Contents

Chapter 17 – Using External APIs; Sound and Text to Speech in Java Programs

Introduction to Computer Science with Java

Chapter 17 – Using External APIs; Sound and Text to Speech in Java Programs

This chapter will help students to become more familiar with the use of external APIs, which sometimes involve importing class libraries that are not part of the Java SDK download. This chapter will use two examples of the use of APIs – adding sound to Java programs and creating Java programs with text-to-speech capability.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the concept of an *interface* in the Java programming language and how it relates to the general concept of a *software interface* for a class or package.
- describe what is meant by the term *Application Programming Interface (API)* and how it relates to a software library. Describe the benefits of a programming language that uses APIs and libraries implementing APIs.
- describe each of the three categories of Java APIs – *the Core Java API*, *official APIs not in the core*, and *third-party APIs*, and how their definition and use differs.
- describe the role of the *Java Community Process* in defining APIs and why some software developers have moved away from it.
- describe what a deprecated API is, why APIs become deprecated, and why it is best not to use deprecated APIs.
- describe what the *Java Sound API* is, how it is implemented, and some of the things that can be done using in it for Java programming.
- describe what *FreeTTS* is, what API it implements, who developed the API, who contributed to the development of the software, and what can be done with it.
- Create Java code that uses Core Java API implementations and that uses a third-party API that requires external libraries to be added to the software development project.

17.1 Interfaces, Application Programming Interfaces, and Libraries in Java

Before we can talk about Application Programming Interfaces (APIs), we need to discuss the word *interface,* which actually has several meanings. We can use *interface* as a verb to mean interacting with someone or something, or we can use *interface* as a noun meaning the mechanism by which we interface with something. In general an *interface* is the publicly available face of something, with which a user may interact. The set of buttons and controls for a car radio, for example, forms the interface that a person may use to interact with the inner workings of the radio. A Java programmer might view the radio as having an encapsulated interface.

In this sense, the publicly available methods of a class in an object-oriented, encapsulated programing environment form the software interface for the class. However, we need to be careful – in object-oriented programming languages like Java the term *interface* has a very specific technical meaning. According to the Java Language Specification (JLS), an interface "*has no implementation, but otherwise unrelated classes can implement it by providing implementations for its abstract methods.*"[1]

An **abstract method** is a method with no definition, only the method header. An abstract method is sometimes called a *bottomless method.* The term **interface** in Java code is a reserved word that specifically means a collection of abstract methods, similar to a class, but only the method headers exist.

Why would we need such a thing as an interface in Java? The answer lies at the heart of object-oriented programming – to provide a standard for encapsulated software that may be used in other software. By declaring an interface, a programmer is saying "this is how a set of methods should look and behave for the outside world." The interface can then be implemented by other software that replaces the abstract methods with concrete methods that actually do something. Classes that implement a known interface are telling the world that the methods in the class work as defined by the interface – they have the same names, the same parameters, and they return the same values. Other programmers then know how to use them in their own code.

Consider the Java *Comparable* interface. It is defined online at:
http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html

It is a simple interface with only one method – *compareTo()*. The definition of the interface tells us that the method compares one object to another and "*returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.*" This means that any class which implements the *Comparable* interface must have a *compareTo()* method to compare objects of the data type for the class, and it must return an integer as specified. We could create a class of houses, for example, and if it implements the comparable interface then we could compare two houses using the class's *compareTo()* method. We could then use the method to determine if one house comes before another house in order, or to sort a list of houses.

The Comparable interface doesn't tell us what property or properties we must use to order the houses, only that the *compareTo()* method can be used to establish the order of a set of houses. How the method is implemented is up to whoever creates a class that implements the interface.

[1] see the *Java Language Specifications for Java 7*, *Chapter 9 – Interfaces*, pg. 263.

So, to recap, in general, a class's **software interface** is its publicly accessible set of members of the class. This could include constants as well as methods – anything in the class that is public. In Java, the term *interface* within program code has a very specific meaning – an abstract definition of a software interface without method details and which can be implemented by other classes.

This brings us to the concept of an Application Programming Interface. Technically, an **Application Programming Interface (API)** is a publicly accessible software interface for the members of a library containing a class or set of classes. It is the software interface for a library defining how we interact with the library, but the term is often used to refer the library itself. Someone might say "*I downloaded the Java Sound API*" referring to the entire library, not just its interface.

Application Programing Interfaces and the libraries that implement them extend a programing language, such as the way the Math class extends Java. They can allow us to use from within our own code sophisticated software written by experts. The *Java Data Base Connectivity (JDBC) API* , for example, defines how a standard SQL relational database may be accessed from within Java code. The Java packages *java.sql* and *javax.sql,* which are included with the Java Standard Edition download, implement the JDBC API. Later in the semester, we will learn to use this API to access relational databases from within our own programs.

This use of API code from within our own programs fits the client-server model. APIs and packages implementing libraries that match the API provide services that can be called upon from a client program. For example, if we use JDBC to connect to a database, our program – the client – uses the services of a JDBC library to access a database.

Languages like Java that were designed to be serious production languages for major software often don't include a comprehensive set of instructions for everything the language can do, but form the basis of a system that can be greatly extended through the use of APIs and software libraries. The Java language itself has no instructions for graphics, sound, Web page connectivity, or many other essential functions for which the language is widely used. This functionality is provided by APIs and packages with libraries that extend the language far beyond its core capabilities. To be a serious Java programmer, one must learn how to use APIs and packages of software libraries.

Of course, to use an API for software in a specific technical field, one must also know about that field, or work with someone who does. If you don't know anything about graphics, for example, it can be very hard to develop graphics software, even with the best APIs and graphics packages.

The many different Java APIs and packages of libraries that implement them, generally fall into three categories:

- The Core Java API;
- Official Java APIs that are not part of the core;
- third party APIs.

In the remainder of this section we will look briefly at each of the three categories of Java APIs, then in the following sections we will examine the use of sound in computer programming and see several APIs that provide services for including sound in Java. We will learn how to use APIs and their corresponding libraries in NetBeans projects, and, along the way, we will also learn a little about including sound capabilities in Java software.

The Core Java API

The **Core Java API** is the set of classes in packages included with a download of the *Java Platform*, also known as the *Java Development Environment* or the *Java Development Kit*. A Java platform includes an implementation of the Java Language Specification (JLS), packages with the Core Java API, and a Java Runtime Environment (JRE).

Currently, there are three editions of the Java Platform – the *Standard Edition (Java SE)*, the *Enterprise Edition (Java EE)* and the *Micro Edition (Java ME)*. **Java SE** is the primary software environment for developing Java software for personal computers and similar systems. **Java EE** is an extension of the Java SE that includes features for creating large scale, secure, multi-tiered Web-based and network-based enterprise-wide software services. **Java ME** is Java development environment for embedded systems software, including software for mobile devices such as cell phones; for printers, video systems, sound systems and other electronic appliances; and for embedded controller chips in everything from digital watches to automobile engines and nuclear power plants. Oracle estimates that there are now more devices running Java software than there are people on the planet. All of this software was developed using the Java platforms with an associated sets of APIs.

The table below is a partial list of the packages included in the Java SE 7 Core API. For a complete list and more detailed information on the classes in the packages of the Java Core API, see the *Java Platform, Standard Edition 7 API Specification* online at: http://docs.oracle.com/javase/7/docs/api . It is the official API specification for the Java™ Platform, Standard Edition 7.

You will notice that you have already been using Java Core APIs in your Java programming projects. No special downloads or movement of JAR files is necessary to use the classes in the JavaCore APIs. They are included as part of the available libraries in a Java platform. However, *import* statements for the specific classes or packages of classes to be used must be included in source code files.

Java Core API Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains classes for creating user interfaces and for painting graphics and images.
java.beans	Contains classes related to developing *beans* -- components based on the JavaBeans™ architecture.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (`BigInteger`) and arbitrary-precision decimal arithmetic (`BigDecimal`).
java.net	Provides the classes for implementing networking applications.
java.nio	Defines buffers, which are containers for data, and provides an overview of the other NIO packages.
java.rmi	Provides the RMI package.
java.security	Provides the classes and interfaces for the security framework.

java.sql	Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java™ programming language.
java.text	Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.
java.util	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array). It also includes classes for concurrent programming and for creating and reading JAR files.
javax.accessibility	Defines a contract between user-interface components and an assistive technology that provides access to those components.
javax.crypto	Provides the classes and interfaces for cryptographic operations.
javax.imageio	The main package of the Java Image I/O API.
javax.lang.model	Classes and hierarchies of packages used to model the Java programming language.
javax.management	Provides the core classes for the Java Management Extensions.
javax.management.loading	Provides the classes which implement advanced dynamic loading.
javax.naming	Provides the classes and interfaces for accessing naming and directory services.
javax.net	Provides classes for networking applications and secure socket programming.
javax.print	Provides the principal classes and interfaces for the Java™ Print Service API.
javax.rmi	Contains user APIs for RMI-IIOP.
javax.script	The scripting API consists of interfaces and classes that define Java ™ Scripting Engines and provides a framework for their use in Java applications.
javax.security	A set of packages that provides a framework for security, authentication and authorization services including public key encryption and software security services,
javax.sound.midi	Provides interfaces and classes for I/O, sequencing, and synthesis of MIDI (Musical Instrument Digital Interface) data.
javax.sound.sampled	Provides interfaces and classes for capture, processing, and playback of sampled audio data.
javax.sql	Provides the API for server side data source access and processing from the Java™ programming language.
javax.swing	Provides a set of "lightweight" (all-Java language) GUI graphics and and event-driven components that, to the maximum degree possible, work the same on all platforms.
javax.swing.plaf.multi	Provides user interface objects that combine two or more look and feels.
javax.tools	Provides interfaces for tools which can be invoked from a program, for example, compilers.
javax.xml	an extensive set of packages for XML services
org.omg.CORBA	Provides the mapping of the OMG CORBA APIs to the Java™ programming language, including the class `ORB`, which is implemented so that a programmer can use it as a fully-functional Object Request Broker (ORB).
org.w3c.dom	Provides the interfaces for the Document Object Model (DOM) which is a component API of the **Java API for XML Processing**.

Official Java APIs not in the Core

Some Official Java APIs are downloaded separately from the Core Java APIs. Typically, new APIs are originally released separately from the Java Core, but are added to the core in later releases of Java.

The history of the *Swing* package is a good example of this. Netscape developed an API named the *Internet Foundation Classes (IFC)* to work with the original release of the Java language. It could be used to develop GUI software in Java. Later, IFC became part of the *Sun Java Foundation Classes (JFC)*, which are now the *Oracle JFC* that includes *Abstract Windowing Toolkit (AWT)*, *Swing*, and *Java 2D* for developing Java GUI software. The components of the JFC are now part of the Core Java API for all Java platform downloads.

Swing moved from being a third party API to an official API not part of the Core Java API, to its current status in the Core Java API. Many other APIs followed this path, migrating into the core. The Core API is larger and more extensive with each release of Java.

(*NOTE: Java 8 is due to be released on March 18, 2014. For more about Java 8, see https://jdk8.java.net/ and http://www.javaworld.com/article/2078836/java-se/love-and-hate-for-java-8.html).*

The table below describes some of the more popular official Java APIs that are not part of every Java core.

Name	Description
Java Advanced Imaging (JAI)	A set of interfaces that support a high-level programming model for manipulating images easily.
JavaHelp	A full-featured, extensible help system that enables you to incorporate online help in applets, components, applications, operating systems, and devices.
Java Media Framework (JMF)	An API that enables audio, video and other time-based media to be added to Java applications and applets.
Java Naming and Directory Interface (JNDI)	An API for directory services.
Java Speech API (JSAPI)	An API allows for speech synthesis and speech recognition.
Java 3D (J3D)	A scene-based 3D graphics API.
Java OpenGL (JOGL)	An API for using OpenGL graphics in Java.
Java USB for Windows (JUSB)	A USB communication API for Java applications
Java TV	A Java ME API for developing Java applications that run on TV and set top box devices.
Java Telephony	A Java ME API for developing telephone services using Java
Java Mail	A Java EE API for providing email services

Links to Information about most official Java APIs are available online at http://www.oracle.com/technetwork/java/api-141528.html

Third Party Java APIs

A **third party Java API** is an API intended for use with the Java programming language that was not implemented by Oracle or, previously by Sun Microsoystems. There are many API packages for Java released by non-profit organizations, user groups, individuals, and corporations. They range from free open source software packages to costly professional software for running a large business. Some of these APIs meet official specifications for Java APIs from Sun or Oracle, but most are specialized and include packages that implement their own APIs.

The **Java Community Process** is a mechanism for proposing Java Specification Requests (JSR) that might eventually become an API definition. Some of the JSRs have been adopted by Sun and Oracle and included in later releases of Java, while others are left as specifications guiding third party API development. Some large third party software developers, such as Apache, do not participate in the Java Community Process as much as they did before Oracle acquired Sun Micro Systems, voicing concern that Oracle now tightly controls the Java Community Process for its own purposes. Many third party APIs are developed independently of the Java Community Process.

The table below includes list of a few of the many third party Java APIs that are available today:

Name and Description	On the Web at:
Amazon Web Services (AWS) AWS is a commercially available sophisticated set of APIs for cloud computing, data storage, networking and ecommerce available from Amazon for other businesses. Many large corporations, including Adobe and the makers of the Canvas system used by CCP for distance education, use AWS.	aws.amazon.com
Apache POI APIs Several APIs that provides Java libraries for reading and writing files in Microsoft Office formats, such as Word, PowerPoint and Excel. They are often used to extract data from Office documents.	http://poi.apache.org
FreeTTS An open source Java Text-to Speech Engine	http://freetts.sourceforge.net/docs/index.php
Google APIs Client Library for Java A set of APIs that allow Java programmers to access features of Google HTTP APIs linked to many Google services.	https://developers.google.com/api-client-library/java/apis
JFugue An easy-to-use API for programming music in Java without the complexities of MIDI.	www.jfugue.org
JMonkeyEngine An open source 3D game engine for Java	jmonkeyengine.org

Lightweight Java Game Library (LWJGL) LWJGL is a open source software to enable commercial quality games to be written in Java.	www.lwjgl.org
NASA Java World Wind SDK API A free API that allows Java programmers to use NASA's World Wind technology for including a virtual globe, satellite imagery, aerial photography, and topographic maps in Java applications. The API also includes 2D and 3D graphics rendering features.	goworldwind.org
Twitter 4J Twitter 4J is free open source software that implements the twitter API allowing you to incorporate Twitter messaging and features in your Java programs.	twitter4j.org/en/index.html

The *Wicked Cool Java* Website, online at http://www.wickedcooljava.com/libraries.jsp has links to some interesting Java code libraries that implement Java APIs.

In January of 2000 *Computerworld* magazine published a short article introducing APIs, which I encourage you to read. It is still available online at: http://www.computerworld.com/s/article/43487.

In general, when we want to use a library that implements an API, there are a number of questions we need to ask about the acquisition of an API library, its use, and distributing software that uses an API:

- **Acquisition:** Where can I get the API? How much does it cost? What license is required for software that uses the API? Does it require other APIs to work properly?
- **Use:** What are the classes and methods in the API that provide the functions I need? How do I use the features of the API in my own classes and methods? What else do I need to do to use the API? Is good documentation available?
- **Distribution:** What libraries and JAR files need to be included for distribution with the project's JAR file? Is there anything else I need to include, and how should the finished software be packaged?

17.2 Sound in Computer Programs

Sounds are caused by vibrating molecules that cause adjacent molecules to vibrate, which in turn cause molecules adjacent to them to vibrate, and so on, resulting in what we know as sound waves. We experience sound from a variety of sources all the time, and can use sound to communicate by controlling the nature of sound waves, such by using our voices.

In their simplest form, sound waves have frequency and amplitude. The frequency, measured in cycles per second, or Hertz Hz, is related to the pitch or tone of the sound we hear. The amplitude, measured in logarithmic decibels (dB), is related to its energy and volume. Humans are capable of hearing sounds as low as around 12Hz and as high as about 18,000 Hz. Our ability to hear sound decreases with age.

Musical tones are formed by the frequency, duration, and shape of sound waves we hear. For example, Middle C on a piano has a frequency of roughly 262 Hz, close to 256 or 2^8. The frequency doubles with each octave, so the C above middle C has a frequency of roughly 523 Hz; the C below middle C has a frequency of roughly 131 Hz. A Wikipedia Commons image showing the frequencies of musical notes related to piano keys is online at: http://upload.wikimedia.org/wikipedia/commons/a/ad/Piano_key_frequencies.png

A musical chord is formed by several notes played at the same time, causing certain harmonic patterns in the sound.

Most people's voices can produce sounds in the range of about 60 Hz up to about 6,000 Hz but it specifically varies from person to person. *(Remember, frequency doubles with every octave, so 12,000 hz is only one octave above 6,000 hz, and some people can reach this tone with their voices.)* Younger children and females tend to have higher pitched voices than older people and males.

Common ordinary everyday human speech has a frequency range of roughly 3,100 hz (3.1 kHz) from the lowest pitched sounds we normally use when speaking to the highest. A *voice grade channel* in a communication medium is a channel that can carry a signal with this frequency range. Most electronic voice grade channels today actually use 4kHz of bandwidth for technical reasons.

Sounds can be recorded by electromechanical devices that turn the vibrations of sound waves into an electrical signal. This is what happens with a microphone, such as in the mouthpiece of a telephone. Conversely, sounds can be produced by any system in which an electrical signal causes mechanical vibrations, such as in a speaker or the earpiece of a telephone. Usually, a thin membrane like the head of a drum or like a human eardrum is used in the process of capturing sound as an electrical signal and in the process of producing sound from an electrical signal.

We can record sound with a computer by programming the system to capture and store the electrical patterns created by a microphone and we can produce sound from a computer by programming the system to generate patterns of electrical signals that cause a membrane in a speaker to vibrate.

In modern systems, the data related to sound is encoded as digital data, reduced to patterns of ones and zeroes. The number of times per second that a sound is sampled and recorded is called the sampling rate of the sound. Higher quality sound has a higher sampling rate and takes up more memory than lower quality sound with lower sampling rates. Typical music formats, such as *MP3*, vary from 8,000 samples per second (8kHz) up to about 48,000 samples per second (48kHz), although there are some formats that go much higher. (The sampling rates are measured in Hertz, but are not related to the frequency of the tone, also measured in Hertz.) Sampling rates of roughly 8, 11, 22 and 44 Khz are common, taking up anywhere from 8 kilobits to 320 kilobits of data per second of audio (kbps). For example, a stereo MP3 file with a 44 kHz sampling rate takes up about 256 kbps.

Computer can analyze sound and turn it into other data, such as in a *speech to text system*, or they can produce sound from underlying data, such as in a *text to speech system*. Computers can record sound and playback recorded sound, but they also can be programmed to produce original sound -- as simple as basic musical notes or as complicated as a synthetic human voice.

In the remainder of this chapter we will focus on computers creating sound, both from recorded sound files, and by synthesizing sounds based on text in a text to speech system. We will use this as a medium for learning about using API's and libraries in Java.

In our first look at using an API to produce sound, we will use the **Sun Audio API** to play a recorded sound. The Sun Audio API has been replaced by the newer **Java Sound API**, which we will use in section four of this chapter. The Java Sound API is actually very complicated, with the ability to capture incoming audio signals, control audio playback, engage in MIDI synthesis, and control basic MIDI sequencing.

MIDI (Musical Instrument Digital Interface) is a set of software and hardware standards for the electronic synthesis, recording and playback of sound. The standard is maintained by the MIDI Manufacturers Association (MMA), a group of experts representing the major makers of music industry equipment. MIDI defines the electronic interfaces necessary for computers to control and to communicate with electronic instruments and for electronic instruments to control and to communicate with each other. It was developed in the early 1980's, at about the same time that personal computers were becoming widely available.

The **Java Sound API** provides extensive MIDI capabilities, but an examination of MIDI is beyond the scope of this course. Anyone who is serious about music on a computer should be familiar with MIDI. The MMA has a 16-page introduction to MIDI, available online at: http://www.midi.org/aboutmidi/index.php , and a set of MIDI tutorials online at: http://www.midi.org/aboutmidi/tutorials.php.

17.3 The Sun Audio API (a Deprecated API)

One of the first mechanisms for programing a computer to deal with sound in a Java program was the Sun Audio API, which has since been deprecated. A **deprecated** API is an API that has been replace by newer technology. Generally newer technology should be used in place of deprecated technology.

Technology can be deprecated for many reasons – newer technology could work better, the older technology might not be secure, several old technology formats might be combined into a single new format, and so on.

Usually software that is deprecated, such as deprecated classes in Java, will still work for a while, but will give a warning message. This allows for a period of "*backward compatibility*" and a transition from the old software to the new, and it prevents problems such as older software not running on a newer processor or with a newer JVM.

Deprecated Sun APIs

Many API's in packages that begin with "*sun*" are older APIs from before the 2009 acquisition of Sun Microsystems by Oracle. Sun finalized their efforts to make Java free open source software in 2006, and stopped using the *sun* name on Java APIs well before then, so most APIs with the *sun* name are at least 10 years old.

A short but popular article on using the *Sun Audio API* to play music files from within Java programs appeared in the February 1997 edition of *JavaWorld* magazine. It is still available online and used by many novice programmers to add audio playback capability to in Java applications.

The code for using the Sun Audio API for playback can easily be replaced by similar code from the newer *Java Sound API*. The Sun Audio technique is included in this chapter for comparison and as an example of a deprecated API.

The Sun Audio API uses an *AudioStream* object for playing sound files. To use an *AudioStream* object, we must first create a *File* object linked to a specific data file, then create an *InputStream* for the file, just as with reading text from a data file. The AudioStream object will then interpret the incoming data as a sound file.

The Sun Audio API will work with *.wav*, *.au* and *.aiff* audio files.

Here is a sample of the code from the NetBeans project *SunAudioDemo* included with the files for this chapter:

```java
/* SunAudioDemo.java
 *
 * CSCI 112 - Spring 2014
 *
 * This program demonstrates the use of the now deprecated Sun Audio API to
 * play back an audio file.  It will work with .wav, .au and .aiff files.
 *
 * For this prograsm to owrk, the file named in line 23, must be in the same
 * directory as the jar file or project file. The original file it plays back is
 * an early recording of Edison reciting mary had a little lamb.
 *
 */

package sunaudiodemo;

import sun.audio.*;    //import the sun.audio package
import java.io.*;

public class SunAudioDemo {
    public static void main(String[] args) throws Exception {

        // identify the sound file as a File class object
        File soundFile = new File("edison.wav");

        // Open an input stream for the File object soundFile
        // This allows Java to read the file.
        InputStream inFile = new FileInputStream(soundFile);

        // Create an AudioStream from the input stream.
        // This tells Java to read the incoming data as sound data
        AudioStream audio = new AudioStream(inFile);

        // play the sound file using the start method from Audioplayer.player
        AudioPlayer.player.start(audio);

    } // end main()
} // end class SunAudioDemo
```

The program first creates a File object named soundFile, linked to the file *Edison.wav.* This is a short *.wav* file with a copy of an early recording made by Thomas Edison reciting "*Mary had a Little Lamb*".

An InputStream object named *inFile* is created for the input from soundFile. The AudioStream object *audio* allows the computer to treat the incoming data as recorded sound. The player method in the AudioPLayer class, which is part of the Sun Audio API, then plays the sound. You need to have headphones or speakers to hear the sound being played on your computer.

Programming Example – Warnings from Deprecated Classes in an Older API

The NetBeans Project ***SunAudioDemo*** is included with the files for this chapter. It uses the deprecated Sun Audio API, and will run, but produces warnings whenever classes from the API are used.

You should unzip the file and try the project in NetBeans, as follows:

STEP 1.

Download and unzip the file ***SunAudioDemo.zip*** included with the files for this chapter.

STEP 2.

Open the ***SunAudioDemo*** project in NetBeans. You can examine the source code, which matches what is shown above in this section.

STEP 3.

Compile the code by using the ***Test*** command on NetBean's Run menu or by pressing **[ALT + F6]**.

Several warnings will appear in the IDE's output window, similar to the following:

warning: AudioStream is internal proprietary API and may be removed in a future release

This warning will appear three times – once for each line of code that attempts to use features of the Sun Audio API, twice for the *AudioStream* class and once for the *AudioPlayer* class. It is telling us that we are using a deprecated API that, at some point in the future, will not be part of Java.

STEP 4.

Despite the warnings, the code will work. Run the program and you should hear the voice of Thomas Edison. You can close the project when finished.

Not all warnings about deprecated APIs are worded exactly the same, but generally, when we see a message telling us that a class *"may be removed in a future release"* we are being told that the class we are using is still part of the Core API, but it won't be in the future, which means that at some point the code will not work.

As of Java 7, the classes in the *Sun Audio API* still work, but we should really use the newer *Java Sound API*. The next section shows us how to do the same thing this project does, but with the *Java Sound API*.

17.4 The Java Sound API

The *Java Sound API*, part of the Core Java API, specifies classes allowing Java software to capture, mix and playback digital audio (also called *sampled* audio) and classes for MIDI synthesis, sequencing and event transport. The API is implemented in two primary packages:

- **javax.sound.sampled** – a package of classes for capture, mixing, and playback of digital audio.
- **javax.sound.midi** – a package of classes for MIDI synthesis, sequencing, and event transport.

The *javax.sound.sampled* package contains the classes we need to play audio files similar to the example in the last section that used a deprecated API. The classes in the *javax.sound.sampled* package will work with .AIFF, AIFF-C, AU, and WAV sound files. *javax.sound.sampled* does not handle MP3 files. A free online audio file format conversion service that can convert from *MP3* to WAV files is on the Web at: http://media.io. Many other similar services are available. The media.io site is supported by donations. Be aware that WAV files will be significantly longer than MP3 files.

Programming Example – Using the Java Sound API to Play a Sound Clip

Classes from the The **javax.sound.sampled** package are used in the zipped NetBeans project *EdisonSoundFrameDemo.zip,* included with the files for this chapter. Part of the code is discussed below.

You should download the file, run the program, and examine the code, shown and discussed briefly below.

STEP 1.

Download and unzip the file **EdisonSoundFrameDemo.zip**, included with the files for this chapter.

STEP 2.

Open the **EdisonSoundFrameDemo project in Netbeans.**

STEP 3.

Run the program. It will play the same sound clip as the SunAudioDemo project in the last section.

STEP 4.

Look through the code to see how it works. A copy of the code and discussion of the code is included below. Don't forget to close the project when you are done.

Notice two things about using **javax.sound.sampled** package – first, it does not give warnings indicating it is deprecated as the SunAudioDemo did (it is not deprecated), and second, it works by simply using import statements. No special files or libraries need to be added to NetBeans to make it work because it is part of the Java Core API.

Here is the code to play the Edison.wav file using The Java Sound API:

```
/* EdisonSoundFrameDemo.java
 *
 * CSCI 112 - Spring 2014   last edited Feb 13, 2014 by C. Herbert
 *
 * This program demonstrates the use of the Java Sound API's Javax.sound.sampled
 * package to playback a digital audio file (also called a sampled audio file).
 *
 * It is similar
 * For this prograsm to work, the file named in line 30, must be in the same
 * directory as the jar file or project file. The original file it plays back is
 * an early recording of Edison reciting "Mary had a little lamb". *
 */

package edisonsoundframedemo;

import java.io.*;
import javax.sound.sampled.*;
import javax.swing.*;

public class EdisonSoundFrameDemo extends JFrame {

    /* To play sound using the Clip class, the process need to be alive. It is
     * intended to run in an Applet or in a GUI application such as this.
     */

    // Constructor
    public EdisonSoundFrameDemo() {

        try {

            // identify the sound file as a File class object
            File soundFile = new File("edison.wav");

            // Create an AudioInputStream for the File object soundFile
            // This allows Java to read the file and read it as audio data in one step.
            AudioInputStream audioIn = AudioSystem.getAudioInputStream(soundFile);

            // create an audio Clip object so we can use the Clip class methods
            Clip myClip = AudioSystem.getClip();

            // open the myClip object and associate it with the audioIn AudioInputStream
            myClip.open(audioIn);

            // start playing the myClip audio Clip
            myClip.start();

        } catch (UnsupportedAudioFileException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (LineUnavailableException e) {
            e.printStackTrace();
        }  // end catch

    } // end JavaSoundPlaybackDemo()
    public static void main(String[] args) {

        // Instantiate a SoundPlaybackDemo object and set its properties.
        // Remember, it is a Jframe object(a JFrame sub-class object).

        EdisonSoundFrameDemo soundFrame = new EdisonSoundFrameDemo();
        soundFrame.setTitle("Sound clip playback demo");
        soundFrame.setSize(300, 200);
        soundFrame.setLocation(200, 100);
        soundFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // activate the JavaSoundPlaybackDemo object
        // no sound will be heard unless the JFrame is activated
        soundFrame.setVisible(true);

    } // end main()
} //end class JavaSoundPlaybackDemo
```

As you can see in the above code, the *Clip* class, part of the *javax.sound.sampled* package, is used to play a sound file. This package requires us to treat a Clip like a GUI component, which means it must be included in an Applet or in a container such as a JFrame. Much of the code shown above sets up a blank *EdisonSoundFrameDemo* object, which extends JFrame. (The frame in this program is purposely left blank to keep the code less complicated for demonstration purposes.) Here is the part of the above code that actually plays the sound:

```
   // identify the sound file as a File class object
   File soundFile = new File("edison.wav");

   // Create an AudioInputStream for the File object soundFile
   // This allows Java to read the file and read it as audio data in one step.
   AudioInputStream audioIn = AudioSystem.getAudioInputStream(soundFile);

   // create an audio Clip object so we can use the Clip class methods
   Clip myClip = AudioSystem.getClip();

   // open the myClip object and associate it with the audioIn AudioInputStream
   myClip.open(audioIn);

   // start playing the myClip audio Clip
   myClip.start();
```

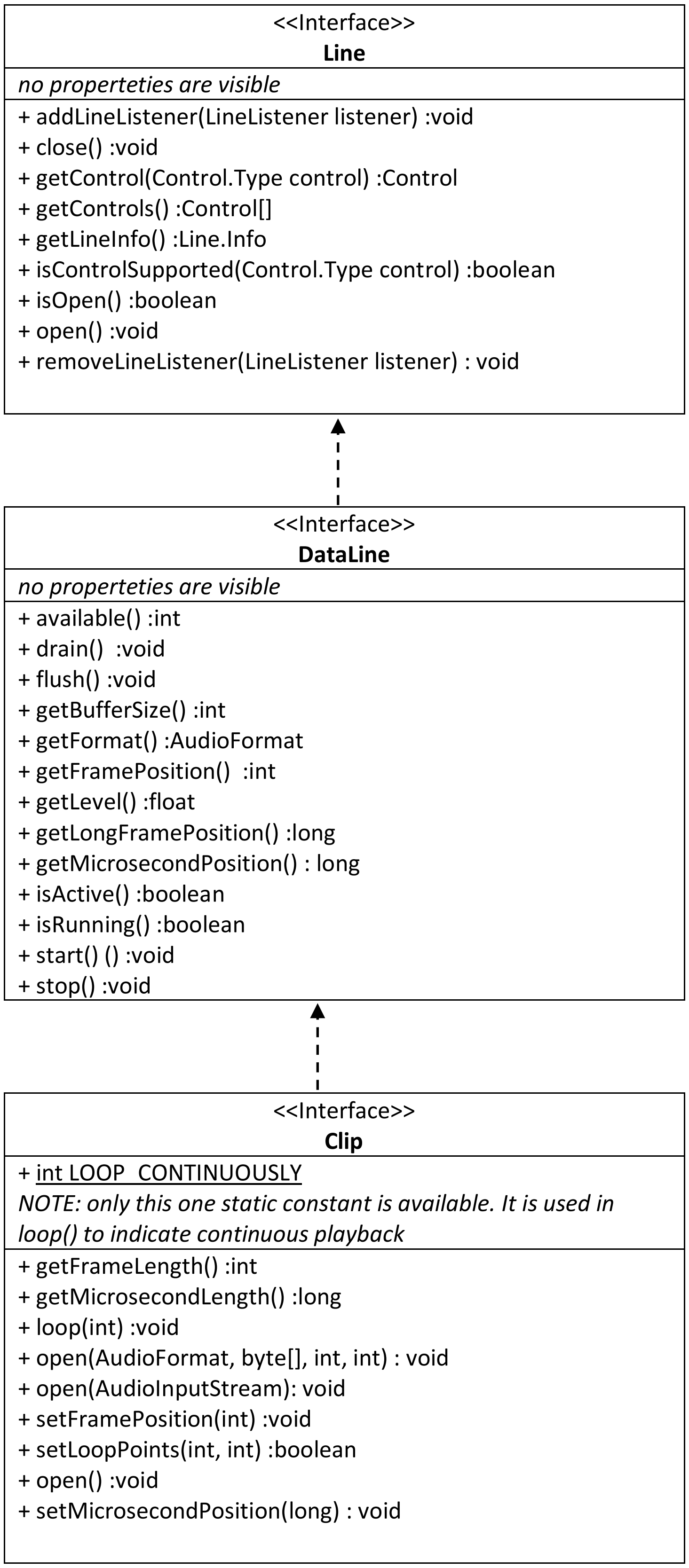
First, the file is set up as a File object, then associated with an *AudioInputStream*. If you compare this code to the code from the Sun Audio API in the last section, you will see that the old way was to create an *InputStream* for the file and associate it with an *AudioStream* object. This is replaced by an *AudioInputStream,* making the coding a little easier.

The Clip class object *myClip* is associated with the incoming AudioInputStream *audioIn* by the instruction ***`myClip.open(audioIn).`*** After that, we can use the methods in the Clip class to play sound files. The table below summarizes a few of these methods. *Clip* in the Java Sound API is an interface, implemented in the *javax.sound.sampled* package. More complete documentation for the Clip interface is available online at: http://docs.oracle.com/javase/7/docs/api/javax/sound/sampled/Clip.html

Method	Description
getFrameLength()	Obtains the media length in sample frames. Each sampling – as in 8K samples per second or 22K samples per second, etc. depending on the sampling rate – is one frame.
getFormat()	Obtains the current audio format (encoding, sample rate, number of channels, etc.) of the Clip object. It is returned as an *AudioFormat* object, which has a toString() method.
getMicrosecondLength()	Obtains the media duration in microseconds
loop(int count)	Starts looping playback from the current position. The parameter is the number of times the playback should repeat, so 0 means 1 playback; 1, means play and repeat once, etc. You can set the position (frame number) first using the setFramePosition(i) method, and the loop will start from there, but repeat the entire clip in each subsequent loop.
open(AudioFormat format, byte[] data, int offset, int bufferSize)	Opens the clip, meaning that it should acquire any required system resources and become operational.
open(AudioInputStream stream)	Opens the clip with the format and audio data present in the provided audio input stream.
setFramePosition(int frames)	Sets the media position in sample frames.
setLoopPoints(int start, int end)	Sets the first and last sample frames that will be played in the loop.
setMicrosecondPosition(long microseconds)	Sets the media position in microseconds.
start()	Starts playing a clip.
stop()	Stops playing a clip. This is a bit tricky to use, and works best with multi-threaded applications, which we will see later in the course.

The UML diagram on the next page shows the relationship among the *Clip* interface, the *DataLine* Interface, and the *Line* Interface. All three of these are among the many interfaces specified in the *Java Sound API*. In the *javax.sound.sampled* package they are implemented as classes. *Clip* extends *DataLine* which extends the *Line* interface. This means the methods from all three are implemented and may be used in the *Clip* class.

The data members of the Interfaces in the *Java Sound API* are mostly hidden, which is typical for most API specifications. Only data members that must be public are specified.



<<Interface>>
Line

no properteties are visible

+ addLineListener(LineListener listener) :void
+ close() :void
+ getControl(Control.Type control) :Control
+ getControls() :Control[]
+ getLineInfo() :Line.Info
+ isControlSupported(Control.Type control) :boolean
+ isOpen() :boolean
+ open() :void
+ removeLineListener(LineListener listener) : void

<<Interface>>
DataLine

no properteties are visible

+ available() :int
+ drain() :void
+ flush() :void
+ getBufferSize() :int
+ getFormat() :AudioFormat
+ getFramePosition() :int
+ getLevel() :float
+ getLongFramePosition() :long
+ getMicrosecondPosition() : long
+ isActive() :boolean
+ isRunning() :boolean
+ start() () :void
+ stop() :void

<<Interface>>
Clip

+ <u>int LOOP_CONTINUOUSLY</u>
NOTE: only this one static constant is available. It is used in loop() to indicate continuous playback

+ getFrameLength() :int
+ getMicrosecondLength() :long
+ loop(int) :void
+ open(AudioFormat, byte[], int, int) : void
+ open(AudioInputStream): void
+ setFramePosition(int) :void
+ setLoopPoints(int, int) :boolean
+ open() :void
+ setMicrosecondPosition(long) : void

Line, *DataLine*, and *Clip* are all Interfaces specified in the Java Audio API.

In the *javax.sound.sampled* package, Clip is a class extending the DataLine class which extends the Line class.

Line and *DataLine* may be used with streaming audio input. *Clip* extends those playing capabilities to recorded sound clips.

Programming Example – The Java Sound API Demo

The Java Sound API is quite extensive. A demonstration program showcasing some of its features can be downloaded from: http://www.oracle.com/technetwork/java/index-139508.html You should download the file, unzip it, then try the JAR file or the HTML file to see what some of the packages that implement the API can do. The source code is included with the download, but it is quite extensive, in several different files.

Our purpose here is not to learn about all the details of sound in Java programming but just to learn enough about it to see how APIs work in Java programming. The code in this section and the assignments at the end of the chapter help us to do this. More information is available online for those who wish to do more work with sound in Java programming:

- A Java Sound API tutorial is online at:
  http://docs.oracle.com/javase/tutorial/sound/index.html

- A list of all of the Java Sound API tutorials is online at:
  http://docs.oracle.com/javase/tutorial/sound/TOC.html

17.5 Using a Third Party API Implementation – The *FreeTTS* Text-To-Speech API

So far we have seen two sound APIs that are part of the Core Java API – the deprecated Sun Audio API and the newer Java Sound API. They do not require us to download implementing libraries or place additional JAR files with in NetBeans projects to use the classes specified in the API. We only need to include the proper *import* statements in our code to make the classes available to our code.

Official Java APIs not in the core, and most third-party APIs, usually require us to add a library or set of libraries with JAR files containing additional Java classes in our Java project. These classes are required to make the features of the API implementation accessible to code in classes we write.

It can be difficult and cumbersome to do this manually – copying libraries to the correct locations, changing settings such as the *classpath* variable, and entering lengthy commands at a system prompt to compile and run our code or to create working jar files that access the APIs from our code. It is a little easier to do this in IDEs such as NetBeans or Eclipse.

In this section we will use the *FreeTTS* third party API implementation to learn how to use an API that requires us to make external library files available to our software. First we will learn a little about what the API is and how it works, then we will learn to write code that uses it to make our computer speak as it reads text from a String variable or from a text data file.

A Text to Speech Demo

Text to speech is the ability of software to read text data and speak the text out loud in a human voice just as a person reading the text might do. It creates voice audio output from text.

The zipped file **TTSDemo** is included with the files for this chapter. **TTSDemo** contains a JAR file with Java software which will recite speech from text, along with an ***lib*** folder that has a library of JAR files with the classes needed to make the demo work. In this demo, the text is hard coded into the program.

In this demo, the computer recites Edison's historic first recording of *Mary had a little lamb*. It is not recorded sound – it is the computer reading, synthesizing, and pronouncing the text.

STEP 1.

Download the file **TTSDemo** from the files for this chapter. It is a large file because of the libraries needed for the synthesized speech. There are more sophisticated ways to use the API in a manner that results in smaller files, but they are beyond the scope of this chapter.

STEP 2.

Unzip the folder. Find the file **TTSDemo.jar** inside the unzipped folder. It should be in the same folder that contains the lib sub-folder

STEP 3.

Run this file to hear the demo. You will need to have speakers or headphones and the audio on your computer turned on to run the demo.

The program was written using the *FreeTTS* text to speech API. Text to speech is not recorded sound. Sophisticated software reads the text and feeds it to a text to speech engine, which synthesizes sound from the syllables in the text. A **text to speech engine** is software that converts text to sound mimicking a human voice. Each text to speech engine can read certain languages and can pronounce the sound using particular voices.

In this example, the *FreeTTS* software reads the English language and pronounces the sound is a US male voice. We will use the same voice for all of the text to speech examples and assignments in the chapter. Remember, our primary purpose is to learn about using APIs. Learning to import and use other voices is beyond the scope of this chapter.

FreeTTS and the Java Speech API

FreeTTS is an implementation of the Java Speech API. The Java Speech API was developed jointly by *Sun Microsystems, Apple, AT&T, Dragon Systems, Inc., IBM, Novell, Philips Speech Processing*, and *Texas Instruments*. It is only an interface standardizing how *text to speech* and *speech recognition* software should interface with Java programs. Neither Sun nor Oracle have produced libraries implementing the API. It was intended as a development standard for third party developers. More information about the Java Speech API is online at: http://www.oracle.com/technetwork/java/jsapifaq-135248.html

The text to speech part of the Java Speech API is implemented by *FreeTTS*, an open source text to speech library for Java based on the *Festival Speech Synthesis System* developed by the *Centre for Speech Technology Research (CSTR)* at the University of Edinburgh and the *FestVox* Project at Carnegie Mellon University. The speech engine in *FreeTTS* is *Flite,* a small run-time speech synthesis engine developed at Carnegie Mellon University.

There are many add-ons and libraries that can be used with *FreeTTS* for other languages and other voices, but we will keep our use of the software simple, using just the English language and the voice named *Kevin16*, a standard US male voice that is included with the standard *FreeTTS* software download.

Programming Exercise - Creating Your Own TTS Demo

In this exercise we will create a *FreeTTS* demo similar to the one used in the exercise above. Most of the code to do this is included in the Netbeans project file for this exercise, but the *FreeTTS* library is missing from the project. You will:

- download and install the FreeTTS software needed to include text to speech in your Java code,
- make the necessary libraries available to your NetBeans project,
- then get the software to run properly.
- You can also customize what the demo says.

The *FreeTTS* library JAR files are needed to use FreeTTS in our software. A Website with extensive information about FreeTTS, including demos of more sophisticated uses of FreeTTS, is online at: http://freetts.sourceforge.net . We can download just the binary file we need from: http://sourceforge.net/projects/freetts/files/ We will download and unzip *freetts-1.2.2-bin.zip*, which has the files we need to make this exercise work.

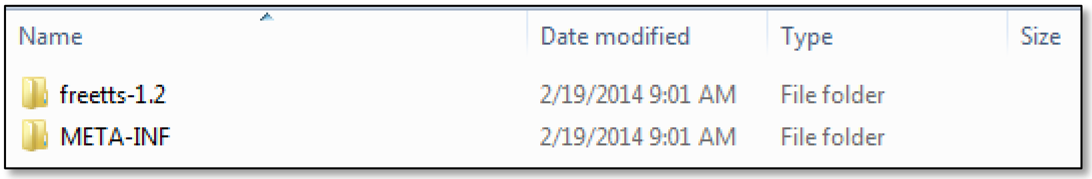
Once we download and install the necessary library of FreeTTS JAR files in NetBeans, it will be available to other NetBeans projects that need to use it. We only need to download and install the library in NetBeans once, then we can use it whenever we want to do so.

STEP 1.

Download and save the file ***freetts-1.2.2-bin.zip*** from the FreeeTTS website:
http://sourceforge.net/projects/freetts/files/latest/download?source=files
It is a 12 Mb file. Make sure you save it someplace where it will be easy to find.

STEP 2.

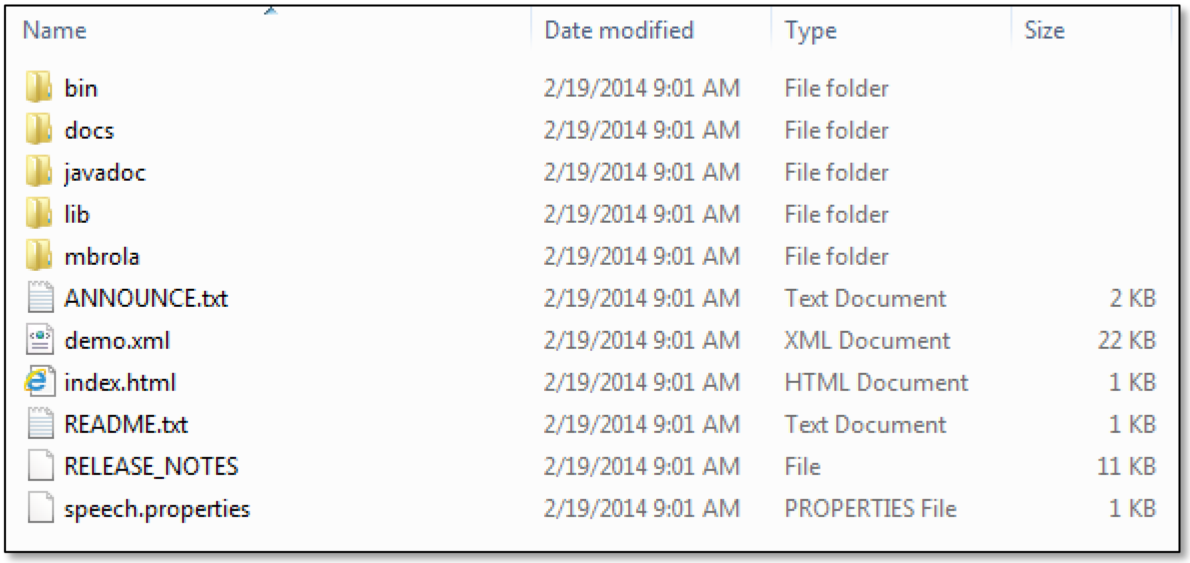
Unzip the file. Again, make sure the unzipped file is in a location that you can easily find. After you unzip the file, open the folder *freetts-1.2.2-bin.* It should contain two folders, as shown below:



Name	Date modified	Type	Size
freetts-1.2	2/19/2014 9:01 AM	File folder	
META-INF	2/19/2014 9:01 AM	File folder	

STEP 3.

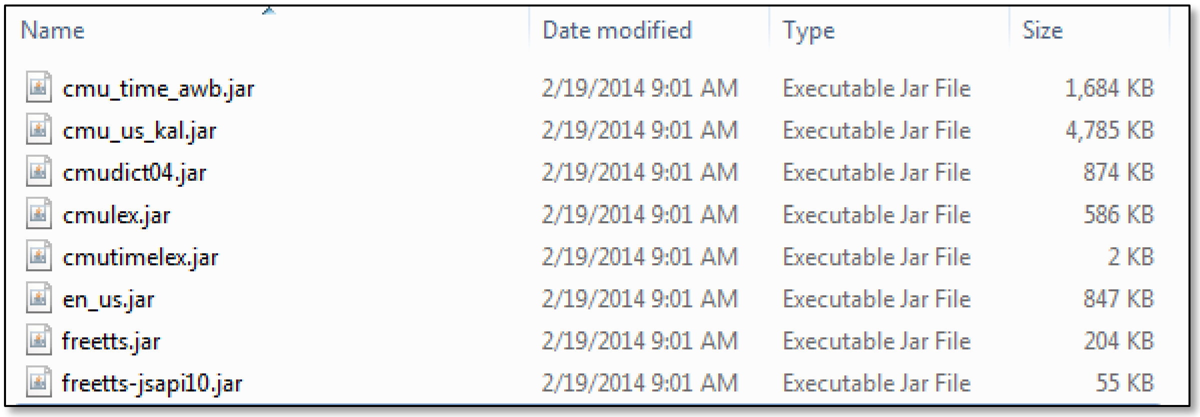
Open the folder *freetts-1.2*. It contains the following folders:



Name	Date modified	Type	Size
bin	2/19/2014 9:01 AM	File folder	
docs	2/19/2014 9:01 AM	File folder	
javadoc	2/19/2014 9:01 AM	File folder	
lib	2/19/2014 9:01 AM	File folder	
mbrola	2/19/2014 9:01 AM	File folder	
ANNOUNCE.txt	2/19/2014 9:01 AM	Text Document	2 KB
demo.xml	2/19/2014 9:01 AM	XML Document	22 KB
index.html	2/19/2014 9:01 AM	HTML Document	1 KB
README.txt	2/19/2014 9:01 AM	Text Document	1 KB
RELEASE_NOTES	2/19/2014 9:01 AM	File	11 KB
speech.properties	2/19/2014 9:01 AM	PROPERTIES File	1 KB

STEP 4.

Open the ***lib*** folder. This folder has the JAR files we need:



Name	Date modified	Type	Size
cmu_time_awb.jar	2/19/2014 9:01 AM	Executable Jar File	1,684 KB
cmu_us_kal.jar	2/19/2014 9:01 AM	Executable Jar File	4,785 KB
cmudict04.jar	2/19/2014 9:01 AM	Executable Jar File	874 KB
cmulex.jar	2/19/2014 9:01 AM	Executable Jar File	586 KB
cmutimelex.jar	2/19/2014 9:01 AM	Executable Jar File	2 KB
en_us.jar	2/19/2014 9:01 AM	Executable Jar File	847 KB
freetts.jar	2/19/2014 9:01 AM	Executable Jar File	204 KB
freetts-jsapi10.jar	2/19/2014 9:01 AM	Executable Jar File	55 KB

The folder will contain additional files, but it is the JAR files in the *lib* folder that are important for our purposes.

We now need to open the NetBeans project and set up the *FreeTTS* library in NetBeans.

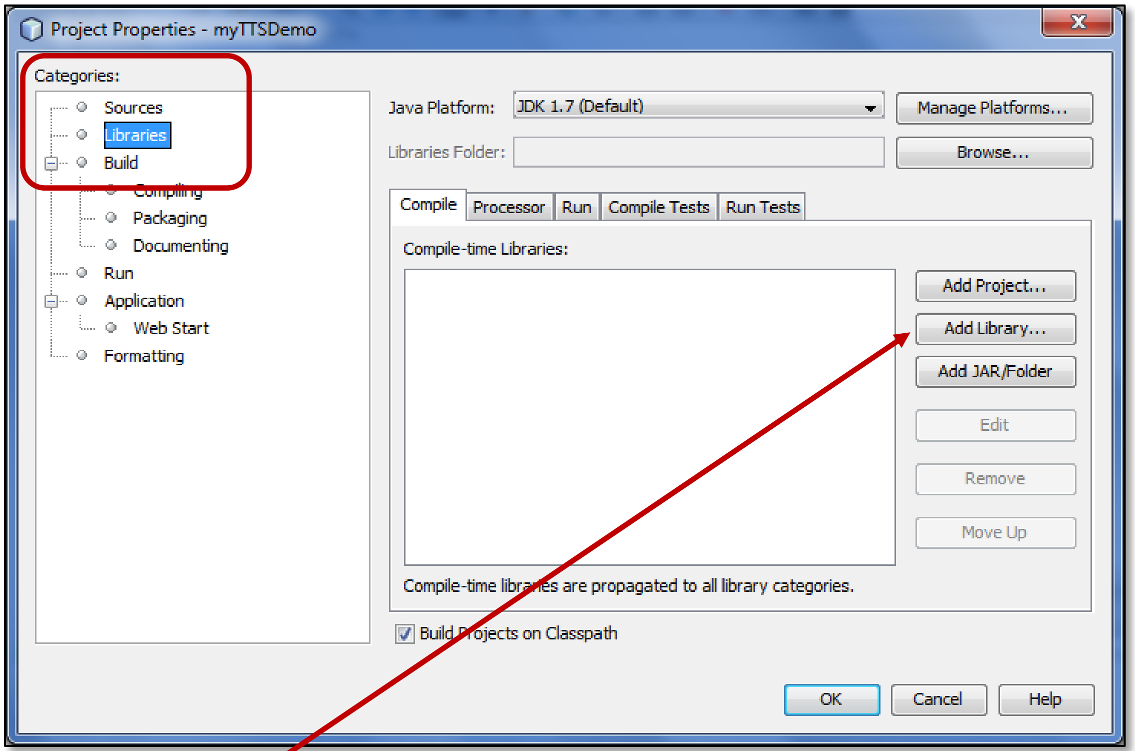
STEP 5.

Download the zipped NetBeans project folder **MyTTSDemo.zip** included with the files for this chapter. Unzip the file and open the **MyTTSDemo** project in NetBeans.

The project will not run. You can attempt to test or run the file, but you will get error messages. We need to set up the *FreeTTS* library in NetBeans before the project will work.

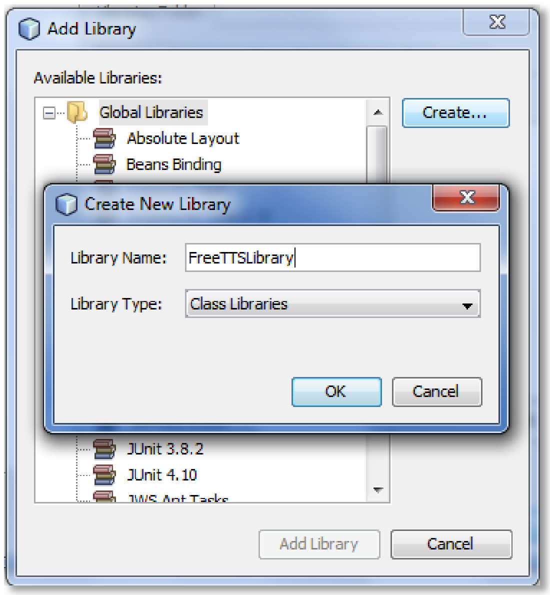
STEP 6.

Right-click the ***MyTTSDemo*** project's node in the NetBeans project window, then select ***Properties*** at the bottom of the menu that appears. The Project Properties Window will open. Select the **Libraries** category, circled in the image below.



STEP 7.

Click the **Add Library** button, then click the **Create** button on the *Add Library* window that appears. A *Create New Library* window will appear on top of the *Add library* window.

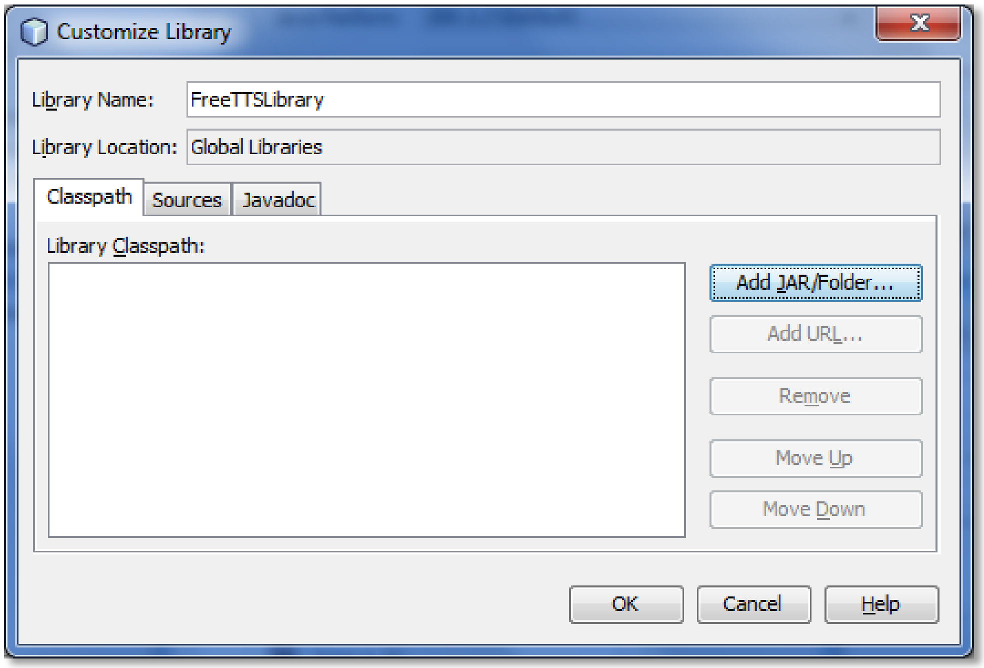


We will create a new Library with the JAR files needed to use FreeTTS in our Java projects in NetBeans. We will name the library *FreeTTS* and add the JAR files to the library.

STEP 8.

Type **FreeTTSLibrary** as the *Library Name* in the *Create New Library* window, then click the **OK** button.

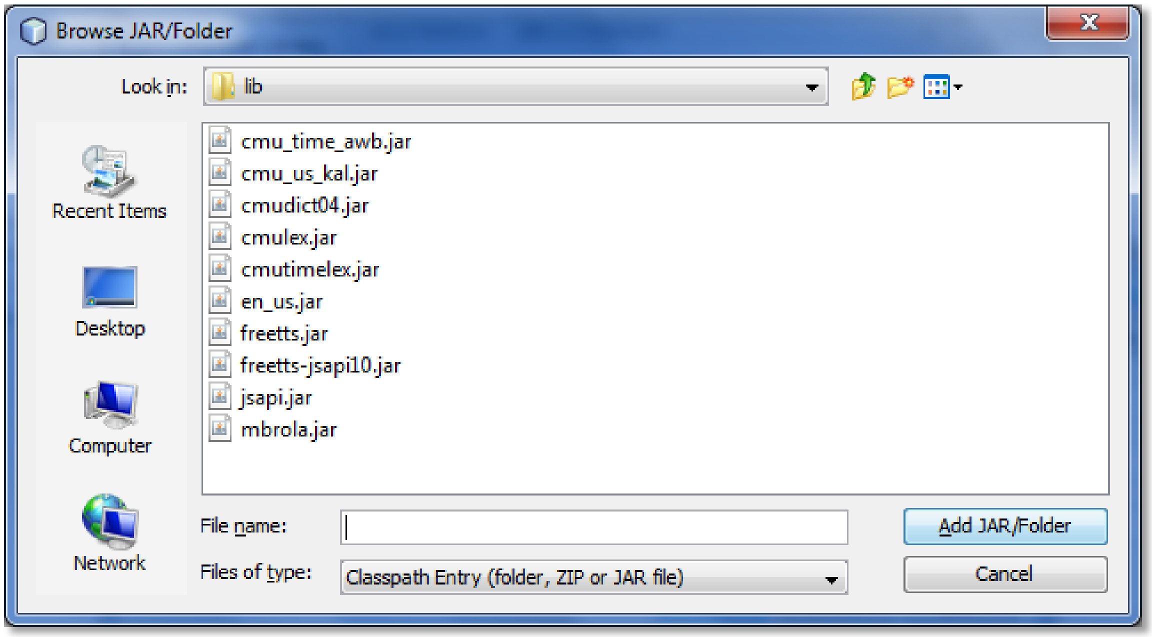
A *Customize Library* window will now appear.



STEP 9.

Click the **Add JAR/Folder...** button in the *Customize Library* window, then navigate to the **lib** folder with the FreeTTS Jar files – this is the same *lib* file you opened in step 4, above.

STEP 10.



cmu_time_awb.jar
cmu_us_kal.jar
cmudict04.jar
cmulex.jar
cmutimelex.jar
en_us.jar
freetts.jar
freetts-jsapi10.jar
jsapi.jar
mbrola.jar

Select all of the JAR files from the *lib* folder, then click the **Add JAR/Folder** button. Click the **OK** button when the *Customize Library* window reappears, then click the **Add Library** button when the *Add Library* window reappears.

The FreeTTS Jar files are now part of the *FreeTTS* library in NetBeans that will be available for use in all of your NetBeans projects.

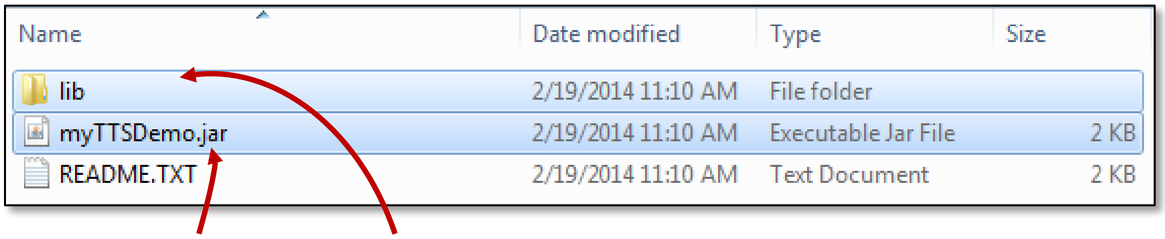
STEP 11.

Now the program should run. **Run the program** to test it, then you can customize what your demo says by changing the text that starts on line 23.

You can now create your own JAR file that demonstrates text to speech by using "Clean and Build".

STEP 12.

Clean and Build the project using the *Clean and Build* option from the *Run* menu. The JAR file will be in the project's *dist* folder, along with an *lib* folder containing the *FreeTTS* JAR files that are needed to make the program work. To distribute the program for others to use, you should zip and distribute the ***MyTTSDemo.jar*** file together with the ***lib*** folder. To run the program, the user should unzip the file then run ***MyTTSDemo.jar.***



Name	Date modified	Type	Size
lib	2/19/2014 11:10 AM	File folder	
myTTSDemo.jar	2/19/2014 11:10 AM	Executable Jar File	2 KB
README.TXT	2/19/2014 11:10 AM	Text Document	2 KB

The **MyTTSDemo.jar** file and the **lib** file must be in the same directory for the *MyTTSDemo.jar* file to work. They can be zipped together for distribution.
Don't forget to close the NetBeans project when you are done.

Cleaning Up after Clean and Build

The FreeTTS JAR library files that we downloaded are not part of a NetBeans project folder, they are stored with the NetBeans libraries. They are long – up to 9MB. Whenever you *Clean and Build* a text to speech project that uses the library, copies of the necessary JAR files are place in a the lib folder in the project's dist folder so they can be distributed with the software. This will add 9 Mb to the size of an otherwise relatively small NetBeans project folder. I suggest you delete the contents of the *dist* folder before you close your NetBeans project. The *dist* folder can always be re-created when needed simply by performing another *Clean and Build.*

Awareness of the unnecessary space taken up by libraries in *dist* folders is important. 20 project ***dist*** folders, for example, could unnecessarily take up close to 200 mb of disk space.

Once the *FreeTTS* library is available in NetBeans, it can be used to "speak" the text in any program. A *Voice* object needs to be set up, then the *Voice* class *speak()* method can be used to say almost any text. Here is the critical code from the *MyTTSDemo* project, above:

```
    // instantiate a Voice object named voice
    Voice voice;

    // set up a Voicemanager object and use it to link voice with a particular voice
    VoiceManager voiceManager = VoiceManager.getInstance();
    voice = voiceManager.getVoice("kevin16");

    // load the selected voice
    voice.allocate();

    // begin speaking the text   -- text can be any String or String variable.
    voice.speak(text);
```

Programming Example - Reading and Speaking Text from a Data file

The NetBeans project *SpeechFileDemo,* included as a zipped file with the files for this chapter, has an example of a program with two methods – the first method reads lines from a text file into an array of Strings, the second method "speaks" the lines one by one. You should **download and run the program**, then examine the code, which is shown and discussed below.

Here is the code for the software:

```
import com.sun.speech.freetts.Voice;
import com.sun.speech.freetts.VoiceManager;
import java.util.Scanner;

public class SpeechFileDemo {

    public static void main(String[] args) throws Exception {

    String[] quotes = new String[200];  // an array to hold movie quotes
    int count;                          // the number of lines (array elements)

        // call a method to read data into quotes[] line by line and return count
        count = readLines(quotes);

        // call a method to speak thel ines in the array, one by one.
        speakLines(quotes, count);

    } // end main()
/**************************************************/

     public static int readLines(String[] lines) throws Exception
    {
        int count = 0; // number of array elements with data

        // Create a File class object linked to the name of the file to read
        java.io.File quotes = new java.io.File("mary.txt");

        // Create a Scanner named infile to read the input stream from the file
        Scanner infile  = new Scanner(quotes);
    /* This while loop reads lines of text into an array. it uses a Scanner class
          * boolean function hasNextLine() to see if there another line in the file.
          */
        while ( infile.hasNextLine() )
        {
            // read a line and put it in an array element
            lines[count] = infile.nextLine();
            count++;  // increment the number of array elements with data

        } // end while

        infile.close();
        return count;    // returns the number of items used in the array.

    } // end readLines()
/**************************************************/

    public static void speakLines(String[] lines, int count) {

        // instantiate a Voice object named voice
        Voice voice;

        // set up a VoiceManager object and use it to link voice with a particular voice
        VoiceManager voiceManager = VoiceManager.getInstance();
        voice = voiceManager.getVoice("kevin16");

        // load the selected voice
        voice.allocate();

        // in a loop, say the lines
        for( int i = 0; i< count; i++)
        {
        System.out.println("line " + i + ": " + lines[i] );
        voice.speak(lines[i]);
        } // end for

    } // end speakLines()
}  // end class SpeechFileDemo
```

The main method simply calls two other methods – **readLines()**, which reads the text file into a String array line by line, and **speakLines()** which then speaks the lines in the array line by line. This should work with almost any text file.

Line 43 of the code has the file name: `java.io.File quotes = new java.io.File("mary.txt");`
In addition to ***mary.txt,*** two other text files are included in the project folder:

- **quotes.txt** has about 50 quotes from famous movies. They are from the American Film Institute's top 100 movie quotes of all time, online at: http://www.afi.com/100years/quotes.aspx
- **hamlet.txt** has the famous soliloquy from Act 3, scene 2 of Shakespeare's *Hamlet*. Even though the text is old and spoken by a computer, you can hear the rhythm and meter in Shakespeare's writing.

You can change the file name in the code to try these two files, or create or download your own text files. The code is able to read almost any plain text file. You should try running the program with the other data files.

In other programs, after you set up a Voice object as in this program, you can use the Voice class *speak()* method to say text almost anyplace that a *print()* or *println()* method can be used.

Many other APIs can be used in the same way that FreeTTS was used in this chapter – first, set up the necessary JAR files in the NetBeans libraries, then you can use classes from those files in your own Java code.

The use of libraries that implement APIs extend the Java language by letting us use code written by experts to add sophisticated functionality to our software. This is one of the reasons for the success and popularity of the Java programming language. Good Java programmer should know how to use packages that implement APIs in Java applications that they develop.

Avoiding Out of Java Heap Space Errors with Big Data or Big Libraries

Many libraries and large datasets need more memory than was allocated for a NetBeans project. This can easily happen with large multimedia files, such as long sound files.

You can change the amount of memory available for a Java program by changing its Virtual memory (VM property) in NetBeans, as follows:

With the project open in NetBeans.

1. **Right-click** the project node in the *Projects* window, then select **properties** from the bottom of the menu that appears.
2. Select "**Run**" in the Categories pane.
3. Enter an argument such as **–Xmx512m** in the "**VM Options**" text box. The number controls how much memory is available for the project. For example, putting -*Xmx512m* in the *VM Options* text box allocates a maximum of 512 Mb of Java heap space for your software.

For more about Java heap space, see the blog post by Javin Paul, a professional Java software developer from India. English is his second language, but the post is clear and informative. It is online at:
http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html

Chapter Review

Section 1 reviewed the concept of interfaces in java, and discussed Application Programming Interfaces APIs, and libraries in Java that implement them. We saw that the term *API* is often used to refer to the library as well as the interface it implements. Key terms included: interface, software interface, Application Programming Interface (API), Core Java API, Java SE, Java EE, Java ME, third party Java API, deprecated, MIDI, and Java Community Process

Section 2 briefly discussed concepts of sound in computer programming.

Section 3 showed an example of a deprecated API implementation, the *Sun Audio API*, and the error messages its use generates. The section used an example of code that plays a wav sound file.

Section 4 discussed and demonstrated the use of the *javax.sound.sampled*package, which implements part of the *Java Sound API*, included in the Core Java API. The section used an example of code that plays a wav sound file, similar to the code shown for the deprecated *Sun Audio API* in section 1. A link to a demonstration of the broader capabilities of the *Java Sound API* was included.

Section 5 showed us how to use a third party API implementation but adding files to NetBeans libraries of JAR files for classes in the API implementation. The *FreeTTS* implementation of the Java Speech API text to speech classes was used as an example. Text to speech programs that read from String variables and that read from a data file were included in customizable exercises.

Chapter Questions

1. What does the term interface mean as a reserved word in the Java language? How does it compare with the concept of a *software interface*?
2. What is the technical meaning of the term *Application Programming Interface (API)*? What do people also often use it to refer to?
3. What is the difference between an API that is part of the Core Java API and an official API that is not part of the Core Java API? Give several examples of API that are part of the core and several that are not.
4. What do we usually need to include in our code to use Core Java API packages and classes? What do we also often need to do to use APIs that are not part of the Core Java API?
5. How do languages like Java that were designed to be serious production languages benefit from the presence and use of APIs?
6. What are the differences among Java SE, Jav EE, and Java ME? Give several examples of the kind of software that would be developed using each of them.
7. What Core APIs have you used in your programming in CSCI 111 and CSCI 112?
8. Who developed the original Java Internet Foundation Classes? What did it evolve into? What parts of its modern equivalent are used today for GUI programming?
9. When will Java 8 be released?

10. How is the Java Community Process related to API development? How is this beneficial? Why have some developers moved away from the Java Community Process in recent years?
11. Give several examples of third party APIs, describing generally what each is used for and who developed the implementation.
12. What are some of the questions we should ask related to the acquisition, use and distribution of third party APIs and software based on them?
13. What is MIDI? Who maintains the MIDI standards? What is the difference between sampled audio files like thos we in the WAV format and MIDI files?
14. What are the major differences between the Sun Audio API and the Java Sound API? Which is better to use for playback of recorded audio, and why?
15. What happens when we test a program that uses a deprecated API in NetBeans? What could eventually happen to software that uses deprecated APIs? Why do APIs become deprecated?
16. What two class of objects in addition to the File object are needed to read audio data from a sampled audio file using the Sun Audio API? how does this compare to what's need using the Java Sound API?
17. What two parts packages in the Core Java API implement parts of the Java Sound API, and what does each do? What sampled audio file formats can be read in using the *javax.sound.sampled* package?
18. What class in the *javax.sound.sampled* package is used to play an audio sound clip? What are some of its methods and what do they do?
19. What does the FreeTTS API implementation do? What API does it implement part of? Who developed the API it is based on? What two universities contributed to its development?
20. What is necessary to include with a JAR file for distribution of a program that uses FreeTTS technology? Why is it a good idea to clear the files in a NetBeans project's *dist* file after creating and copying a distributable software package and before closing the NetBeans project?

Chapter Exercises

1. Create a JFrame with several buttons, each button with a picture of a famous movie star. When the user clicks a picture, the computer will recite (text to speech) a line said by that movie star, or play an audio clip of the start reciting a line. Your program should use text to speech or audio clips, but not both.

2. Create a Juke box like interface using a JFrame based GUI that can play five different audio clips. Your GUI software should look attractive and should include buttons that play each clip. In the interest of smaller file size, avoid using long audio clips in your project.

3. Rewrite the "I am thinking of a number between 1 and 100…" program that was used in CSCI 111 and earlier in this course to use text to speech to make the program talk to the user in addition to producing printed output.

4. Rewrite the "I am thinking of a number between 1 and 100…" program that was used in CSCI 111 and earlier in this course to play sound effect clips that go along with the printed output. You should find and include short sound clips to be played when the guess is too low, too high, or correct.

5. Create a program that is the equivalent of verbal flash cards to teach young children arithmetic. Your program should randomly pick two one digit integers, for example, A and B, then using text to speech build a text string that says something like *" How much is " + A + " plus " + B +"?"* (The question mark at the end will affect the way the String is pronounced.) Your program should ask the user to enter an answer, then verbally tell the student that it is right, or tell the student something like *"Sorry, the answer is" + sum* +"." Your program should give the use five different randomly generated problems, and should verbally tell the user something like "you got" 4 out of 5 correct." when the program ends. Your program should include a short spoken greeting and intro at the beginning.

6. Write a program using text to speech that counts from 1 to 25 and tells us if each number is a multiple of 3, a multiple of 5, a multiple of 7, a multiple of 11, or a Fibonacci number. The messages should be displayed on the screen and spoken.

7. Create a program that includes a trivia game with five spoken questions. Your program should speak the question, then ask the user to enter an answer. It should tell the user by voice if the user got the correct answer, or tell the user if the answer was incorrect and what the correct answer is. At the end of the game, your program should tell the user the user's score. You should use questions that have clear short answers, such as "What was the last name of the 38th president of the United States?", or "What is the official two-letter abbreviation for the last U.S. state to host the Summer Olympics?"

8. Create a "*name that tune*" type program with five questions. Your program should play a short sound clip, then on the screen, it should ask the user to enter the name of the song or pick the song from a menu. The exact format of the game and the program is up to you, but the user should be given five sound clips to identify, one at a time, with a score at the end.

9. Create a JFrame GUI that acts like a Foley machine for a play or radio production. A Foley artist is a sound effects specialist who makes sound effects for cartoons, radio segments, etc. Your GUI should be a machine that the Foley artist can use to generate sound effects. It should have labels with buttons that each play a short sound effect. The user should be able to play the GUI like an instrument by clicking the buttons to make various humorous sounds. The design of the instrument is up to you, but don't make it too complicated. There are many free sound effects WAV files available on the Web on sites such as: http://www.grsites.com/archive/sounds/category/23.

10. The *JFugue* API can be used to play music by setting up a String that tells the computer what notes to play. More information about *JFuge* is online at http://www.jfugue.org. Create an application that plays a piece of music (not too long) using *JFugue*. You will need to learn how to use the software to complete this exercise. *JFugue* is supposed to be easy to use, but you will need to visit the JFugue *Getting Started* page, online at: http://www.jfugue.org/howto.html.