when we apply the rules to evolve the array one generation.

```
 public static void main(String[] args) {
    new RhythmicAutomata();
 }
```

The `main()` method creates an instance of this class. Next we build the music one part at a time, bar by bar. The constructor makes a couple of method calls to fill the array (see details below), then sets up a loop that generates 23 bars of music, printing the state of the array to the terminal each time through.

```
public RhythmicAutomata() {
    initialise();
    makeMusic(part1, 42);
    for (int i=0; i<23; i++) {
        nextBar();
        makeMusic(part1, 42);
        printBar();
    }
    initialise();
    makeMusic(part2, 37);
    for (int i=0; i<23; i++) {
        nextBar();
        makeMusic(part2, 37);
        printBar();
```

Applying the CA rules for one generation.

```
    private void nextBar() {
        int sum = 0;
        for (int i=0; i<barLength; i++) {
            for (int j=1; j<3; j++) {
                if (i-j > 0) sum += bar[i - j];
                if (i+j < barLength) sum += bar[i +j];
            }
            if (sum == 0) {
                if (bar[i] == 1) tempBar[i] = 0;
                else tempBar[i] = 1;
            }
            if (sum == 1) tempBar[i] = bar[i];
            if (sum == 2) tempBar[i] = bar[i];
            if (sum == 3) {
                if (bar[i] == 1) tempBar[i] = 0;
                else tempBar[i] = 1;
            }
            if (sum == 4) {
                tempBar[i] = 0;
                sum = 0;
            }
            bar = tempBar;
        }
    }
```

This is the guts of the program. In this method the rules are applied. First, the number of live (1) cells within the range up to three places either side of the current cell are counted. Depending upon its number of active neighbors a cell either lives or dies. If it has 1 or 2 neighbors it is unchanged, if it has 3 neighbors and is alive it changes its state, and if it has four neighbors it's getting too crowded and it dies. The `tempBar` array is used to store the values of the next iteration during this process, once completed its contents are transferred to the `bar` variable and become the current state.

Create one bar of music.

```
private void makeMusic(Part p, int pitch) {
    Phrase phr = new Phrase();
    for (int i=0; i<barLength; i++) {
        if (bar[i] == 0) phr.addNote(new Note (REST, SQ));
```

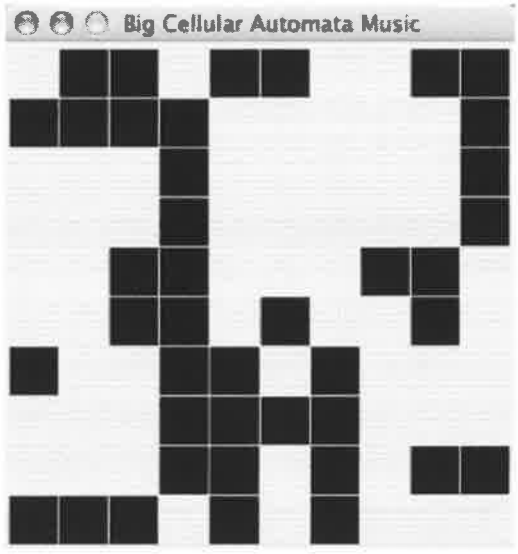


Figure 11.2. The visual output of the two dimensional CA example program.

Let's have a look at the source code.

```
import jm.JMC;
import jm.util.*;
import jm.music.data.*;
import jm.audio.*;
import jm.audio.synth.*;
import jm.music.tools.ca.*;
import jm.gui.ca.*;
```

We import the usual suspects for the jMusic data, and the `audio` classes for a simple synthesis output, and the `util` class for MIDI and AU file writing. To access the Cellular Automata classes in jMusic we need to import the `jm.music.tools.ca.*` and `jm.gui.ca.*` packages. Within these packages the `jm.music.tools.CellularAutomata` class is the engine room for this program, while the graphics are in the `jm.music.tools.ca.DrawCA` class.

```
public class AutomataMusic implements JMC{
    static boolean[][] cells = new boolean[10][10];
    int maxNotes = 100;
    Score score = new Score("The Cellular Drunk");
    Part piano = new Part("piano",0, 0);
    Phrase phr1 = new Phrase();
    Phrase phr2 = new Phrase();
    TriangleInst triInst = new TriangleInst(44100);
```

```
        }
        noteNumber = liveCellCounter + (127 - cells.length*cells[0].length);
        Note n = new Note(noteNumber, SQ);
        phr1.addNote(n);

        try {
            Thread.sleep(100);
            System.out.println("Iteration " + i + " of + maxNotes);
        } catch (Exception e) {}
        DCA.repaint();
        DCA2.repaint();
    }
```

In this code block most of the important work is specified. A loop is set up to repeat for the maximum number of notes we want in our score.

The CA object has it's `evolve()` method called, which applies the rules to the matrix one time.

The for-loop goes through each cell and checks it's state (on or off, true or false) and keeps a tally in the `liveCellCounter` variable.

The tally of live notes is mapped to pitch, via the variable called `noteNumber`. So that we don't get very low notes the pitch is offset. In this case the pitch offset needs to be 27 because the maximum possible on-cells = 100 and the highest allowable MIDI pitch is 127. The mathematics might look a bit complicated, but that is to allow for a wide range of possible cell matrix sizes. Of course the larger the matrix the wider the probable pitch range.

A pause is inserted into the programs thread (100 milliseconds) so that the graphics don't go too fast. This deliberately slows down the running of the program - the things we do to look pretty!

Having used the cellular automata matrix we get it to redraw its appearance in the last couple of lines of the code segment. Each of the `DrawCA` objects has its `repaint()` method called.

```
    piano.addPhrase(phr1);
    score.addPart(piano);

    Write.midi(score, "AutomataMusic.mid");
    View.show(score, 330, 10);

    Write.au(score, "AutomataMusic.au", triInst);
    System.out.println(CA.toString());
```

12: Making Waves

Sound in the physical sense can be understood as waves of air pressure moving from vibrating sources to our ears. The history of trying to understand sound waves goes back at least to Pythagoras and the ancient Greeks. It is also the case that larger scale musical structures, such as phrases, are also describes a curves. In this chapter, we will explore how wave functions, typically visualized as curves and contours, can be used as both a source for compositional ideas and as a way of rendering those compositions into audio files. But, before looking at mathematically generated curves and lines can translate into musical gestures we will look at how hand drawn curves can be turned into music.

Describing Musical Gestures as Curves

Sound waves, in the acoustic sense, are described in the computer as audio samples and digital wavetables. As we have discussed in previous chapters, this is the most fundamental way in which music, as digital audio, is described within the computer and we will examine this in again later in this chapter. However, there are other levels of abstraction at which we can describe music. In particular, at the note level where individual musical events are described and at the phrase, chord, chunk or building-block level where groups of notes are considered as a single entity. Waves, or more simply curves, can be used to describe paths for music at any of these levels of abstraction. For example, it is common to visualize samples as a describing a wave form, notes as a melodic curve, and phrase structures outlining a an arch form.

Curves, in mathematics, can have more formal description as functions, for example the sine function that describes a sine wave shape. In computer music we can combine the traditional thinking of musical structures as following contours with the mathematical descriptions of lines and cures as functions. In this chapter we will see examples of how musical gestures can be described in Java at different levels of abstraction and to varying degrees of description.

Sketch

The idea of mapping curves to musical parameters was used by the composer, instrument designer and architect Iannis Xenakis in designing his UPIC system in the late 1970s in France. The UPIC name is an acronym standing for Unité Polyagogique Information de CEMAMu. The system com-

The Sketch window is also available from the `HelperGUI` graphical interface has a `View.sketch()` button that displays the current composition and allows for free hand additions.

Generative Music

As we have seen in previous chapters, music can be composed in our imagination then transcribed into a code (Java, CPN or another), and it can be specified in broad outline and filled in by the algorithmic choices of the computer. In this section we see how music can arise as a sonic realisation of data; sonification. While all these practices are ways of generating music and sound, the term generative music usually implies that the music was created by a machine following an algorithm or heuristic principles. In a weak sense, all music made with Java could be said to be algorithmic and therefore generative, but in a strong sense of generative music the human control is secondary. A rich form of generative music is the sonification of mathematical functions and, since the current chapter is concerned with waveforms, sonification of wave functions are a good place to start.

Music from functions

A mathematical function describes a line or curve. These curves can be seen to describe a musical gesture, for example a melodic contour. A function can be mapped against any musical parameter, pitch in the case of melodic curve, or volume in the case of an amplitude envelope. The link between the function values and the musical parameters is entirely arbitrary; indeed a function may influence more than one parameter at a time. The function `x = 2 * y` produces a straight line that moves twice as far in the horizontal direction (x) for every step it takes in the vertical direction (y). A graph of the function is shown in figure 12.2.

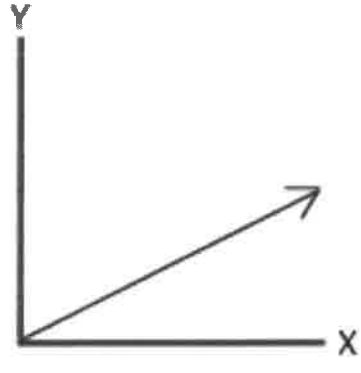


Figure 12.2: An analog trajectory.

SineMelody program extends `HelperGUI` the implementation of which has been shown previously, so only the `compose()` method is shown here.

```
public Score compose() {
    Score score = new Score("Sine melody", 130);
    Part p = new Part("Melody", MARIMBA, 0);
    Phrase phr = new Phrase();
    double density = 25.0;
    for (int i = 0; i < (2*3.14*density); i++) {
        int pitch = (int)(Math.sin(i/density) * 50 + 60);
        Note note = new Note(pitch, THIRTYSECOND_NOTE);
        phr.addNote(note);
    }
    p.addPhrase(phr);
    score.addPart(p);
    return score;
}
```

Extensions of this program could include changing the sine function for a different one. Some simple additive synthesis sonification is interesting to experiment with. Here, for example, is the code to sonify the result of adding this sine wave with another an octave above it (the fundamental with the first harmonic).

```
(int)((Math.sin(i/density) + Math.sin(i/density*2))/2*50+60);
```

The function shape in this example simply provides a melodic curve, and can be played back by any instrument.

Fractal music

Any mathematical function can be sonified in the way the sine function was above. Another popular set of functions to sonify are fractals. Fractal patterns are renowned for their self-similarity of structure. That is, patterns are one level are repeated at larger and smaller scales. The classic example in nature is the fern branch, where each sub-branch is a miniature version of the larger branch. Visualizations of fractal sets are also popular, depicted as colorful images of semi-organic appearance such as the Mandelbrot set. The inherent structure of fractal patterns has lead many to hope they may be useful in generating music. This turns out to be partially true, but like most mathematical music mappings, they work best when there are considerable constraints in place. The program `Fractal.java` implements a pitch mapping of fractals. It extends the `HelperGUI` which makes it convenient to generate new melodies one after another to observe the fractal patterns. In this example the pitches are constrained to the C major scale. Below is the code for mak-

```
            rg[kg] = (float)(Math.random());
        }
    }

    private void calcNotes() {
        for (k=0; k<npts; k++) {
            threshold = np;
            ng = nbits;
            while(k%threshold != 0) {
                ng--;
                threshold = threshold / 2;
            }
            sum = 0;
            for (kg=0; kg<nbits; kg++) {
                if (kg<ng) rg[kg]=(float)(Math.random());
                sum += rg[kg];
            }
            Note note = new Note((int)(sum/nbits*127), Q);
            phr.addNote(note);
        }
    }

    private void output() {
        View.show(score);
        Write.midi(score, "FractalMusic.mid")
    }
}
```

More information about creating music from fractals can be found in "Computer Music" by Charles Dodge and Thomas A Jerse (1997), "Composing Music with computers" by Eduardo Miranda (2001), and "Fractals in Music" by Charles Madden (1999).

Audio Instruments

Audio instruments are at the core of jMusic's audio features, in the same way that Notes are at the centre of its musical structures. An instrument is an audio data generating class, that can be used to render scores as audio files based on digital audio data from either files or synthesis processes. The declaration of a new instrument is quite straight forward, as in this declaration of the `SineInst` instrument which is a simple sine wave oscillator.

The rendered file will have one or more channels depending upon the instruments used. Most instruments default to mono (one channel) or stereo (two channels), but instruments, and the au format can support any number of channels. However, only mono and stereo files can be played back inside Java.

```
Play.au();
```

To play back an au file from disk us the `Play.au()` method, and pass it the file name to be played.

```
Play.au("Arpeggio.au");
```

For convenience, the HelperGUI can be used to play back an audio files from classes that extend it.

Audio rendering

Now, let's look at an example that uses these audio features of jMusic. In the *Arpeggio* program a score of a repeating phrase is created first. Then, a sine wave instrument is declared and used to render the score as an audio file to disk. Finally, the file is played back. This program has only three lines that make it different from the previous MIDI-based programs. These deal with the audio instrument declaration, file rendering, and file playback.

```
Instrument sineWave = new SineInst();
Write.au(score, "Arpeggio.au", sineWave);
Play.au("Arpeggio.au");
```

Here is a listing of the entire Arpeggio program.

```
import jm.JMC;
import jm.music.data.*;
import jm.music.tools.*;
import jm.util.*;
import jm.audio.*;

public class Arpeggio implements JMC {

    public static void main(String[] args) {
        new Arpeggio();
    }

    public Arpeggio() {
```

play each line represents one value. In this program, a compact way of creating a simplistic jMusic score is shown where each music data object is taken as an argument to constructing the one that contains it. A file called `name.au` is written, but is empty and can be deleted.

```
Score s = new Score(new Part(new Phrase(new Note(C4, 0.01))));
Instrument inst = new PrintSineInst(4000);
Write.au(s, "name.au", inst);
```

The result of running this program will appear similar to figure 12.4. The 'o' characters plot the amplitude where higher values are to the right, and the precise sample values are shown in a column on the right of the display.

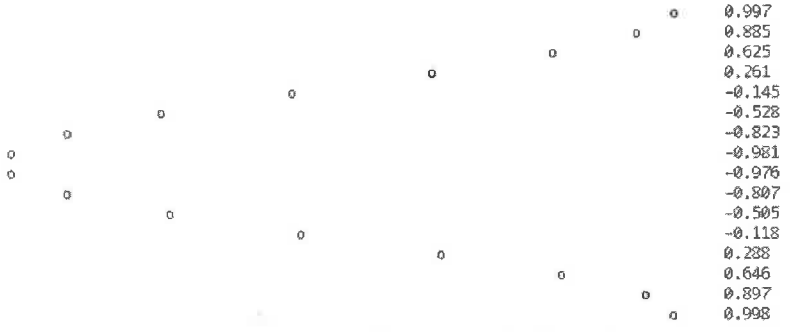


Figure 12.4. PrintAudio output.

There are a few of important things to notice about the data. The numbers are floating point values. All audio processing in jMusic happens as floating point numbers which ensures that the resolution (quality) is extremely high. The numbers are both positive and negative. The audio values centre around zero with louder values being those most distant from zero. The numbers are close to zero in value. Numbers produced by each note in jMusic are between -1.0 and 1.0 in value and so if three notes play at the same time their combined maximum value will be 3.0 or -3.0. Because the largest floating point values in Java can be in the millions you can see that there is plenty of head room in the use of floating point values (we can have very dense textures without the risk of distortion from clipping).

```
View.au()
```

A more conventional waveform display is available in jMusic and can be accessed with the `View` class. To see the output of the *Arpeggio* program using this viewer, add this line to the end of the program.

```
View.au("Arpeggio.au");
```

There is a `View.au()` button on the `HelperGUI` interface to allow display of composed files in the waveform viewer. The details required to make the `HelperGUI` work with audio are outlined below.

SineTest

This program generates sine waves with only a small frequency difference between them, the concurrent playback of these will produces audible 'beating' that emerges from the interaction between the sine waves. The speed of this beating is determined by the ratio between the sine wave frequencies. Listening for beating between sounds that are close in pitch is a common technique for tuning instruments, particularly stringed instruments such as guitar, violin, and piano.

```
import jm.JMC;
import jm.music.data.*;
import jm.midi.*;
import jm.util.Write;
import jm.util.Read;
import jm.util.View;
import jm.audio.Instrument;
import jm.music.tools.Mod;

public final class SineTest implements JMC{
    public static void main(String[] args){
        Score s = new Score();
        s.setTempo(60);
        Part p = new Part("Sine", 0);

        Phrase phr = new Phrase(0.0);
        for (int i=0; i<50;i++) {
            Note n = new Note(Math.random()*100+1000, 4.0);
            n.setPan(Math.random());
            Phrase phrase2  = new Phrase(n, Math.random()*30);
            p.addPhrase(phrase2);
        }
        p.addPhrase(phr);
        s.addPart(p);

        View.show(s);

        Instrument inst = new SineInst(22000);
```

channel of the sample file, therefore the rendered file has the same specifications in this regard as the input sample.

Each SimpleSampleInst is associated with one audio file. A collage of sound files can be created using many instances of the `SimpleSampleInst`, and scores can combine sample-based and synthesis-based instruments. Also, remember that other instruments can be designed with additional features and controls that you may require. One current restriction is that each instrument used to render the a score must use the same sample rate and channel attributes.

The *SampleRender* program, shown below, uses the `SimpleSampleInst` to construct a short 'melody' from a spoken word sample.

```
import jm.JMC;
import jm.music.data.*;
import jm.util.*;
import jm.audio.*;

public class SampleRender implements JMC {
    public static void main(String[] args) {
        new SampleRender();
    }
    public SampleRender() {
    int[] pitchArray = {C4, G3, C4, E4, F4, G4, C5, G5, G4, G3, G2, C3};
    double[] rhythmArray = {1.0, 0.25, 0.25, 0.25,0.25,
            0.5, 0.5, 0.125, 0.125, 0.25, 0.5, 2.0};
    Phrase phr = new Phrase();
    phr.addNoteList(pitchArray, rhythmArray);
    Score score = new Score(new Part (phr));
    Instrument ssi = new SimpleSampleInst("Hi.au", FRQ[C4]);
    Write.au(score, "SampleRender.au", ssi);
    View.au("SampleRender.au");
    }
}
```

Rendering audio with any file

In the final analysis digital audio data is simply a list of numbers. Therefore, it is a logical step that any list of numbers can be turned into an audio file. All computer files consist of numbers and can therefore be sonified. The data in most files will simply sound like noise bursts because it is not intended to be organized in any musical way, nevertheless, this can still be an interesting exercise and a basis for creative musical application. jMusic includes an instrument specifically designed to

```
import jm.music.data.*;
import jm.util.*;
import jm.music.tools.Mod;
// This version creates several parts and adds them
// to a score. Several patterns can be used also.
// There is an accent added at quarter note intervals.
// The repeat length is modified to suite the pattern length.
public class ArpAudioMidi implements JMC {
    int[] pattern1 = {0,3,7,0};
    int[] pattern2 = {0,12,10,7,0,3,5,7};
    int[] pattern3 = {3,7,15,19};
    int[] progression = {0,2,0,5,0};
    int repeats = 8;
    double tempo = 130.0;
    Score arpScore = new Score("Argeggio Music", tempo);
    Part arpPart, arpPart2, arpPart3;

    public static void main(String[] args) {
        ArpAudioMidi myArp = new ArpAudioMidi();
        myArp.makeScore();
        myArp.saveAs();
    }
    public void makeScore() {
        // bass
        arpPart = new Part("Apreggio 1", SYNTH_BASS, 1);
        // part, root, arp pattern, random?
        arpeggiator(arpPart, C2, pattern1, true);
        arpScore.addPart(arpPart);
        // upper part
        arpPart2 = new Part("Apreggio 2", POLYSYNTH, 2);
        arpeggiator(arpPart2, C5, pattern2, false);
        arpScore.addPart(arpPart2);
        // middle part
        arpPart3 = new Part("Apreggio 3", SYNVOX, 3);
        arpeggiator(arpPart3, C3, pattern3, true);
        arpScore.addPart(arpPart3);
    }
    public void arpeggiator(Part arpPart, int rootNote, int[] pattern,
            boolean randomOrder) {
        for (int j=0; j<progression.length; j++) {
```

To access an element in an array use the position in the array. To get the third element of `myArray` declared above, use `myArray[2];`. Notice that because we start counting at zero, 2 is the third element.

Vectors

Vectors are dynamic arrays of objects. Vectors can expand and contract as elements are added to them or deleted from them (unlike arrays that have a fixed size). To create a vector use the following code, `Vector myVector = new Vector();`.

To add an element to a vector use the `addElement()` method. For example, to add a note n to myVector use the code, `myVector.addElement(n);` vectors can only contain Java objects, and you can also add primitive types (such as int, float, double, or boolean) to a vector.

To retrieve an element from a vector use, `Note n2 = (Note)myVector.getElementAt(0);` This may look a bit complicated at first. It is necessary to specify the vector index to the `getElementAt()` method. It will be returned as type `object` from the array and therefore must be cast (converted) into the type required – in this case a `Note`.

Inheritance

Classes may extend another class, which means that the new class will inherit all the functionality of the previous class it extended. This is called inheriting the features of the class. The new class can then add more data or functionality as required. The notion of inheritance is an important aspect of object oriented programming. It is useful when class requirements are similar but not identical. A class with the overlapping functions can be created and each of the specific classes can extend the one with the base functionality. When a class is declared is may extend another class, like in this example;

```
public class Car extends MotorVehicle {
        ….class info. here …
}
```

An example of inheritance in the jMusic library is the `cpn` package that has classes to display a phase as common practice notation. Notes are displayed as CPN on a stave. All staves are similar, in that they have five lines and should display notes and rests, so jMusic has a `Stave` class that outlines these features. There is a `TrebleStave` and a `BassStave` class that each extend the `Stave` class. Each of these extended classes positions notes in a different location, for example middle C on the `TrebleStave` is one ledger line below the stave, while the same pitch in the

13: Sonic Spectra

Audio waveforms can be viewed from at least two perspectives, in the amplitude domain (as pressure changes over time) and in the frequency domain (as a summation of sine tones at different frequencies). In previous chapters we focused on audio in the time domain. In this chapter we will continue this by examining pitch and frequency in more detail, and explore aspects of sound spectra in the frequency domain including the harmonic series, timbre, and related compositional practices.

Music in the Physical World

Music is the perception of organized sound, and sound is created by moving objects. As a result it is not surprising that the studies of music and physics come together quite naturally. To create music or sound with a computer is to recreate the vibrating properties of physical or imaginary sonic devices and to stimulate loud speakers to produce these sounds. In the following sections we will look at some of the aspects of music and sound where physics plays an important role.

Pitch in hertz

Pitches sounds involve repeating waveforms, often called periodic waves, that have a fundamental pitch which is determined by how frequently the wave repeats. The time between the onset of each repletion is the period of the wave. Waveforms with a longer period have a lower fundamental frequency and pitch. Shorter periods equate to a higher fundamental frequency and pitch. The simplest periodic waveform is the sine wave, but all pitched sounds are periodic.

When `Notes` are constructed in jMusic their pitch can be specified as a MIDI note value or as a frequency in hertz. Notes can take an `int` or a `double` value as a pitch value. When taking an `int` value it is assumed that a MIDI note number (i.e., `60 = C4`) are desired. When taking a double value for pitch, it is assumed the value is a frequency in hertz (i.e., `440.0 = A4`). Because note frequency is represented as a double value (64 bit fractional) the pitch accuracy available in jMusic is ridiculously fine (e.g., 1026.08940567 Hz). This is great news for micro-tonal composers. When specifying the frequency value, division can be used as a convenient way of accurately calculating harmonic intervals from a fundamental. For example,

```
Note noteE5just = new Note(440.0 * 3 / 2, CROTCHET);
```

```
            Note note = new Note(Math.random()*500+1000,4.0);
            note.setPan(Math.random());
            Phrase phrase = new Phrase(note, Math.random()*10);
            part.addPhrase(phrase);
        }
        score.addPart(part);
        Instrument inst = new SineInst(22000);
        Write.au(score, "SineTest.au", inst);
        View.print(score);
    }
}
```

Math Moment

Linear and non-linear: When considering the relationship between chromatic pitch and frequency it is clear that the two are not equal. In fact, the chromatic pitch scale is linear, every step is the same distance apart – one half-step (semitone), while the frequency scale is exponential, every octave step is double the previous one. We say that the frequency spectrum is non-linear. As well, the common practice notation stave is non linear also because a some steps between lines and spaces are a half-step (semitone) while others are a whole-step (tone). In many cases conversion between linear and non-linear systems can become quite complex. As you can imagine there are many non-linear systems, because any relationship that does not have a 1:1 ratio at each step is non-linear. Another example of nonlinearity is the loudness scale, measured in decibels. Decibels are measured on a logarithmic scale. However, MIDI velocity values and mixing desk sliders usually generate linear values, therefore conversion between these scales is almost always taking place in the software or hardware device. These linear and non-linear measuring relationships are not arbitrary, they are based on the way physical systems in our world operate and on how our brain perceives them, therefore they are systems we need to understand as we create music in Java.

Tuning systems

Tuning systems define a set of pitches at particular frequencies. In many cases (but not always) the tuning system defines a pitch pattern within one octave and relative to a starting pitch; a pattern which is repeated in higher and lower octaves. By convention, modern Western music uses the chromatic tuning system which divides and octave into twelve equal steps. Other tuning systems divide the octave into more or less steps, for example a pentatonic system would have five steps, and the relationship between steps may vary widely. Even the term octave, implying eight, is derivative of the western system of eight note scale, but we will use it here to describe a range between one frequency and another double it. As you can imagine the range of possible tuning systems is

```
    double octave = (int)(pitches[i] / 12 - 5) * 2.0;
    if (octave == 0) octave = 1;
    if (octave < 0) octave = -1 * (octave/1.0);
```

Here is the full code for the tuning systems example:

```
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.audio.Instrument;

public final class TuningSystems implements JMC{
    public static void main(String[] args){
        new TuningSystems();
    }
    // Batok Mikrokosmos
    public TuningSystems() {
        // soprano
        int[] pitchSop = {G4,A4,B4,BF4,A5,AS5,B4,A4,BF4,AF4,G4,A4,B4,BF4,
                C5,CS5,D5,CS5,B4,C5,BF4,C5,B4,A4,BF4,G4,D5,CS5,B4,C5,BF4,
                C5,B4,A4,BF4,G4,REST,E4,A4,BF4,REST,E4,A4,B4,REST,
                E4,A4,C5,REST};
        double[] rhythmSop = {Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,2.5,
            Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,
            C,Q,Q,Q,C,Q,Q,Q,C,Q,Q,Q,2.5};
        // alto
        int[] pitchAlto = {G4,D4,CS4,D4,G4,D4,A3, REST, E4,DS4,CS4,
            REST,D4,CS4,B3,FS4,CS4,REST,FS4,CS4,REST,FS4,CS4,REST,F3};
        double[] rhythmAlto = {DC,C,DC,C,DC,C,2.5,DC,C,DC,C,DC,C,DC,C,
            Q,Q,DC,Q,Q,DC,Q,Q,DC,2.5};
        //set up the instruments
        Instrument insts = new SquareLPFInst(44100);
        //test
        Phrase phr = new Phrase();
        for (int i=0; i<56; i++) {
            phr.addNote(new Note(i+24, SQ));
        }
        Score s = new Score(new Part(phr));
        // tuning system frequency ratios
        double[] chromatic = {1.0,1.059,1.122,1.189,1.26,1.335,1.414,1.498,
               1.587, 1.682, 1.782, 1.888};
```

```
            if (pitches[i] == REST ) {
                int pitch = REST;
                phrase.addNote(new Note(pitch, rhythms[i]));
            } else {
                int degree = pitches[i]%12; // assumes we're in C maj or A min
                double octave = (int)(pitches[i] / 12 - 5) * 2.0;
                if (octave == 0) octave = 1;
                if (octave < 0) octave = -1 * (octave/1.0);
                double pitch = FRQ[pitches[i] - degree] * modeRatios[degree] *
                     octave;
                phrase.addNote(new Note(pitch, rhythms[i]));
            }
        }
        part.addPhrase(phrase);
        return part;
    }
}
```

The Harmonic Series

Complex sounds, such as those in the acoustic world, have a wave shape that is more complex than a single sine wave. The more complex the waveform, the more uneven and jagged the waveform looks, and the more bright or harsh it will sound. This complexity is the result of combining many sine waves of different frequencies together. Each different frequency is known as an overtone. While the frequencies of musical sounds has been studied for thousands of years, since the ancient Greeks, the fact that sound timbre can be described as combined overtones of different frequencies was proved mathematically by Joseph Fourier in the 17th Century. He showed that for periodic waveforms the sine wave components are restricted to integer multiples of the fundamental frequency. Such that if the fundamental frequency is 100 hertz, the component sine waves are at frequencies of 100, 200, 300, 400 hertz, and so on. An overtone at these simple multiples of the fundamental is known as a harmonic, and the set of all these harmonics make up the harmonic series. The processes of deconstructing a sound into its harmonic components is called a Fourier analysis in recognition of this discovery.

Additive synthesis

As we have seen, the sine wave is the basis of textual spectra. Additive synthesis uses multiple sine waves to create complex timbres, and provides detailed control over spectral development because each partial (sine wave component) can have its own temporal evolution by controlling individual

More jMusic Instruments

In this section we will look at how the concepts of adding audio waveforms of different frequencies can be applied to creating synthesis instruments in jMusic.

Instrument-based spectrum

A more traditional approach to additive synthesis is to render a single note with an instrument that generates a complex waveform via additive synthesis. This is the approach used in the *AdditiveTimpani* program. In this program the definition of overtone frequencies and amplitudes is directed toward the instrument. The score itself is quite simple, with one note for each sound event – in this case Timpani hit. The `AddSynthInst` instrument is used and accepts the following arguments:

```
int - sample rate
double[] - frequency multipliers
double[] - amplitude multipliers
double[][] - break point envelope values for each overtone
```

The values for some of these arguments are shown below.

```
private double[] freqVals = {1.0,1.42,1.53,1.77,1.94,2.01,2.4};
private double[] volVals = {1.0,0.8,0.7,0.6,0.5,0.3,0.1};
```

Notice, that there are seven arguments for both `freqVals` and `volVals`, one for each of seven overtones. The `freqVals` data are used as a multiplier of the fundamental, which is derived from the note pitch. The `volVals` data are used to scale the volume of each overtone, notice that the higher overtones have less volume. A second aspect of the volume for each overtone is its envelope. The envelope data for the fundamental are shown below. Envelopes are covered in more detail in a later chapter, for now it is sufficient to understand that the data are pairs of Cartesian points describing the shape of the loudness contour of the overtone.

```
private double[] pointArray1 = {0.0,0.0,0.002,1.0,0.3,0.4,0.7,0.05,1.0,0.0};
```

An array for each envelope can be provided and collected as a two dimensional array (an array of arrays) before being passed to the `AddSynthInst` constructor. In *AdditiveTimpani* only three envelope arrays are created and are reused many times to save some effort.

```
Instrument inst = new AddSynthInst(44100, freqVals, volVals, points);
```

A timpani sound is somewhat pitched but still has a distinctive noisiness characteristic of percussive sounds. The frequency ratios used in this example reflect this imperfection by being values

```
            note.setDuration(note.getRhythmValue() * 2.5);
            phr.addNote(note);
        }
        part.addPhrase(phr);
        score.addPart(part);
        // Construct the double array of break point envelope
        points[0] = pointArray1;
        points[1] = pointArray2;
        for (int i=2; i<freqVals.length; i++) {
            points[i] = pointArray3;
        }
        // Create instrument, render and view score.
        Instrument inst = new AddSynthInst(44100, freqVals, volVals, points);
        Write.au(score, "AdditiveTimpani.au", inst);
        View.au("AdditiveTimpani.au");
    }
}
```

AddSynthInst

```
import jm.audio.io.*;
import jm.audio.synth.*;
import jm.music.data.Note;
import jm.audio.AudioObject;
import jm.audio.Instrument;

public final class AddSynthInst extends Instrument{
       private double[] overtoneRatios;
       private double[] overtoneVolumes;
       private double[][] allEnvPoints;
       private double[] envPoints={0.0,0.0,0.05,1.0,0.15,0.4 0.9,0.3,1.0,0.0};
       private int sampleRate;

       public AddSynthInst(int sampleRate){
              this.sampleRate = sampleRate;
              double[][] tempPoints = new double[5][];
              for (int i=0; i<5; i++) {
                     tempPoints[i] = envPoints;
              }
              allEnvPoints = tempPoints;
              double[] temp1 = {1.0f, 3.0f, 5.0f, 7.0f, 9.0f};
```

the second envelope is applied to partials with asymmetric ratios that produce a gong-like timbre. The note, therefore, sounds as if it gradually transforms form a clarinet to a gong.

The *Morph* example uses the `AddMorphInst` instrument (detailed below) to create a simple audio output.

```
import jm.JMC;
import jm.music.data.*;
import jm.audio.*;
import jm.util.*;

public final class Morph implements JMC{
    public static void main(String[] args){
        new Morph();
    }
        public Morph() {
        Score score = new Score(new Part(new Phrase(new Note(C3, 10.0))));
        Instrument inst = new AddMorphInst(22050);
        Write.au(score, "Morph.au", inst);
        View.au("Morph.au");
    }
}
```

The `AddMorphInst` class describes an An additive synthesis instrument implementation which cross fades between two different spectra.

```
import jm.audio.io.*;
import jm.audio.synth.*;
import jm.music.data.Note;
import jm.audio.Instrument;

public final class AddMorphInst extends Instrument {
    // class variables
    /** the relative frequencies which make up this note */
    private double[] freqVals = {1.0, 3.0, 5.0, 7.0, 9.0, 1.0, 0.5699, 0.57,
        0.92, 1.1999, 1.2, 1.7, 2.74, 3.0, 3.76, 4.07, 7.3, 9.4, 10.0};
    /** the volumes to use for each frequency */
    private double[] volVals = {1.0, 0.7, 0.5, 0.3, 0.2, 1.0, 0.8, 0.75, 0.7,
        0.65, 0.6, 0.55, 0.5, 0.45, 0.4, 0.35, 0.3, 0.15, 0.1};
    /** The points to use in the construction of Envelopes */
    private double[] pointArray1 = {0.0, 0.0, 0.1, 1.0, 0.3, 0.4, 1.0, 0.0};
    private double[] pointArray2 = {0.0, 0.0, 0.5, 0.2, 0.8, 1.0, 1.0, 0.0};
```

14: Interactivity

This book has focused on musical composition with Java, and the approach as generally been to write and run programs that render output as either MIDI or audio files. Some interactivity was introduced in when using the `HelperGUI` class to present a visual interface that allowed you to change variables then rerun your program, that is, interactivity assisted with rapid re-rending of music without the need to recompile and restart your programs. In this chapter we will examine how your programs can run in real-time, by this we mean how they can generate music on the fly, playing back note by note as the music is generated. This will allow us to interact with the music to effect it during creation. This style of interactivity is common these days with faster computers can could more accurately be described as being instrument building or performing, than composing, but these boundaries are increasingly meaningless.

Real-time music generation

In order to playback the music as it is created (in real-time) jMusic uses a different structure from what we've used for composition. Notes are still the primary musical object, but phrases, parts and scores are replaced by the `RTLine` and `RTMixer` class. A class that extends `RTLine` is responsible for 'producing' notes as requested. A class that extends `RTMixer` coordinates a number of `RTLines` to keep them in sync. Instruments are required to play the notes and these are declared by your application and passed to the `RTLine`. Figure 14.1 provides an overview of this structure.

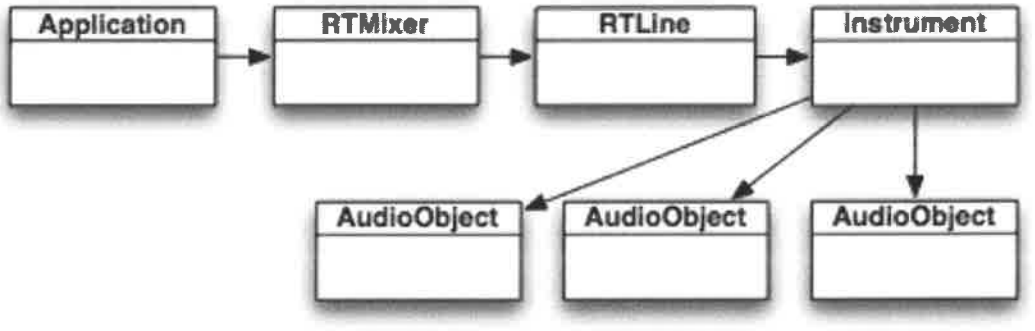


Figure 14.1. The structure of a simple real-time class hierarchy in jMusic.

This BassLineA class is called by another class named RTComposition. Notice that the BassLineA constructor requires an array of instruments to be passed to it. Correspondingly, the RTComposition class creates an array of one instrument to pass to BassLineA.

```
import jm.JMC;
import jm.music.data.*;
import jm.audio.*;
import jm.music.rt.RTLine;

public class RTCompositionA implements JMC {
    private RTMixer mixer;
    private BassLineA bassLine;

    public static void main(String[] args) {
        new RTCompositionA();
    }

    public RTCompositionA() {
        int sampleRate = 44100;
        int channels = 2;
        RTSawLPFInstA synthBass = new RTSawLPFInstA(sampleRate,
            1000, channels);
        Instrument[] insts = new Instrument[] {synthBass};

        bassLine = new BassLineA(insts);
        bassLine.setTempo(104);

        RTLine[] lineArray = new RTLine[] {bassLine};
        mixer = new RTMixer(lineArray);
        mixer.begin();
    }
}
```

The RTComposition class first creates an instrument with the desired sample rate and the number of channels (2 indicating stereo). The instrument, named synthBass, is added to an instrument array, named insts, and is used as an argument when constructing an instance of the BassLineA class, named bassLine.

Having created the bass line and provided it with a synth bass instrument, the next step in jMusic real-time audio is to mix all the lines together using the RTMixer class. In this example there is

next stage of our real-time journey where we will control the filter cutoff position from a graphical interface.

Real Time GUI Control

The first step for our graphical interface will be to add Start and Stop buttons to begin and end (or pause) the bass line playback. The interface makes use of Java GUI libraries and the `ActionListener` interface is used to react to users clicking on the buttons. Below is a revised version of the `RTComposition` class with the added GUI code. The classes `BassLineA` and `RTSawLFOInstA` are unchanged.

```
import jm.JMC;
import jm.music.data.*;
import jm.audio.*;
import jm.music.rt.RTLine;

import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.*;

public class RTCompositionB implements JMC, ActionListener {
    private RTMixer mixer;
    private BassLineA bassLine;
    private JButton goBtn, stopBtn;
    private boolean firstTime = true;

    public static void main(String[] args) {
        new RTCompositionB();
    }

    public RTCompositionB() {
        int sampleRate = 44100;
        int channels = 2;
        RTSawLPFInstA synthBass = new RTSawLPFInstA(sampleRate, 1000,
            channels);
        Instrument[] insts = new Instrument[] {synthBass};
        bassLine = new BassLineA(insts);
        bassLine.setTempo(104);
        displayGUI();
```

The additional methods in this class are `displayGUI()` and `actionPerformed()`. The `awt` and `swing` packages are imported for the interface components. The `displayGUI()` method is called at the end of the constructor and the button clicks will initiate the `actionPerformed()` method because the buttons are registered with the `ActionListener`. There is a Boolean flag named `firstTime` that is used in the `actionPerformed()` method when the `goBtn` is clicked. When `firstTime` is true, the mixer is declared and started, then `firstTime` is set to false. On subsequent clicks on the `stopBtn` and `goBtn` the mixer is paused or unpaused.

The next extension to the *RTComposition* program is to add a fader that controls of the filter cutoff. This requires the addition of a `JSlider` and a `JLabel` component. The `ChangeListener` interface is used to detect adjustment of the slider, and information from this is passed to the `RTMixer` (see figure 14.1) and the label value is updated.

```
import jm.JMC;
import jm.music.data.*;
import jm.audio.*;
import jm.music.rt.RTLine;

import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.*;

public class RTCompositionC implements JMC, ActionListener,
            ChangeListener {
    private RTMixer mixer;
    private BassLineC bassLine;
    private JButton goBtn, stopBtn;
    private boolean firstTime = true;
    private JSlider cutoff;
    private JLabel value;

    public static void main(String[] args) {
        new RTCompositionC();
    }

    public RTCompositionC() {
        int sampleRate = 44100;
```

```
                RTLine[] lineArray = new RTLine[] {bassLine};
                mixer = new RTMixer(lineArray);
                mixer.begin();
                firstTime = false;
                cutoff.setEnabled(true);
            } else mixer.unPause();
            goBtn.setEnabled(false);
            stopBtn.setEnabled(true);
        }
        if (e.getSource() == stopBtn) {
            mixer.pause();
            goBtn.setEnabled(true);
            stopBtn.setEnabled(false);
        }
    }

    public void stateChanged(ChangeEvent e){
        if (e.getSource() == cutoff) {
            value.setText("" +(cutoff.getValue() * 100));
            mixer.actionLines(
            new Integer(cutoff.getValue()*100), 1);
        }
    }
}
```

The *RTCompositionC* program requires new versions of the line and instrument classes; `BassLineC` and `RTSawLPFInstC`. These classes each have an additional method to handle parsing the filter cutoff value. When the slider value is adjusted, a reference to the slider value is sent as an `Integer` along with an arbitrary reference number that indicates the value is a filter cut off value. This process, that jMusic uses to sending values to the filter, may seem somewhat complex but it is flexible enough to allow any number of controllers to be sent to any number of audio objects in instruments. The important line from the `stateChanged()` method is this.

```
mixer.actionLines(new Double(cutoff.getValue() * 100.0), 1);
```

The `actionLines()` method is the first step in a series of message passing steps to get the controller information from the application class to the correct `AudioObject`, in this case an instance of the `Filter` class. The path is depicted in figure 14.2. The application passes the value(s) to the `RTMixer` which broadcasts it to all `RTLines` registered with the mixer. The

This method is the only difference between the classes `BassLineA` and `BassLineC`. It is called from the mixer class which passes an object with the controller value (in this case a `Double`) and an `actionNumber`. In the `RTCompositionC` class we assigned the filter cutoff as action number 1. The value has to find its way to the `Filter` object which is in the instrument. So the next step is to pass the cutoff value to each instrument in the line. There is only one value in this case, but it is possible for there to be more, so we pass the value as an element in a double array. The value is passed to the `setController()` method of the instrument.

```
public void setController(double[] controlValues){
        filt.setCutOff(controlValues[0]);
}
```

The `setController()` method is the only addition to the `RTSawLPFInstC` class. Any instrument that wants to receive controller messages in this way needs to implement this method. It's job is to pass the controller value to the `Filter` instance, named `filt`. It does this by calling the `setCutOff()` method and passing the first (zeroth) element in the array.

In summary, the process of adjusting the filter cutoff from the slider involves four steps;

1) get the slider value and pass it (or a scaled version of it) to the mixer along with a controller index;

2) the mixer will broadcast the data to all lines;

3) the lines will pass on the information to any associated instruments and;

4) the instruments pass the value to the relevant audio object (in this case the filter object).

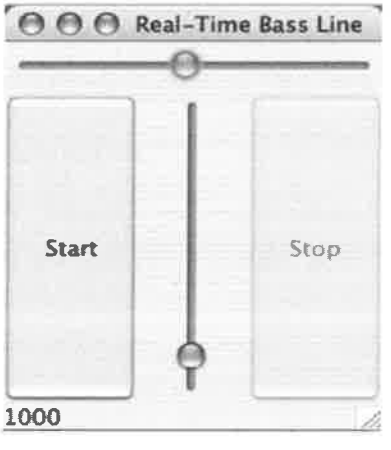


Figure 14.3: The control interface with a pan and filter cutoff sliders.

have our programs respond to that audio input in real time. In the later part of this section we'll see how to achieve this with jMusic.

MIDI input and output

The Musical Instrument Digital Interface (MIDI) protocol work by sending messages. There are a number of message types including note on and off, control change, program change, and system exclusive messages. The messages come in a package but Java has libraries that construct and deconstruct these to and from their constituent parts. jMusic adds an easy to use layer on top of the standard Java libraries that does the most commonly required tasks. The `jm.midi.MidiCommunication` class provides this functionality. It is an abstract class, this means that it cannot be used directly but must be extended. See the `MidiComTest` example below for an example. There is one abstract method that must be overridden also, the `handleMidiInput()` method which manages incoming MIDI data. The other important method is `sendMidiOutput()` which enables you to control external MIDI devices by sending messages to them. Both these methods act in real-time, that is they operate immediately they are called and there is no scheduling of events built into the `MidiCommunication` class (but of course you could add that if you wanted). The `handleMidiInput()` and `sendMidiOutput()` methods take four arguments, which are aspects of the MIDI message. To fully appreciate these arguments it is necessary to understand how a short MIDI message is constructed.

A standard short MIDI message is made up of three bytes; a status byte and two data bytes. The structure of a MIDI message is shown in figure 14.4. The status byte contains two parts, the type of message (note on, controller etc.) and the channel for the message (remembering that the MIDI specification allows for 16 channels – 0 to 15), The two data bytes contain different information depending on the message type but always have values between 0 and 127. For example, in the Note On and Note Off messages the first data byte has the pitch of the note and the second it's velocity (dynamic), while for a Controller message the first data byte contains the controller number (e.g. 7 for volume) and the second data byte contains the controller value.

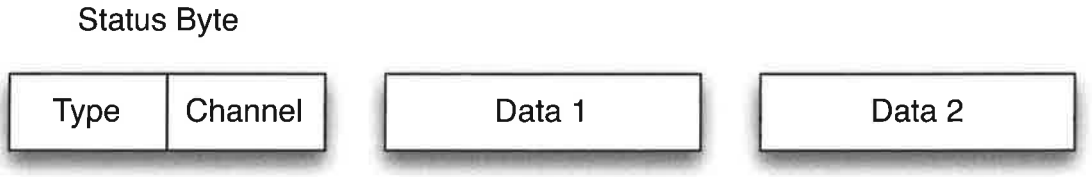


Figure 14.4. The structure of a short MIDI message.

Appendix A: Constants in the JMC

These words and abbreviations can be used in place of numbers in classes that implements the JMC interface. In many cases one value is represented by a series of alternate words, for example, the rhythm value 1.0 can be QUARTER_NOTE, Q, CROTCHET, or C. Notice that most of the constants are in upper case, which makes them easier to recognize amongst the Java code. Java is a case sensitive language and so they must be written using the case that appears here. Their numerical values are not shown here, but you can consult the jMusic documentation for the exact value of each constant.

RhythmValues

C, CD, CDD, CROTCHET, CROTCHET_TRIPLET, CT, DC, DDC, DDEN, DDHN, DDM, DDQ, DDQN, DEMI_SEMI_QUAVER, DEMI_SEMI_QUAVER_TRIPLET, DEN, DHN, DM, DOTTED_CROTCHET, DOTTED_EIGHTH_NOTE, DOTTED_HALF_NOTE, DOTTED_MINIM, DOTTED_QUARTER_NOTE, DOTTED_QUAVER, DOTTED_SEMI_QUAVER, DOTTED_SIXTEENTH_NOTE, DOUBLE_DOTTED_CROTCHET, DOUBLE_DOTTED_EIGHTH_NOTE, DOUBLE_DOTTED_HALF_NOTE, DOUBLE_DOTTED_MINIM, DOUBLE_DOTTED_QUARTER_NOTE, DOUBLE_DOTTED_QUAVER, DQ, DQN, DSN, DSQ, DSQT, EIGHTH_NOTE, EIGHTH_NOTE_TRIPLET, EN, ENT, HALF_NOTE, HALF_NOTE_TRIPLET, HN, HNT, M, MD, MDD, MINIM, MINIM_TRIPLET, MT, Q, QD, QDD, QN, QNT, QT, QUARTER_NOTE, QUARTER_NOTE_TRIPLET, QUAVER, QUAVER_TRIPLET, SB, SEMI_QUAVER, SEMI_QUAVER_TRIPLET, SEMIBREVE, SIXTEENTH_NOTE, SIXTEENTH_NOTE_TRIPLET, SN, SNT, SQ, SQD, SQT, THIRTYSECOND_NOTE, THIRTYSECOND_NOTE_TRIPLET, TN, TNT, TSN, TSNT, WHOLE_NOTE, WN

Pitches

A0, A1, A2, A3, A4, A5, A6, A7, A8, F0, F1, F2, F3, F4, F5, F6, F7, F8, AFN1, AN1, AS0, AS1, AS2, AS3, AS4, AS5, AS6, AS7, AS8, ASN1, B0, B1, B2, B3, B4, B5, B6, B7, B8, F0, F1, F2, F3, F4, F5, F6, F7, F8, BFN1, BN1, BS0, BS1, BS2, BS3, BS4, BS5, BS6, BS7, BS8, BSN1, C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, CN1, CS0, CS1, CS2, CS3, CS4, CS5, CS6, CS7, CS8, CS9, CSN1, D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, DFN1, DN1, DS0, DS1, DS2, DS3, DS4, DS5, DS6, DS7, DS8, DS9, DSN1, E0, E1, E2, E3, E4, E5, E6, E7, E8, E9, F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, EFN1, EN1, ES0, ES1, ES2, ES3, ES4, ES5, ES6, ES7, ES8, ES9, ESN1, F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, FFN1, FN1, FS0, FS1, FS2, FS3, FS4, FS5, FS6, FS7, FS8, FS9, FSN1, G0, G1, G2, G3, G4, G5, G6, G7, G8, G9, GF0, GF1, GF2, GF3, GF4, GF5, GF6, GF7, GF8, GF9, GFN1, GN1, GS0, GS1, GS2, GS3, GS4, GS5, GS6, GS7, GS8, GSN1, REST

HIGH_FLOOR_TOM, HIGH_TIMBALE, HIGH_TOM, LONG_GUIRO, LONG_WHISTLE, LOW_AGOGO, LOW_BONGO, LOW_CONGA, LOW_FLOOR_TOM, LOW_MID_TOM, LOW_TIMBALE, LOW_TOM, LOW_WOOD_BLOCK, MARACAS, MUTE_CUICA, MUTE_HI_CONGA, MUTE_TRIANGLE, OPEN_CUICA, OPEN_HI_CONGA, OPEN_HI_HAT, OPEN_TRIANGLE, PEDAL_HI_HAT, RIDE_BELL, RIDE_CYMBAL_1, RIDE_CYMBAL_2, SHORT_GUIRO, SHORT_WHISTLE, SIDE_STICK, SPLASH_CYMBAL, TAMBOURINE, VIBRASLAP

Scales and modes

AEOLIAN_SCALE, BLUES_SCALE, CHROMATIC_SCALE, DIATONIC_MINOR_SCALE, DORIAN_SCALE, HARMONIC_MINOR_SCALE, INDIAN_SCALE, LYDIAN_SCALE, MAJOR_SCALE, MELODIC_MINOR_SCALE, MINOR_SCALE, MIXOLYDIAN_SCALE, NATURAL_MINOR_SCALE, PENTATONIC_SCALE, TURKISH_SCALE

Tuning systems

EQUAL, JUST, MEAN, PYTHAGOREAN

Audio waveforms

COSINE_WAVE, PULSE_WAVE, SABERSAW_WAVE, SAWDOWN_WAVE, SAWTOOTH_WAVE, SINE_WAVE, SQUARE_WAVE, TRIANGLE_WAVE

Audio noise types

BROWN_NOISE, FRACTAL_NOISE, SMOOTH_NOISE, STEP_NOISE, WHITE_NOISE

Phrase position alignments

AFTER, BEFORE, CENTER_ALIGN, CENTER_ON_END, CENTER_ON_START, CENTRE_ALIGN, CENTRE_ON_END, CENTRE_ON_START, END_ON_CENTER, END_ON_CENTRE, END_TOGETHER, START_ON_CENTER, START_ON_CENTRE, START_TOGETHER

Appendix B: Methods in the Mod class

These are the methods of the `Mod` class in jMusic. Only the methods taking a phrase as an argument. Most `Mod` class methods are overloaded to accept `Parts`, `Scores` and `CPhrases` as arguments in place of phrases. Some similar method signatures have been omitted from this list, for a full list consult the jMusic documentation.

`accents``(Phrase phrase, double meter, double[] accentedBeats, int accentAmount)`

Increase the dynamic of notes at regularly occurring pulse locations which generates the sound of regular meter.

`addToDuration``(Phrase phrase, double amount)`

Vary the duration of each note in the phrase by the specified amount.

`addToLength``(Phrase phrase, double amount)`

Vary both the rhythm value and duration of each note in the phrase by the specified amount.

`addToRhythmValue``(Phrase phrase, double amount)`

Vary the rhythm value of each note in the phrase by the specified amount.

`append``(Phrase phrase1, Phrase phrase2)`

Adds phrase2 to the end of phrase1.

`bounce``(Phrase phrase)`

Adjusts all Note pan values to alternate between extreme left and right from note to note.

`changeLength``(Phrase phrase, double newLength)`

fadeIn(Phrase phrase, double fadeLength)

Linearly increases from zero the dynamic in the phrase.

fadeOut(Phrase phrase, double fadeLength)

Linearly decreases from zero the dynamic in the phrase.

fillRests(Phrase phrase)

Lengthens notes followed by a rest in the phrase by creating one longer note and deleting the rest.

increaseDuration(Phrase phrase, double multiplier)

Vary the duration of each note in the phrase by the multiplier.

increaseDynamic(Phrase phr, int amount)

Increases the dynamic by a certain amount - obviously a negative number will decrease it.

inversion(Phrase phrase)

Mirror the pitch of notes in the phrase around the first note's pitch.

merge(Phrase phrase1, Phrase phrase2)

Combines the phrases from a second CPhrase into the first.

mutate(Phrase phrase, int pitchCount, int rhythmCount, int[] pitches, int lowestPitch, int highestPitch, double[] rhythms)

Mutates the phrase by changing pitches and rhythm values.

normalise(Phrase phrase)

Increases dynamic values so that the loudest is at maximum level.

slurDown(Phrase phrase)

All descending sequences of notes are slurred by having their duration extended to 100% of the rhythm value.

slurUp(Phrase phrase)

All ascending sequences of notes are slurred by having their duration extended to 100% of the rhythm value.

spread(Phrase phrase)

Randomly sets the pan position of all notes across.

tiePitches(Phrase phrase)

Joins consecutive notes in the phrase that have the same pitch, creating one longer note.

tieRests(Phrase phrase)

Joins consecutive rests in the phrase creating one longer note.

transpose(Phrase phrase, int transposition)

Transpose the phrase up or down in semitones.

transpose(Phrase phrase, int transposition, int[] mode, int key)

Transpose the phrase up or down in scale degrees.

varyLength(Phrase phrase, double minLength, double maxLength)

Adjusts all note duration values to vary randomly between specified values.

Appendix C: Installing jMusic

Install Java

First, visit the `java.sun.com` site to install Java Standard Edition. Macintosh users note that Java is pre-installed on OS X.

Below are the instructions for downloading and installing jMusic.

Install jMusic on Linux / Unix

1. Download and install Java from Sun. (Skip this step if you have a Java already.)
2. Download the `jmusic_1.5.tar.gz` file from SourceForge and uncompress.
3. Place the `jmusic.jar` file and `inst` directories in your Java class path. This is usually done by editing your shell .*rc file (in OS X or in Linux a ~/.bashrc file) and add the location of the jMusic directories to the class path environment variable.
4. (Optional) To use the MidiShare package (msjm) for MIDI input and output you will need to download and install MidiShare from Grame, including the Java developer kit with the grame.jar file. You should run msconnect to connect the various midi input/output sources that you might have.

Eg. (for the tcshell) setenv CLASSPATH ${home}/jmusic/lib/jmusic.jar/:${home}/jmusic/inst/:.

Eg. (for the bash shell) export CLASSPATH=.:$HOME/jMusic/jmusic.jar:$HOME/jMusic/inst/

Note: The jmqt package requires Apple's QuickTime and will not run on Lunix or Unix. It has been deprecated since the introduction of JavaSound. You can safely ignore it.

Install jMusic on Mac OS X

1. From `http://sourceforge.net/projects/jmusic/download` the `jmusic_1.5.tar.gz` file from and uncompress. Place the jMusic folder in your user directory. e.g., `/User/andrew/jMusic`
2. Place the `jmusic.jar` file and the `inst` directory in your Java class path by modifying your `.profile` file (see below).

3. Some environment variables must be set before jMusic can be run successfully. Under Win2000, NT or XP.

   * Open the System Properties Dialog. (By right-clicking My Computer and going to Properties OR by naviagting through Start | Settings | Control Panel | System)

   * Choose the `Advanced` tab

   * Click `Environment Variables...`

   * Ensure that a variable called `CLASSPATH` exists and it contains `c:\jmusic\jmusic.jar;c:\jmusic\inst`

4. You may wish to try creating, compiling and running a simple jMusic program to test if the tools has been set up correctly. Here is a tutorial to try which will work on all flavours of Windows.

5. (Optional) To use the `MidiShare` package (`msjm`) for MIDI input and output you will need to visit `http://midishare.sourceforge.net/` and download and install `MidiShare`. You should run `msconnect` to connect the various MIDI input/output sources that you might have. The `midishare` user file is called `midishare186.zip` The `midishare` developer file is called `midishare2000.zip` They are in the same ftp folder, and the layout is a bit confusing (careful). You need to put the dlls that you want to use, as well as the `midishare32dll` (which has all the important native classes) into the system folder, so that they can be awakened at `midishare` startup time. Make sure the `.ini` files are put into the windows folder.

For even more information on getting going with Java try the getting started instructions from Sun's Java homepage.

References

Burraston, D., E. Edmonds, et al. (2004). Cellular Automata in MIDI based Computer Music. International Computer Music Conference. Miami, ICMA: 71-81.

Cook, N. (1990). Music, Imagination and Culture. Oxford, Oxford University Press.

Cook, P. R. (2002). Real Sound Synthesis for Interactive Applications, A. K. Peters.

Cope, D. (2001). Virtual Music: Computer synthesis of musical style. Cambridge, MA, MIT Press.

Dodge, C. and T. A. Jerse (1997). Computer Music. New York, Schirmer Books.

Gardner, H. (1985). The Mind's New Science: A History of the Cognitive Revolution. New York, Basic Books.

Gardner, M. (1970). "Mathematical Games: the fantastic combinations of John Conway's new solitaire game of 'life'." Scientific American **223**(4): 120-123.

Lindley, C. A. (2000). Digital Audio with Java. Englewood Cliffs, N.J, Prentice Hall.

Madden, C. (1999). Fractals in Music: Introductory mathematics for musical analysis. Salt Lake City, High Art Press.

Miranda, E. R. (2001). Composing Music with Computers. Oxford, Focal Press.

Moore, F. R. (1991). "The Dyfunctions of MIDI." Computer Music Journal **12**(1): 19-28.

Niemeyer, P. and J. Knudsen (2000). Learning Java. Seastopol, California, O'Reilly & Associates.

Reynolds, C. W. (1987). "Flocks, Herds, and Schools: A distributed behavioral model." Computer Graphics **21**(4): 25-34.

Risset, J.-C. (1969). Introductory Catalogue of Computer-Synthesized Sounds. Murray Hill, NJ, Bell Laboritories.

Roads, C. (1996). The Computer Music Tutorial. Cambridge, MA, MIT Press.

Schwanauer, S. and D. Levitt (1993). Machine Models of Music. Cambridge, MA, MIT Press.

Scruton, R. (1983). "Understanding Music." Ratio **25**: 97-120.

Selfridge-Field, E. (2001). Composition, Combinatronics, and Simulation: A historical and philosophical enquiry. Virtual Music: Computer synthesis of musical style. D. Cope. Cambridge, MA, The MIT Press: 187-220.

Xenakis, I. (1971). Formalized Music: Thought and Mathematics in Composition. Bloomington, Indiana University Press.