Appendix C

# Odds and Ends

This book's first edition presented several APIs that are not officially included in the second edition: Internationalization, Preferences, References, and Reflection. My main reasons for removing these APIs from the second edition: simplify this book and save space for new content that's probably more useful to potential Android application developers.

I recognize that you probably want this missing content, and so I've created this appendix to give it to you. Appendix C presents upgraded Internationalization, Preferences, References, and Reflection API content. Because I briefly mentioned classloaders and native code in the book, it also introduces you to classloaders and the Java Native Interface (for interacting with native code)—and more.

## Class and the Reflection API

Chapter 4 presented two forms of runtime type identification (RTTI). Java's Reflection API offers a third RTTI form in which applications can dynamically load and learn about loaded classes, interfaces, enums, and annotation types. The API also lets applications instantiate classes, call methods, access fields, and so on. This form of RTTI is known as *reflection* or *introspection*.

Chapter 6 presented a `StubFinder` application that used part of the Reflection API to load a class and identify all of the loaded class's public methods that are annotated with `@Stub` annotations. This tool is one example where using reflection is beneficial. Another example is the *class browser*, a tool that enumerates the members of a class.

■ **CAUTION:** Reflection should not be used indiscriminately. Application performance suffers because it takes longer to perform operations with reflection than without reflection. Also, reflection-oriented code can be harder to read, and the absence of compile-time type checking can result in runtime failures.

The `java.lang` package's `Class` class is the entry point into the Reflection API, whose types are mainly stored in the `java.lang.reflect` package. `Class` is generically declared as `Class<T>`, where `T` identifies the class, interface, enum, or annotation type that's being modeled by the `Class` object. `T` can be replaced by `?` (as in `Class<?>`) when the type being modeled is unknown.

Table C–1 describes some of `Class`'s methods.

*Table C–1.* `Class` *Methods*

| Method | Description |
|---|---|
| `static Class<?> forName(String typename)` | Return the `Class` object associated with `typename`, which must include the type's qualified package name when the type is part of a package (`java.lang.String`, for example). If the class or interface type has not been loaded into memory, this method takes care of *loading* (reading the classfile's contents into memory), *linking* (taking these contents and combining them into the runtime state of the virtual machine so that they can be executed), and *initializing* (setting class fields to default values, running class initializers, and performing other class initialization) before returning the `Class` object. This method throws `java.lang.ClassNotFoundException` when the type cannot be found, `java.lang.LinkageError` when an error occurs during linkage, and `java.lang.ExceptionInInitializerError` when an exception occurs during a class's static initialization. |
| `Annotation[] getAnnotations()` | Return an array (that's possibly empty) containing all annotations that are declared for the class represented by this `Class` object. |
| `Class<?>[] getClasses()` | Return an array containing `Class` objects representing all public classes and interfaces that are members of the class represented by this `Class` object. This includes public class and interface members inherited from superclasses, and public class and interface members declared by the class. This method returns a zero-length array when this `Class` object has no public member classes or interfaces. This method also returns a zero-length array when this `Class` object represents a primitive type, an array class, or void. |
| `Constructor[] getConstructors()` | Return an array containing `java.lang.reflect.Constructor` objects representing all the public constructors of the class represented by this `Class` object. A zero-length array is returned when the represented class has no public constructors, this `Class` object represents an array class, or this `Class` object represents a primitive type or void. |
| `Annotation[] getDeclaredAnnotations()` | Return an array containing all annotations that are directly declared on the class represented by this `Class` object—inherited annotations are not included. The returned array might be empty. |

| Method | Description |
|---|---|
| `Class<?>[] getDeclaredClasses()` | Return an array of `Class` objects representing all classes and interfaces declared as members of the class represented by this `Class` object. This includes public, protected, default (package) access and private classes, and interfaces. This method returns a zero-length array when the class declares no classes or interfaces as members, or when this `Class` object represents a primitive type, an array class, or void. |
| `Constructor[] getDeclaredConstructors()` | Return an array of `Constructor` objects representing all constructors declared by the class represented by this `Class` object. These are public, protected, default (package) access, and private constructors. The returned array's elements are not sorted and are not in any order. If the class has a default constructor, it's included in the returned array. This method returns a zero-length array when this `Class` object represents an interface, a primitive type, an array class, or void. |
| `Field[] getDeclaredFields()` | Return an array of `java.lang.reflect.Field` objects representing all fields declared by the class or interface represented by this `Class` object. This array includes public, protected, default (package) access, and private fields, but excludes inherited fields. The returned array's elements are not sorted and are not in any order. This method returns a zero-length array when the class/interface declares no fields, or when this `Class` object represents a primitive type, an array class, or void. |
| `Method[] getDeclaredMethods()` | Return an array of `java.lang.reflect.Method` objects representing all methods declared by the class or interface represented by this `Class` object. This array includes public, protected, default (package) access, and private methods, but excludes inherited methods. The elements in the returned array are not sorted and are not in any order. This method returns a zero-length array when the class or interface declares no methods, or when this `Class` object represents a primitive type, an array class, or void. |
| `Field[] getFields()` | Return an array containing `Field` objects representing all public fields of the class or interface represented by this `Class` object, including those public fields inherited from superclasses and superinterfaces. The elements in the returned array are not sorted and are not in any order. This method returns a zero-length array when this `Class` object represents a class or interface that has no accessible public fields, or when this `Class` object represents an array class, a primitive type, or void. |

| Method | Description |
|---|---|
| `Method[] getMethods()` | Return an array containing `Method` objects representing all public methods of the class or interface represented by this `Class` object, including those public methods inherited from superclasses and superinterfaces. Array classes return all the public member methods inherited from the `java.lang.Object` class. The elements in the returned array are not sorted and are not in any order. This method returns a zero-length array when this `Class` object represents a class or interface that has no public methods, or when this `Class` object represents a primitive type or void. The class initialization method `<clinit>()` (see Chapter 3) isn't included in the returned array. |
| `int getModifiers()` | Return the Java language modifiers for this class or interface, encoded in an integer. The modifiers consist of the virtual machine's constants for `public`, `protected`, `private`, `final`, `static`, `abstract`, and `interface`; they should be decoded using the methods of class `java.lang.reflect.Modifier`.<br><br>If the underlying class is an array class, its `public`, `private`, and `protected` modifiers are the same as those of its component type. If this `Class` object represents a primitive type or void, its `public` modifier is always true, and its `protected` and `private` modifiers are always false. If this `Class` object represents an array class, a primitive type, or void, its `final` modifier is always true and its `interface` modifier is always false. The values of its other modifiers are not determined by this specification. |
| `String getName()` | Return the name of the class represented by this `Class` object. |
| `Package getPackage()` | Return a `java.lang.Package` object that describes the package in which the class represented by this `Class` object is located, or return null when the class is a member of the unnamed package. I discussed the `Package` class in Chapter 7. |
| `Class<? super T> getSuperclass()` | Return the `Class` object representing the superclass of the entity (class, interface, primitive type, or void) represented by this `Class` object. When the `Class` object on which this method is called represents the `Object` class, an interface, a primitive type, or void, null is returned. When this object represents an array class, the `Class` object representing the `Object` class is returned. |
| `boolean isAnnotation()` | Return true when this `Class` object represents an annotation type. If this method returns true, `isInterface()` also returns true because all annotation types are also interfaces. |
| `boolean isEnum()` | Return true if and only if this class was declared as an enum in the source code. |
| `boolean isInterface()` | Return true when this `Class` object represents an interface. |

| Method | Description |
|---|---|
| `T newInstance()` | Create and return a new instance of the class represented by this `Class` object. The class is instantiated as if by a `new` expression with an empty argument list. The class is initialized when it has not already been initialized. This method throws `java.lang.IllegalAccessException` when the class or its noargument constructor isn't accessible; `java.lang.InstantiationException` when this `Class` object represents an abstract class, an interface, an array class, a primitive type, or void, or when the class doesn't have a noargument constructor (or when instantiation fails for some other reason); and `ExceptionInInitializerError` when initialization fails because the object threw an exception during initialization. |

Table C–1's description of the `forName()` method reveals one way to obtain a `Class` object. This method loads, links, and initializes a class or interface that isn't in memory and returns a `Class` object representing the class or interface. Listing C–1 demonstrates `forName()` and additional methods described in this table.

*Listing C–1. Using reflection to decompile a type*

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

public class Decompiler
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java Decompiler classname");
         return;
      }
      try
      {
         decompileClass(Class.forName(args[0]), 0);
      }
      catch (ClassNotFoundException cnfe)
      {
         System.err.println("could not locate " + args[0]);
      }
   }

   static void decompileClass(Class<?> clazz, int indentLevel)
   {
      indent(indentLevel * 3);
      System.out.print(Modifier.toString(clazz.getModifiers()) + " ");
      if (clazz.isEnum())
         System.out.println("enum " + clazz.getName());
      else
      if (clazz.isInterface())
      {
         if (clazz.isAnnotation())
            System.out.print("@");
         System.out.println(clazz.getName());
      }
      else
         System.out.println(clazz);
      indent(indentLevel * 3);
      System.out.println("{");
      Field[] fields = clazz.getDeclaredFields();
      for (int i = 0; i < fields.length; i++)
      {
         indent(indentLevel * 3);
         System.out.println("   " + fields[i]);
      }
      Constructor[] constructors = clazz.getDeclaredConstructors();
      if (constructors.length != 0 && fields.length != 0)
         System.out.println();
      for (int i = 0; i < constructors.length; i++)
      {
         indent(indentLevel * 3);
         System.out.println("   "+constructors[i]);
      }
      Method[] methods = clazz.getDeclaredMethods();
      if (methods.length != 0 &&
          (fields.length != 0 || constructors.length != 0))
         System.out.println();
      for (int i = 0; i < methods.length; i++)
      {
         indent(indentLevel * 3);
         System.out.println("   "+methods[i]);
      }
      Method[] methodsAll = clazz.getMethods();
      if (methodsAll.length != 0 &&
          (fields.length != 0 || constructors.length != 0 ||
           methods.length != 0))
         System.out.println();
      if (methodsAll.length != 0)
      {
         indent(indentLevel * 3);
         System.out.println("   ALL PUBLIC METHODS");
         System.out.println();
      }
      for (int i = 0; i < methodsAll.length; i++)
      {
         indent(indentLevel * 3);
         System.out.println("   "+methodsAll[i]);
      }
      Class<?>[] members = clazz.getDeclaredClasses();
      if (members.length != 0 && (fields.length != 0 ||
          constructors.length != 0 || methods.length != 0 ||
          methodsAll.length != 0))
         System.out.println();
      for (int i = 0; i < members.length; i++)
         if (clazz != members[i])
         {
            decompileClass(members[i], indentLevel + 1);
            if (i != members.length - 1)
               System.out.println();
         }
      indent(indentLevel * 3);
      System.out.println("}");
   }

   static void indent(int numSpaces)
   {
      for (int i = 0; i < numSpaces; i++)
         System.out.print(' ');
   }
}
```

Listing C–1 presents the source code to a decompiler tool that uses reflection to obtain information about this tool's solitary command-line argument, which must be a Java reference type (e.g., a class). The decompiler lets you output the type and name information for a class's fields, constructors, methods, and nested types. It also lets you output the members of interfaces, enums, and annotation types.

After verifying that one command-line argument has been passed to this application, `main()` calls `forName()` to try to return a `Class` object representing the class or interface identified by this argument. When successful, the returned object's reference is passed to `decompileClass()`, which decompiles the type. (`decompileClass()` is recursive in that it invokes itself for every encountered nested type.)

`forName()` throws an instance of the checked `ClassNotFoundException` class when it cannot locate the class's classfile (perhaps the classfile was erased before executing the application). It also throws `LinkageError` when a class's classfile is malformed and `ExceptionInInitializerError` when a class's static initialization fails.

**NOTE:** `ExceptionInInitializerError` is often thrown as the result of a class initializer throwing an unchecked exception. For example, the class initializer in the following `FailedInitialization` class results in `ExceptionInInitializerError` because `someMethod()` throws `java.lang.NullPointerException`:

```
public class FailedInitialization
{
   static
   {
      someMethod(null);
   }

   public static void someMethod(String s)
   {
      int len = s.length(); // s contains null
      System.out.println(s + "'s length is " + len + " characters");
   }
}
```

Much of the printing code focuses on making the output look nice. For example, this code manages indentation, and only allows a newline character to be output to separate one section from another; a newline character isn't output unless content appears before and after the newline.

Compile Listing C–1 (`javac Decompiler.java`) and run this application with `java.lang.Byte` as the solitary command-line argument (`java Decompiler java.lang.Byte`). You will observe the following output, which provides insight into how `Byte` is implemented:

```
public final class java.lang.Byte
{
   public static final byte java.lang.Byte.MIN_VALUE
   public static final byte java.lang.Byte.MAX_VALUE
   public static final java.lang.Class java.lang.Byte.TYPE
   private final byte java.lang.Byte.value
   public static final int java.lang.Byte.SIZE
   private static final long java.lang.Byte.serialVersionUID

   public java.lang.Byte(byte)
   public java.lang.Byte(java.lang.String) throws java.lang.NumberFormatException

   public boolean java.lang.Byte.equals(java.lang.Object)
   public java.lang.String java.lang.Byte.toString()
   public static java.lang.String java.lang.Byte.toString(byte)
   public int java.lang.Byte.hashCode()
   public int java.lang.Byte.compareTo(java.lang.Byte)
   public int java.lang.Byte.compareTo(java.lang.Object)
   public byte java.lang.Byte.byteValue()
   public short java.lang.Byte.shortValue()
   public int java.lang.Byte.intValue()
   public long java.lang.Byte.longValue()
   public float java.lang.Byte.floatValue()
   public double java.lang.Byte.doubleValue()
   public static java.lang.Byte java.lang.Byte.valueOf(byte)
   public static java.lang.Byte java.lang.Byte.valueOf(java.lang.String) throws
java.lang.NumberFormatException
   public static java.lang.Byte java.lang.Byte.valueOf(java.lang.String,int) throws
java.lang.NumberFormatException
   public static int java.lang.Byte.compare(byte,byte)
   public static java.lang.Byte java.lang.Byte.decode(java.lang.String) throws
java.lang.NumberFormatException
   public static byte java.lang.Byte.parseByte(java.lang.String) throws
java.lang.NumberFormatException
   public static byte java.lang.Byte.parseByte(java.lang.String,int) throws
java.lang.NumberFormatException

   ALL PUBLIC METHODS

   public boolean java.lang.Byte.equals(java.lang.Object)
   public java.lang.String java.lang.Byte.toString()
   public static java.lang.String java.lang.Byte.toString(byte)
   public int java.lang.Byte.hashCode()
   public int java.lang.Byte.compareTo(java.lang.Byte)
   public int java.lang.Byte.compareTo(java.lang.Object)
   public byte java.lang.Byte.byteValue()
   public short java.lang.Byte.shortValue()
   public int java.lang.Byte.intValue()
   public long java.lang.Byte.longValue()
   public float java.lang.Byte.floatValue()
   public double java.lang.Byte.doubleValue()
   public static java.lang.Byte java.lang.Byte.valueOf(byte)
   public static java.lang.Byte java.lang.Byte.valueOf(java.lang.String) throws
java.lang.NumberFormatException
   public static java.lang.Byte java.lang.Byte.valueOf(java.lang.String,int) throws
java.lang.NumberFormatException
   public static int java.lang.Byte.compare(byte,byte)
   public static java.lang.Byte java.lang.Byte.decode(java.lang.String) throws
java.lang.NumberFormatException
   public static byte java.lang.Byte.parseByte(java.lang.String) throws
java.lang.NumberFormatException
   public static byte java.lang.Byte.parseByte(java.lang.String,int) throws
java.lang.NumberFormatException
   public final void java.lang.Object.wait() throws java.lang.InterruptedException
   public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
   public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
   public final native java.lang.Class java.lang.Object.getClass()
   public final native void java.lang.Object.notify()
   public final native void java.lang.Object.notifyAll()

   private static class java.lang.Byte$ByteCache
   {
      static final java.lang.Byte[] java.lang.Byte$ByteCache.cache

      private java.lang.Byte$ByteCache()

      ALL PUBLIC METHODS

      public final void java.lang.Object.wait() throws java.lang.InterruptedException
      public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
      public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
      public boolean java.lang.Object.equals(java.lang.Object)
      public java.lang.String java.lang.Object.toString()
      public native int java.lang.Object.hashCode()
      public final native java.lang.Class java.lang.Object.getClass()
      public final native void java.lang.Object.notify()
      public final native void java.lang.Object.notifyAll()
   }
}
```

One of Table C–1's methods not demonstrated in Listing C–1 is `newInstance()`, which is useful for instantiating a dynamically loaded class, provided that the class has a noargument constructor.

Suppose you plan to create a file disassembler application that lets the user disassemble different kinds of files, based on the file extension. For example, the file disassembler can disassemble a `.class` file to reveal its internal structure. When the file extension isn't recognized, the file disassembler generates a hexadecimal listing.

The file disassembler includes only a default disassembler for generating a listing in hexadecimal format. Additional disassemblers are made available via plugins. The reason is that you don't want to integrate each disassembler's code into the file disassembler, because you would then have to recompile the file disassembler application each time you introduced a new disassembler.

Instead, you create a `plugins` directory to serve as a package for storing disassembler classfiles. Each disassembler is implemented by a class that extends Listing C–2's `Disassembler` class.

*Listing C–2. Abstracting a disassembler*

```
package plugins;

import java.io.IOException;

import java.util.List;

public abstract class Disassembler
{
   public abstract List<String> disassemble(String filename) throws IOException;
}
```

Listing C–2's `Disassembler` class presents a single `List<String> disassemble(String filename)` method that accepts a filename as its argument. It attempts to open this file and read its contents. These contents are then converted into a list of strings that is returned. A `java.io.IOException` instance is thrown when there's a problem with opening or reading content from the file.

Listing C–3 presents the source code to a disassembler for Java classfiles. This disassembler is unfinished; consider it an exercise to finish it.

*Listing C–3. A disassembler for disassembling Java classfiles*

```
package plugins;

import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class Class extends Disassembler
{
   @Override
   public List<String> disassemble(String filename) throws IOException
   {
      DataInputStream dis = null;
      try
      {
         FileInputStream fis = new FileInputStream(filename);
         dis = new DataInputStream(fis);
         List<String> lines = new ArrayList<String>();
         StringBuilder sb = new StringBuilder();
         if (dis.read() != 0xca || dis.read() != 0xfe ||
             dis.read() != 0xba || dis.read() != 0xbe)
         {
            lines.add(filename + " is not a classfile");
            return lines;
         }
         lines.add("00000000  Signature            CAFEBABE");
         lines.add("00000004  Minor Version        " +
                   (short) dis.readUnsignedShort());
         lines.add("00000006  Major Version        " +
                   dis.readUnsignedShort());
         lines.add("00000008  Constant Pool Count  " +
                   dis.readUnsignedShort());
         return lines;
      }
      finally
      {
         if (dis != null)
            try
            {
               dis.close();
            }
            catch (IOException ioe)
            {
               assert false; // shouldn't happen in this context
            }
      }
   }
}
```

Listing C–3 is pretty straightforward. The `DataInputStream` class is used to conveniently read multibyte data from the file.

Finally, Listing C–4 presents an `FD` application class that uses the Reflection API to dynamically load a disassembler plugin.

*Listing C–4. Dynamically loading, instantiating, and using a disassembler plugin*

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

import java.util.List;

import plugins.Disassembler;

public class FD // File Disassembler
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage  : java FD filespec");
         System.err.println("example: java FD FD.class");
         return;
      }
      if (args[0].equals("."))
         return;
      try
      {
         int index = args[0].indexOf('.');
         if (index == -1)
         {
            dumpInHexFormat(args[0]);
            return;
         }
         String ext = Character.toUpperCase(args[0].charAt(index + 1)) +
                      args[0].substring(index + 2);
         Class<?> clazz = Class.forName("plugins" + "." + ext);
         Disassembler d = (Disassembler) clazz.newInstance();
         List<String> lines = d.disassemble(args[0]);
         for (String line: lines)
            System.out.println(line);
      }
      catch (ClassNotFoundException cnfe)
      {
         dumpInHexFormat(args[0]);
      }
      catch (IllegalAccessException iae)
      {
         System.err.println("Illegal access: " + iae.getMessage());
      }
      catch (InstantiationException ie)
      {
         System.err.println("Cannot instantiate loaded class");
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
   }

   static void dumpInHexFormat(String filename)
   {
      FileInputStream fis = null;
      try
      {
         fis = new FileInputStream(filename);
         StringBuilder sb = new StringBuilder();
         int offset = 0;
         int ch;
         while ((ch = fis.read()) != -1)
         {
            if ((offset % 16) == 0)
               System.out.printf("%08X ", offset);
            System.out.printf("%02X ", ch);
            if (ch < 32 || ch > 126)
               sb.append('.');
            else
               sb.append((char) ch);
            if ((++offset % 16) == 0)
            {
               System.out.println(sb);
               sb.setLength(0);
            }
         }
         if (sb.length() != 0)
         {
            for (int i = 0; i < 16 - sb.length(); i++)
               System.out.print("   ");
            System.out.println(sb);
         }
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
      finally
      {
         if (fis != null)
            try
            {
               fis.close();
            }
            catch (IOException ioe)
            {
               assert false; // shouldn't happen in this context
            }
      }
   }
}
```

Listing C–4's `main()` method first validates that a single command-line argument, identifying the name of the file (including the file extension, such as `.class`) to disassemble, is specified. To prevent a string index out of bounds exception when a single period character is specified, `main()` checks for this possibility and terminates the application in this case.

Continuing, `main()` searches the filename for a period. When a period isn't found, `main()` realizes that no file extension is specified, and calls the `dumpInHexFormat()` method with the filename as this method's solitary argument to dump the file contents as a hexadecimal listing, and then terminates the application.

The file extension (minus the leading period) is now extracted from the string and the first character of the extension is uppercased, because plugin class names conventionally match file extension names except for the first character. For example, the plugin class name for a `.class` file extension is `Class`.

It's now a simple matter to attempt to load the plugin classfile from the `plugins` directory/package. However, when the plugin classfile cannot be loaded (probably because you specified an extension for which no plugin currently exists; for example, `.png` was specified but no `Png` classfile exists), `ClassNotFoundException` is thrown. Its catch block dumps the file's contents in hexadecimal.

At this point, `main()` tries to instantiate the plugin class by calling `newInstance()`. This method throws `IllegalAccessException` when it cannot locate a noargument constructor (e.g., the default noargument constructor) in the classfile. It throws `InstantiationException` when there's something wrong with the class described by the classfile (e.g., perhaps the class is abstract).

Finally, the plugin's `disassemble()` method is called with the filename as this method's argument. If `IOException` isn't thrown, `main()` outputs the returned list of strings to standard output.

The `dumpInHexFormat()` method first attempts to open a file input stream to the file identified by its `filename` parameter. It then enters a loop where it reads the file's contents one byte at a time. The contents are output on a line-by-line basis where each line begins with eight hexadecimal digits that describe a zero-based offset into the file.

Following the offset is a sequence of up to 16 hexadecimal digit pairs, where each pair presents the hexadecimal equivalent of the byte value. Following this sequence is a sequence of literal characters whose Unicode values are greater than or equal to the space character and less than or equal to 126 (the tilda character [~]).

Accomplish the following tasks to create the `FD` application:

1. Create a `plugins` subdirectory of the current directory.
2. Copy Listing C–2 to a `Disassembler.java` file and store this file in `plugins`.
3. Copy Listing C–3 to a `Class.java` file and store this file in `plugins`.

4. Copy Listing C–4 to an `FD.java` file and store this file in the current directory.

5. Compile `FD.java` via `javac FD.java`.

6. Compile `Disassembler.java` and `Class.java` via `javac plugins/*.java`.

If all goes well, you should observe `FD.class` in the current directory. You should also observe `Disassembler.class` and `Class.class` in the `plugins` directory.

At the command line, execute `java FD FD.class`. You should observe the following output:

```
00000000  Signature           CAFEBABE
00000004  Minor Version       0
00000006  Major Version       51
00000008  Constant Pool Count  195
```

Next, execute `java FD FD.java`. You should now see a hexadecimal listing, with the first part shown below:

```
00000000 69 6D 70 6F 72 74 20 6A 61 76 61 2E 69 6F 2E 46 import java.io.F
00000010 69 6C 65 3B 0D 0A 69 6D 70 6F 72 74 20 6A 61 76 ile;..import jav
00000020 61 2E 69 6F 2E 46 69 6C 65 49 6E 70 75 74 53 74 a.io.FileInputSt
00000030 72 65 61 6D 3B 0D 0A 69 6D 70 6F 72 74 20 6A 61 ream;..import ja
00000040 76 61 2E 69 6F 2E 49 4F 45 78 63 65 70 74 69 6F va.io.IOExceptio
00000050 6E 3B 0D 0A 0D 0A 69 6D 70 6F 72 74 20 6A 61 76 n;....import jav
00000060 61 2E 75 74 69 6C 2E 4C 69 73 74 3B 0D 0A 0D 0A a.util.List;....
00000070 69 6D 70 6F 72 74 20 70 6C 75 67 69 6E 73 2E 44 import plugins.D
```

Table C–1's descriptions of the `getAnnotations()` and `getDeclaredAnnotations()` methods reveal that each method returns an array of `Annotation`, an interface that's located in the `java.lang.annotation` package. `Annotation` is the superinterface of `Override`, `SuppressWarnings`, and all other annotation types.

Table C–1's method descriptions also refer to `Constructor`, `Field`, and `Method`. Instances of these classes represent a class's constructors and a class's or an interface's fields and methods.

`Constructor` represents a constructor and is generically declared as `Constructor<T>`, where `T` identifies the class in which the constructor represented by `Constructor` is declared. `Constructor` declares various methods, including the following methods:

- `Annotation[] getDeclaredAnnotations()` returns an array of all annotations declared on the constructor. The returned array has zero length when there are no annotations.

- `Class<T> getDeclaringClass()` returns a `Class` object that represents the class in which the constructor is declared.

- `Class[]<?> getExceptionTypes()` returns an array of `Class` objects representing the types of exceptions listed in the constructor's throws clause. The returned array has zero length when there is no throws clause.
- `String getName()` returns the constructor's name.
- `Class[]<?> getParameterTypes()` returns an array of `Class` objects representing the constructor's parameters. The returned array has zero length when the constructor declares no parameters.

> ■ **TIP:** If you want to instantiate a class via a constructor that takes arguments, you cannot use `Class`'s `newInstance()` method. Instead, you must use `Constructor`'s `T newInstance(Object... initargs)` method to perform this task. Unlike `Class`'s `newInstance()` method, which bypasses the compile-time exception checking that would otherwise be performed by the compiler, `Constructor`'s `newInstance()` method avoids this problem by wrapping any exception thrown by the constructor in an instance of the `java.lang.reflect.InvocationTargetException` class.

`Field` represents a field and declares various methods, including the following getter methods:

- `Object get(Object object)` returns the value of the field for the specified `object`.
- `boolean getBoolean(Object object)` returns the value of the Boolean field for the specified `object`.
- `byte getByte(Object object)` returns the value of the byte integer field for the specified `object`.
- `char getChar(Object object)` returns the value of the character field for the specified `object`.
- `double getDouble(Object object)` returns the value of the double precision floating-point field for the specified `object`.
- `float getFloat(Object object)` returns the value of the floating-point field for the specified `object`.
- `int getInt(Object object)` returns the value of the integer field for the specified `object`.
- `long getLong(Object object)` returns the value of the long integer field for the specified `object`.

- `short getShort(Object object)` returns the value of the short integer field for the specified `object`.

`get()` returns the value of any type of field. In contrast, the other listed methods return the values of specific types of fields. These methods throw a `NullPointerException` instance when `object` is `null` and the field is an instance field, a `java.lang.IllegalArgumentException` instance when `object` is not an instance of the class or interface declaring the underlying field (or not an instance of a subclass or interface implementor), and an `IllegalAccessException` instance when the underlying field cannot be accessed (e.g., it's private).

Listing C–5 demonstrates `Field`'s `getInt(Object)` and `getDouble(Object)` methods along with their `void setInt(Object obj, int i)` and `void setDouble(Object obj, double d)` counterparts.

*Listing C–5. Reflectively getting and setting the values of instance and class fields*

```
import java.lang.reflect.Field;

class X
{
   public int i = 10;
   public static final double PI = 3.14;
}

public class FieldAccessDemo
{
   public static void main(String[] args)
   {
      try
      {
         Class<?> clazz = Class.forName("X");
         X x = (X) clazz.newInstance();
         Field f = clazz.getField("i");
         System.out.println(f.getInt(x)); // Output: 10
         f.setInt(x, 20);
         System.out.println(f.getInt(x)); // Output: 20
         f = clazz.getField("PI");
         System.out.println(f.getDouble(null)); // Output: 3.14
         f.setDouble(x, 20);
         System.out.println(f.getDouble(null)); // Never executed
      }
      catch (Exception e)
      {
         System.err.println(e);
      }
   }
}
```

Listing C–5 declares classes `X` and `FieldAccessDemo`. I've included `X`'s source code with `FieldAccessDemo`'s source code for convenience. However, you can imagine this source code being stored in a separate source file.

FieldAccessDemo's main() method first attempts to load X, and then tries to instantiate this class via newInstance(). If successful, the instance is assigned to reference variable x.

main() next invokes Class's Field getField(String name) method to return a Field instance that represents the public field identified by name, which happens to be i (in the first case) and PI (in the second case). This method throws java.lang.NoSuchFieldException when the named field doesn't exist.

Continuing, main() invokes Field's getInt() and setInt() methods (with an object reference) to get the instance field's initial value, change this value to another value, and get the new value. The initial and new values are output.

At this point, main() demonstrates class field access in a similar manner. However, it passes null to getInt() and setInt() because an object reference isn't required to access a class field. Because PI is declared final, the call to setInt() results in a thrown instance of the IllegalAccessException class.

**NOTE:** I've specified catch(Exception e) to avoid having to specify multiple catch blocks.

Method represents a method and declares various methods, including the following methods:

- int getModifiers() returns a 32-bit integer whose bit fields identify the method's reserved word modifiers (e.g., public, abstract, or static). These bit fields must be interpreted via the Modifier class. For example, you might specify (method.getModifiers() & Modifier.ABSTRACT) == Modifier.ABSTRACT to find out if the method (represented by the Method object whose reference is stored in method) is abstract—this expression evaluates to true when the method is abstract.

- Class<?> getReturnType() returns a Class object that represents the method's return type.

- Object invoke(Object receiver, Object... args) calls the method on the object identified by receiver (which is ignored when the method is a class method), passing the variable number of arguments identified by args to the called method. The invoke() method throws NullPointerException when receiver is null and the method being invoked is an instance method, IllegalAccessException when the method isn't accessible (e.g., it's private), IllegalArgumentException when an incorrect number of arguments are passed to the method (and other reasons), and InvocationTargetException when an exception is thrown from the called method.

- boolean isVarArgs() returns true when the method is declared to receive a variable number of arguments.

Listing C–6 demonstrates `Method`'s `invoke(Object, Object...)` method.

*Listing C–6. Reflectively invoking instance and class methods*

```
import java.lang.reflect.Method;

class X
{
   public void objectMethod(String arg)
   {
      System.out.println("Instance method: " + arg);
   }

   public static void classMethod()
   {
      System.out.println("Class method");
   }
}

public class MethodInvocationDemo
{
   public static void main(String[] args)
   {
      try
      {
         Class<?> clazz = Class.forName("X");
         X x = (X) clazz.newInstance();
         Class[] argTypes = { String.class };
         Method method = clazz.getMethod("objectMethod", argTypes);
         Object[] data = { "Hello" };
         method.invoke(x, data); // Output: Instance method: Hello
         method = clazz.getMethod("classMethod", (Class<?>[]) null);
         method.invoke(null, (Object[]) null); // Output: Class method
      }
      catch (Exception e)
      {
         System.err.println(e);
      }
   }
}
```

Listing C–6 declares classes `X` and `MethodInvocationDemo`. `MethodInvocationDemo`'s `main()` method first attempts to load `X`, and then tries to instantiate this class via `newInstance()`. When successful, the instance is assigned to reference variable `x`.

`main()` next creates a one-element `Class` array that describes the types of `objectMethod()`'s parameter list. This array is used in the subsequent call to `Class`'s `Method getMethod(String name, Class<?>... parameterTypes)` method to return a `Method` object for invoking a `public` method named `objectMethod` with this parameter list. This method throws `java.lang.NoSuchMethodException` when the named method doesn't exist.

Continuing, `main()` creates an `Object` array that specifies the data to be passed to the method's parameters; in this case, the array consists of a single `String` argument. It then reflectively invokes `objectMethod()` by passing this array along with the object reference stored in `x` to the `invoke()` method.

At this point, `main()` shows you how to reflectively invoke a class method. The `(Class<?>[])` and `(Object[])` casts are used to suppress warning messages that have to do with variable numbers of arguments and null references. Notice that the first argument passed to `invoke()` is `null` when invoking a class method.

The `java.lang.reflect.AccessibleObject` class is the superclass of `Constructor`, `Field`, and `Method`. This superclass provides methods for reporting a constructor's, field's, or method's accessibility (is it private?) and making an inaccessible constructor, field, or method accessible. `AccessibleObject`'s methods include the following:

- `<T extends Annotation> T getAnnotation(Class<T> annotationType)` returns the constructor's, field's, or method's annotation of the specified type when such an annotation is present; otherwise, null returns.
- `boolean isAccessible()` returns true when the constructor, field, or method is accessible.
- `boolean isAnnotationPresent(Class<? extends Annotation> annotationType)` returns true when an annotation of the type specified by `annotationType` has been declared on the constructor, field, or method. This method takes inherited annotations into account.
- `void setAccessible(boolean flag)` attempts to make an inaccessible constructor, field, or method accessible when `flag` is `true`.

■ **NOTE:** The `java.lang.reflect` package also includes an `Array` class whose class methods make it possible to reflectively create and access Java arrays.

I previously showed you how to obtain a `Class` object via `Class`'s `forName()` method. Another way to obtain a `Class` object is to call `Object`'s `getClass()` method on an object reference; for example, `Employee e = new Employee(); Class<? extends Employee> clazz = e.getClass();`. The `getClass()` method doesn't throw an exception because the class from which the object was created is already present in memory.

There is one more way to obtain a `Class` object, and that is to employ a *class literal*, which is an expression consisting of a class name, followed by a period separator, that's followed by reserved word `class`. Examples of class literals include `Class<Employee> clazz = Employee.class;` and `Class<String> clazz = String.class`.

Perhaps you're wondering about how to choose between `forName()`, `getClass()`, and a class literal. To help you make your choice, the following list compares each competitor:

- `forName()` is very flexible in that you can dynamically specify any reference type by its package-qualified name. If the type isn't in memory, it's loaded, linked, and initialized. However, lack of compile-time type safety can lead to runtime failures.

- `getClass()` returns a `Class` object describing the type of its referenced object. When called on a superclass variable containing a subclass instance, a `Class` object representing the subclass type is returned. Because the class is in memory, type safety is assured.

- A class literal returns a `Class` object representing its specified class. Class literals are compact and the compiler enforces type safety by refusing to compile the source code when it cannot locate the literal's specified class.

■ **NOTE:** You can use class literals with primitive types, including void. Examples include `int.class`, `double.class`, and `void.class`. The returned `Class` object represents the class identified by a primitive type wrapper class's `TYPE` field or `java.lang.Void.TYPE`. For example, each of `int.class == Integer.TYPE` and `void.class == Void.TYPE` evaluates to true.

You can also use class literals with primitive type-based arrays. Examples include `int[].class` and `double[].class`. For these examples, the returned `Class` objects represent `Class<int[]>` and `Class<double[]>`.

## Classloaders

The virtual machine relies on *classloaders* to dynamically load compiled classes and other reference types (classes, for short) from classfiles, Java Archive (JAR) files, URLs, and other sources into memory. Classloaders insulate the virtual machine from filesystems, networks, and so on.

This section introduces you to classloaders by first presenting the various kinds of classloaders supported by Java. It then presents the mechanics of loading a class. After demonstrating classloaders and revealing various difficulties with them, it closes by discussing resources.

## Kinds of Classloaders

When the virtual machine starts running, three classloaders are available:

- *Bootstrap* (also known as *primordial* or *default*) uses the operating system file I/O mechanism to load classes from the core Java libraries (e.g., the `java.lang` and `java.io` packages found in `rt.jar`) and is written in native code.

- *Extension* loads classes from the JAR files located in the extensions directories (e.g., `<JAVA_HOME>/lib/ext`). It's implemented by the `sun.misc.Launcher$ExtClassLoader` class (in `rt.jar`).

- *System* (also known as *application*) loads an application's classes and resources found on the classpath and is implemented by the `sun.misc.Launcher$AppClassLoader` class (in `rt.jar`).

Additionally, the standard class library provides the abstract `java.lang.ClassLoader` class as the ultimate root class for all classloaders (including extension and system) except for bootstrap.

`ClassLoader` is subclassed by the concrete `java.security.SecureClassLoader` class, which takes security information into account; and `SecureClassLoader` is subclassed by the concrete `java.net.URLClassLoader` class, which lets you load classes and resources from a search path of URLs referring to JAR files and directories. (`ExtClassLoader` and `AppClassLoader` extend `URLClassLoader`.)

> **NOTE:** Android supplies additional `DexClassLoader` and `PathClassLoader` classes that extend the common `BaseDexClassLoader` class, which itself extends `ClassLoader`. These three classes exist in the `dalvik.system` package.

Starting with Java 1.2, classloaders have a hierarchical relationship in which each classloader except for bootstrap has a parent classloader. The bootstrap classloader is the *root classloader*, in much the same way as `Object` is the root reference type.

Along with these kinds of classloaders, Java recognizes current and context classloaders.

The *current classloader* is the classloader that loads the class to which the currently executing method belongs, and is implied by methods such as `Class`'s `Class<?> forName(String className)` method. Behind the scenes, `forName()` relies on the current class loader to load the specified class.

When the current classloader (the bootstrap classloader is never current after loading the core Java libraries) is asked to load a class, it asks its parent to perform this task. When the parent cannot load the class, the parent asks its parent classloader to do so, and this process continues on up the hierarchy. If none of these classloaders can load the class, the current classloader is given a chance. This behavior is known as *delegation*.

The *context classloader* (introduced by Java 1.2) is the classloader associated with the current thread. The current thread's context classloader is inherited from the thread's parent thread, and defaults to the system classloader (which happens to be the classloader associated with the application's main—ultimate parent—thread). Also, the context class loader has a parent classloader and supports the same delegation model for class loading as previously described.

The `java.lang.Thread` class declares a `void setContextClassLoader(ClassLoader cl)` method that a parent thread invokes to specify a child thread's classloader before starting the child thread. A companion `ClassLoader getContextClassLoader()` method is also declared.

## Class-Loading Mechanics

Central to `ClassLoader` are its `Class<?> loadClass(String name)` and `protected Class<?> loadClass(String name, boolean resolve)` methods, which try to load the class with the specified `name`. They throw `ClassNotFoundException` when the class cannot be found, or return a `Class` object representing the loaded class. The former method invokes the latter method, passing `false` to `resolve`. (In the Android reference implementation, `loadClass(String, boolean)`'s second parameter is ignored.)

■ **NOTE:** The `String` object passed to `name` must specify the class's binary name, which adheres to the convention that's specified in the *Java Language Specification* (`http://docs.oracle.com/javase/specs/`). Examples include `"java.lang.String"` and `"java.net.URLClassLoader$3$1"`.

The `loadClass(String name, boolean resolve)` method performs the following tasks:

1. It invokes `ClassLoader`'s `protected final Class<?> findLoadedClass(String name)` method to return the class with the given binary `name` when the calling classloader has been recorded by the virtual machine as an initiating loader of a class with that binary `name`. Otherwise, null is returned. This method returns the class from the calling classloader's class cache when the class was previously loaded (and is in this cache, for performance reasons).

2. When `findLoadedClass()` returns null, `loadClass(String, boolean)` invokes `loadClass(String, boolean)` on the nonnull parent classloader, or invokes an internal method requesting that the bootstrap classloader handle this task.

3. When neither the parent nor any of its parents (including bootstrap) locates the class, `ClassNotFoundException` is thrown and the initial `loadClass(String, boolean)` method call invokes `ClassLoader`'s `protected Class<?> findClass(String name)` method to find and load the class.

4. The `findClass()` method locates the class or throws `ClassNotFoundException` when the class cannot be found; this exception is thrown out of `loadClass(String, boolean)` and `loadClass(String)`. (`ClassLoader`'s `findClass()` method always throws `ClassNotFoundException`.)

5. Assuming that the class is located, `findClass()` loads the class's compiled representation into an array of bytes. It then (ultimately) invokes one of `ClassLoader`'s `defineClass()` methods to convert these bytes into a `Class` object, but only after `defineClass()` runs these bytes through the bytecode verifier (see Chapter 1) to ensure that they don't compromise virtual machine security.

6. Assuming that all is well, `findClass()` returns a `Class` object. When `true` is passed to `loadClass(String, boolean)`'s `resolve` parameter, `ClassLoader`'s `protected final void resolveClass(Class<?> c)` method is called to resolve the class. *Resolution* causes any other classes that are immediately referenced from the loaded class (e.g., a `static` variable of another class type) to be loaded by this classloader, classes immediately referenced from those classes to be loaded, and so on. (Classes used as instance variables, parameters, or local variables are not normally loaded at this time. They're loaded when actually referenced by the class.)

`findLoadedClass()` is declared `protected` so that it can be accessed by subclasses in different packages. This method is declared `final` so that it cannot be overridden. The same is true for `resolveClass()`.

■ **NOTE:** When you need your own classloader, you should first consider using `URLClassLoader`, which will save you a lot of work, and it also leverages the security features provided by its `SecureClassLoader` parent. If you prefer to subclass `ClassLoader` directly, you'll minimally need to override `findClass()`.

## Playing with Classloaders

I've created a pair of applications to help you start to explore classloaders. Listing C–7 presents the first application, which reveals that the main thread's context classloader is the system classloader.

*Listing C–7. Proving that the main thread's context classloader is the system classloader*

```
public class ClassLoaderDemo
{
   public static void main(String[] args)
   {
      System.out.println(Thread.currentThread().getContextClassLoader());
      System.out.println(ClassLoader.getSystemClassLoader());
   }
}
```

Listing C–7's `main()` method first obtains the main thread's context classloader and outputs this classloader's name. This method then invokes `ClassLoader`'s `ClassLoader getSystemClassLoader()` class method, which returns a reference to the system classloader's `ClassLoader` object. The name of this classloader is then output.

Compile Listing C–7 (`javac ClassLoaderDemo.java`) and run the application (`java ClassLoaderDemo`). You should observe output similar to that shown below:

```
sun.misc.Launcher$AppClassLoader@26e2e276
sun.misc.Launcher$AppClassLoader@26e2e276
```

My second application demonstrates `URLClassLoader`. It uses this classloader to load a class named `Hello` via a URL. This class's `Hello.class` classfile is located in directory `x`, which is a subdirectory of the current directory (the directory in which the virtual machine is launched via the `java` tool). Listing C–8 presents `Hello.java`.

*Listing C–8. A simple demonstration class*

```
public class Hello
{
   static
   {
      System.out.println("Welcome to Hello");
   }

   static int x = 1;

   public static void main(String[] args)
   {
      System.out.println("Hello");
      System.out.println("Number of arguments = " + args.length);
      System.out.println("x = " + x);
   }
}
```

`Hello` must be declared `public`; otherwise, `URLClassLoader` outputs an error message about its not being able to access this class.

Assuming that `Hello.java` is located in an `x` subdirectory of the current directory, compile Listing C–8 via `javac x/Hello.java`.

Listing C–9 presents an application that loads and runs this class.

*Listing C–9. Attempting to load `Hello.class` via a file-based URL*

```
import java.lang.reflect.Method;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class ClassLoaderDemo
{
   final static String _URL_ = "file:///" + System.getProperty("user.dir") + "/x/";

   public static void main(String[] args)
   {
      boolean init = true;
      if (args.length == 1 && args[0].equalsIgnoreCase("noinit"))
         init = false;
      try
      {
         URL[] urls = new URL[] { new URL(_URL_) };
         URLClassLoader urlc = new URLClassLoader(urls);
         Class<?> clazz = Class.forName("Hello", init, urlc);
         System.out.println(clazz.getClassLoader());
         run(clazz);
      }
      catch (ClassNotFoundException cnfe)
      {
         System.err.println("Class not found");
      }
      catch (MalformedURLException murle)
      {
         System.err.println("URL is malformed");
      }
   }

   static void run(Class<?> clazz)
   {
      try
      {
         Method main = clazz.getMethod("main", new Class[] { String[].class });
         Object[] args = new Object[] { new String[0] };
         main.invoke(null, args);
      }
      catch (Exception e)
      {
         System.err.println(e.getMessage());
      }
   }
}
```

Listing C–9's `main()` method first examines it array of command-line arguments for the presence of a `noinit` argument, and resets the `init` variable (which is initially true) to false when this argument is specified. (I'll explain this variable's purpose shortly.)

`main()` next attempts to instantiate `URLClassLoader` by passing an array of one `java.net.URL` instance to this class's constructor. The string-based URL argument passed to `URL`'s constructor consists of the "`file:///`" protocol prefix (to access the local filesystem), followed by `System.getProperty("user.dir")`'s result (obtain the current working directory), followed by "`/x/`". The final "`/`" is required; otherwise, `URLClassLoader` assumes that `x` is the name of a JAR file. (Because this JAR file doesn't exist, and obviously doesn't contain `Hello.class`, `ClassNotFoundException` would be thrown.)

Assuming that `URLClassLoader` is successfully created, `main()` invokes `Class`'s `Class<?> forName(String name, boolean initialize, ClassLoader loader)` method to try to load the class identified by `name` and using `loader`. Passing `true` to `initialize` causes the loaded class to be statically initialized. Otherwise, the loaded class isn't statically initialized. The `init` variable lets you explore both initialization scenarios.

The returned `Class` object's `ClassLoader getClassLoader()` method is invoked to return the `ClassLoader` instance used to load the class, and this name is subsequently output.

At this point, `ClassLoaderDemo`'s `void run(Class<?> clazz)` class method is called to instantiate the loaded class and invoke its `main()` method with help from the Reflection API (discussed previously in this appendix).

Assuming that `ClassLoaderDemo.java` is located with `x` in the current directory, compile this source file via `javac ClassLoaderDemo.java`. Then run the application via `java ClassLoaderDemo`. The application responds by presenting the following output (except possibly for `56092666`):

```
Welcome to Hello
java.net.URLClassLoader@56092666
Hello
Number of arguments = 0
x = 1
```

This output order proves that `Class.forName()` statically initialized `Hello` (`true` was passed to `initialize`), by invoking its `<clinit>()` virtual machine method (see Chapter 3).

Run the application via `java ClassLoaderDemo noinit`. The application responds by presenting the following output (except possibly for `56092666`):

```
java.net.URLClassLoader@56092666
Welcome to Hello
Hello
Number of arguments = 0
x = 1
```

This output order proves that `Class.forName()` didn't statically initialize `Hello`. However, this class is statically initialized just before `run()` launches its `main()` method.

Copy `Hello.class` into the same directory as `ClassLoaderDemo.class` and then execute `java ClassLoader`. This time, you'll observe the following output (except possibly for `26e2e276`):

```
Welcome to Hello
sun.misc.Launcher$AppClassLoader@26e2e276
Hello
Number of arguments = 0
x = 1
```

The reason for this different output is that `URLClassLoader` delegates to its parent classloader, which happens to be the system classloader. The system classloader locates `Hello.class` in the current directory and causes `Class.forName()` to return this class's `Class` object.

## Classloader Difficulties

Context classloaders can be a source of difficulty as discussed in "Find a way out of the ClassLoader maze" (`www.javaworld.com/javaworld/javaqa/2003-06/01-qa-0606-load.html`), a JavaWorld article by Vladimir Roubtsov. I've created a simple example that demonstrates this difficulty.

My example is based on two versions of a class named `Version`. Listing C–10 presents the first version's source code.

*Listing C–10. A simple demonstration class*

```
public class Version
{
   public static void main(String[] args)
   {
      System.out.println("Version 1");
   }
}
```

Assuming that `Version.java` is located in an `x` subdirectory of the current directory, compile Listing C–10 via `javac x/Version.java`.

Create a `y` subdirectory of the current directory and copy a modified version of Listing C–10 (replace `Version 1` with `Version 2`) to `y`. Then compile this source code via `javac y/Version.java`.

Listing C–11 presents a `ClassLoaderDemo` application that demonstrates a conflict between these versions.

*Listing C–11. Entering classloader hell*

```
import java.lang.reflect.Method;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class ClassLoaderDemo
{
   final static String CD = System.getProperty("user.dir");
   final static String _URL1_ = "file:///" + CD + "/x/";
   final static String _URL2_ = "file:///" + CD + "/y/";

   public static void main(String[] args)
   {
      try
      {
         URL[] urls = new URL[] { new URL(_URL1_) };
         URLClassLoader urlc1 = new URLClassLoader(urls);
         Class<?> clazz1 = Class.forName("Version", true, urlc1);
         run(clazz1);
         urls = new URL[] { new URL(_URL2_) };
         URLClassLoader urlc2 = new URLClassLoader(urls);
         Class<?> clazz2 = Class.forName("Version", true, urlc2);
         run(clazz2);
         Thread.currentThread().setContextClassLoader(urlc1);
         run(Thread.currentThread().getContextClassLoader().loadClass("Version"));
         Thread.currentThread().
            setContextClassLoader(ClassLoader.getSystemClassLoader());
         run(Thread.currentThread().getContextClassLoader().loadClass("Version"));
      }
      catch (ClassNotFoundException cnfe)
      {
         System.err.println("Class not found");
      }
      catch (MalformedURLException murle)
      {
         System.err.println("URL is malformed");
      }
   }

   static void run(Class<?> clazz)
   {
      try
      {
         Method main = clazz.getMethod("main", new Class[] { String[].class });
         Object[] args = new Object[] { new String[0] };
         main.invoke(null, args);
      }
      catch (Exception e)
      {
         System.err.println(e.getMessage());
      }
   }
}
```

Listing C–11 loads both `Version` classes; each classloader caches its own version. It then sets the context classloader, first to the `URLClassLoader` instance and then to the system classloader. Each time, it subsequently calls `loadClass(String)`.

Compile Listing C–11 (`javac ClassLoaderDemo.java`) and run the application (`java ClassLoaderDemo`). The following output is generated:

```
Version 1
Version 2
Version 1
Class not found
```

The system classloader doesn't include `Version` in its cache because it didn't load `Version`. As a result, `loadClass(String)` throws `ClassNotFoundException`.

Although this example is contrived, it illustrates problems that can occur when you're not aware of what classloader is the context classloader. You'll either observe a thrown `ClassNotFoundException` instance or you may end up with the wrong version of a class.

## Classloaders and Resources

Classloaders are typically used to load classes, but can also load arbitrary resources (e.g., images) via `ClassLoader` methods such as `InputStream getResourceAsStream(String name)`. Although you could call these methods directly, it's common practice to work with `Class`'s `URL getResource(String name)` and `InputStream getResourceAsStream(String name)` methods instead. These methods differ in that `getResourceAsStream()` ultimately invokes `getResource()`, and then invokes `URL`'s `InputStream openStream()` method on the resulting `URL` instance to return an input stream.

I've created an application that demonstrates `Class`'s `getResourceAsStream()` method. Listing C–12 presents its source code.

*Listing C–12. Loading an image resource and viewing a prefix of its content*

```
import java.io.File;
import java.io.InputStream;
import java.io.IOException;

public class ClassLoaderDemo
{
   final static String IMAGE = "mars.jpg";

   public static void main(String[] args)
   {
      System.out.println(ClassLoaderDemo.class.getClassLoader());
      InputStream is = ClassLoaderDemo.class.getResourceAsStream(IMAGE);
      if (is == null)
      {
         System.err.printf("%s not found%n", IMAGE);
         return;
      }
      try
      {
         byte[] image = new byte[(int) new File(IMAGE).length()];
         int _byte, i = 0;
         while ((_byte = is.read()) != -1)
            image[i++] = (byte) _byte;
         for (i = 0; i < 16; i++)
            System.out.printf("%02X ", image[i]);
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
   }
}
```

Listing C–12 specifies expression `ClassLoaderDemo.class` to obtain the `Class` object returned from the classloader that loaded `ClassLoaderDemo.class`, and then invokes `getResourceAsStream(IMAGE)` on the returned `Class` object—this image needs to be located in the current directory.

Assuming that this image resource can be found (`getResourceAsStream()` doesn't return null), a byte array is allocated to hold the array, and the image's bytes are read over the input stream and stored in the byte array. The first 16 bytes from this array are then sent to standard output.

Compile Listing C–12 (`javac ClassLoaderDemo.java`) and run the application (`java ClassLoaderDemo`). You should observe the following output:

```
sun.misc.Launcher$AppClassLoader@26e2e276
FF D8 FF E0 00 10 4A 46 49 46 00 01 02 01 00 48
```

It's common to store resources in JAR files and then access them via `getResourceAsStream()`. For example, execute the following command (which assumes that the current directory contains `mars.jpg`) to create an `image.jar` file containing `mars.jpg`:

```
jar cf image.jar mars.jpg
```

After creating this file, erase `mars.jpg` from the current directory. Continue by executing the following command:

```
java -cp image.jar;. ClassLoaderDemo
```

`image.jar` and the current directory must be on the classpath so that the system classloader can load classes and resources as necessary.

This time, you'll observe a thrown `java.lang.ArrayIndexOutOfBoundsException` instance. This exception occurs because you're trying to populate a zero-length byte array, and this array has zero length because `(int) new File(IMAGE).length()` returns 0 (`mars.jpg` cannot be found—you just erased it).

One way to solve this problem is to remove `(int) new File(IMAGE).length()` and hardcode the image's length instead. Although this refactoring works, it isn't a good idea because it can result in a runtime exception should the size of the image change.

A better solution is to calculate the length of the file by reading the input stream, and then obtain a new input stream for reading file content. Listing C–13 demonstrates this technique.

*Listing C–13. Loading an image resource in a more robust manner and viewing a prefix of its content*

```
import java.io.File;
import java.io.InputStream;
import java.io.IOException;

public class ClassLoaderDemo
{
   final static String IMAGE = "mars.jpg";

   public static void main(String[] args)
   {
      System.out.println(ClassLoaderDemo.class.getClassLoader());
      InputStream is = ClassLoaderDemo.class.getResourceAsStream(IMAGE);
      if (is == null)
      {
         System.err.printf("%s not found%n", IMAGE);
         return;
      }
      try
      {
         byte[] image = new byte[getLength(is)];
         is = ClassLoaderDemo.class.getResourceAsStream(IMAGE);
         int _byte, i = 0;
         while ((_byte = is.read()) != -1)
            image[i++] = (byte) _byte;
         for (i = 0; i < 16; i++)
            System.out.printf("%02X ", image[i]);
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
   }

   static int getLength(InputStream is) throws IOException
   {
      byte[] buffer = new byte[1024];
      int length = 0;
      int bytesRead;
      while ((bytesRead = is.read(buffer)) > 0)
         length += bytesRead;
      return length;
   }
}
```

Listing C–13 introduces a `getLength(InputStream)` class method that reads the contents of the input stream into a fixed-size buffer in multiple steps. The number of bytes read is accumulated in a `length` variable whose value is returned from the method.

This time, `ClassLoaderDemo` will successfully read the contents of the image, whether or not the image is stored in a JAR file.

## Console

You're writing a console-based application that runs on the server. This application needs to prompt the user for a username and a password before granting access. Obviously, you don't want the password to be echoed to the console (i.e., a command window).

Before Java 6, you had no way to accomplish this task without resorting to the Java Native Interface (discussed later in this appendix). Although `java.awt.TextField` provides a `void setEchoChar(char c)` method for this purpose, this method is only appropriate for GUI-based/non-Android applications.

■ **NOTE:** AWT stands for *Abstract Window Toolkit*, a windowing toolkit that makes it possible to create crude user interfaces consisting of windows, buttons, textfields (via the `TextField` class), and so on. The AWT was released as part of Java 1.0 in 1995, and continues to be part of Java's standard class library. The AWT isn't supported by Android.

Java 6 responded to this need by introducing the `java.io.Console` class. This class declares methods that access the platform's character-based console device, but only when that device is associated with the current virtual machine.

■ **NOTE:** Whether or not a virtual machine has a console is dependent upon the underlying platform, and also upon the manner in which the virtual machine is invoked. When the virtual machine is started from an interactive command line without redirecting the standard input and output streams, its console will exist and (typically) will be connected to the keyboard and display from which the virtual machine was launched. When the virtual machine is started automatically (e.g., by a background job scheduler), it usually won't have a console.

To determine if a console is available, you need to call the `java.lang.System` class's `Console console()` class method, as follows:

```
Console console = System.console();
if (console == null)
{
   System.err.println("no console device is present");
   return;
}
```

This method returns a `Console` reference when a console is present; otherwise, it returns null. After verifying that null wasn't returned, you can use the reference to call `Console`'s methods—see Table C–2.

*Table C–2.* `Console` *Methods*

| Method | Description |
|---|---|
| `void flush()` | Flush the console, immediately writing any buffered output. |
| `Console format(String fmt, Object... args)` | Write a formatted string to the console's output stream. The `Console` reference is returned so that you can chain method calls together (for convenience). This method throws `java.util.IllegalFormatException` when the format string contains illegal syntax. (I discussed format strings in the context of the `java.util.Formatter` class in Chapter 10.) |
| `Console printf(String format, Object... args)` | An alias for `format()`. |
| `Reader reader()` | Return the `java.io.Reader` instance associated with the console. This instance can be passed to a `java.util.Scanner` constructor for sophisticated scanning/parsing. (I discuss `Scanner` later in this appendix.) |
| `String readLine()` | Read a single line of text from the console's input stream. The line (minus line-termination characters) is returned in a `String` object. This method returns null when the end of the stream is reached. It throws `java.io.IOError` when an I/O error occurs. |
| `String readLine(String fmt, Object... args)` | Write a formatted prompt string to the console's output stream, and then read a single line of text from its input stream. The line (minus line-termination characters) is returned in a `String` object. This method returns null when the end of the stream has been reached. It throws `IllegalFormatException` when the format string contains illegal syntax and `IOError` when an I/O error occurs. |
| `char[] readPassword()` | Read a password or passphrase from the console's input stream with echoing disabled. The password/passphrase (minus line-termination characters) is returned in a `char` array. This method returns null when the end of the stream has been reached. It throws `IOError` when an I/O error occurs. |

| Method | Description |
|---|---|
| `char[] readPassword(String fmt, Object... args)` | Write a formatted prompt string to the console's output stream, and then read a password/passphrase from its input stream with echoing disabled. The password/passphrase (minus line-termination characters) is returned in a `char` array. This method returns null when the end of the stream has been reached. It throws `IllegalFormatException` when the format string contains illegal syntax and `IOError` when an I/O error occurs. |
| `PrintWriter writer()` | Return the unique `java.io.PrintWriter` instance associated with this console. |

I've created a `Login` application that invokes `Console` methods to obtain a username and a password. Listing C–14 presents `Login`'s source code.

*Listing C–14. Simulating a username/password login operation*

```
import java.io.Console;
import java.io.IOError;

public class Login
{
   public static void main(String[] args)
   {
      Console console = System.console();
      if (console == null)
      {
         System.err.println("no console device is present");
         return;
      }
      try
      {
         String username = console.readLine("Username:");
         char[] pwd = console.readPassword("Password:");
         // Do something useful with the username and password. For something
         // to do, this application just prints out these values.
         System.out.println("Username = " + username);
         System.out.println("Password = " + new String(pwd));
         // Prepare username String for garbage collection. More importantly,
         // destroy the password.
         username = "";
         for (int i = 0; i < pwd.length; i++)
            pwd[i] = 0;
      }
      catch (IOError ioe)
      {
         console.printf("I/O error: %s\n", ioe.getMessage());
      }
   }
}
```

Compile Listing C–14 (`javac Login.java`) and run this application (`java Login`). It first prompts you to enter a username that's displayed, and then prompts for a password that isn't displayed. After entering the password, messages identifying the username and password are written to standard output.

■ **NOTE:** After obtaining and (presumably) doing something useful with the entered username and password, it's important to get rid of these items for security reasons. Most importantly, you'll want to remove the password by zeroing out the `char` array.

## Extension Mechanism and ServiceLoader API

Java provides an extension mechanism for extending the Java platform with new APIs. This mechanism is related to the ServiceLoader API, which Java 6 introduced as a replacement for the older undocumented `sun.misc.Service` and `sun.misc.ServiceConfigurationError` classes.

### Extension Mechanism

Java 1.2 introduced the *extension mechanism* as a standard and scalable way to extend the Java platform via standard *extensions*. These are custom APIs packaged in JAR files that are stored in the JDK's `jre/lib/ext` (Solaris/Linux) or `jre\lib\ext` (Windows) directory, or in the JRE's `lib/ext` (Solaris/Linux) or `lib\ext` (Windows) directory.

When you start an application that requires a standard extension, the runtime environment locates and loads the extension from this directory, without requiring the JAR file to be added to the classpath (see Chapter 5). Starting with Java 1.3, standard extensions are also known as *optional packages*.

The `java.ext.dirs` system property specifies the locations for installed optional packages. The default setting is the JRE's `lib/ext` (or `lib\ext`) directory. Beginning with Java 6, you can append to this system property the path to a platform-specific directory that's shared by all installed (Java 6 or higher) JREs. As Java 7's JDK documentation on the extension mechanism architecture specifies (see `http://download.oracle.com/javase/7/docs/technotes/guides/extensions/spec.html`), this path is one of the following:

- Windows: `%SystemRoot%\Sun\Java\lib\ext`
- Linux: `/usr/java/packages/lib/ext`
- Solaris: `/usr/jdk/packages/lib/ext`

Check out the “The Extension Mechanism” in the JDK documentation (`http://download.oracle.com/javase/7/docs/technotes/guides/extensions/index.html`) and in *The Java Tutorials* (`http://download.oracle.com/javase/tutorial/ext/index.html`) to learn more about the extension mechanism.

## ServiceLoader

Java supports the concepts of services and service providers for creating plugin architectures. For example, you might want to create a suite of compressors/decompressors (codecs) for use in compressing and decompressing audio and video content.

A *service* is a well-known set of interfaces and (usually abstract) classes. A *service provider* is a specific implementation of a service. The classes in a provider typically implement the interfaces and subclass the classes defined by the service itself. Service providers can be installed in an implementation of the Java platform in the form of extensions (i.e., JAR files placed into any of the usual extension directories). Providers also can be made available by adding them to the application’s classpath or by some other platform-specific means.

The standard class library reveals the presence of service provider interfaces (SPIs), which are packages of service types that vendors must implement (e.g., `java.nio.channels.spi`). Java 1.3 extended the JAR file format to support a standard way to specify SPI implementations as pluggable service providers by requiring vendors to store *provider configuration files*, which are text files that identify concrete provider classes, in a JAR file’s `META-INF/services` directory—the SPI implementation would also be stored, according to its package structure, in the JAR file. For example, to introduce a new JDBC driver, the vendor would create a JAR file with a `META-INF/services/java.sql.Driver` provider configuration file.

Various Java APIs initially used (and some might still be using) the undocumented `sun.misc.Service` class or another facility to look up services and instantiate service providers. Other APIs (e.g., Image I/O) rely on the Java 6-introduced ServiceLoader API for these tasks.

ServiceLoader consists of two classes: `java.util.ServiceLoader` and `java.util.ServiceConfigurationError`. The former class, whose generic type is `ServiceLoader<S>`, loads service providers via classloaders; the latter class describes an error that’s thrown when a problem occurs while a service provider is being loaded (e.g., `IOException` is thrown while reading the provider configuration file).

`ServiceLoader<S>` is a small class consisting of only six methods, which are described in Table C–3.

*Table C–3. `ServiceLoader` Methods*

| Method | Description |
|---|---|
| `Iterator<S> iterator()` | Lazily load all available service providers for this service loader's service. The iterator first returns providers from an internal cache. Then it lazily loads and instantiates remaining providers, storing them in the cache. |
| `static <S> ServiceLoader<S> load(Class<S> service)` | Create a new service loader for the given `service` type, using the current thread's context classloader to load service provider configuration files and service provider classes. |
| `static <S> ServiceLoader<S> load(Class<S> service, ClassLoader loader)` | Create a new service loader for the given `service` type, using the specified `loader` to load service provider configuration files and service provider classes. Pass `null` to use the system classloader. When there is no system classloader, the bootstrap classloader is used. |
| `static <S> ServiceLoader<S> loadInstalled(Class<S> service)` | Create a new service loader for the given `service` type, using the extension classloader to load service provider configuration files and service provider classes. When the extension classloader cannot be found, the system classloader is used. When there is no system classloader, the bootstrap classloader is used. |
| `void reload()` | Clear the cache so that all providers will be reloaded. Subsequent invocations of the `iterator()` method lazily look up and instantiate providers. Use this method in situations where providers can be dynamically installed while the virtual machine is running. |
| `String toString()` | Return a string that contains the fully qualified package name of the service passed to one of the "`load`" methods. |

Consider the previous codec example. Listing C–15 presents an interface that describes a service for obtaining a codec's encoder and decoder.

*Listing C–15. Representing a compressor/decompressor pair as a service*

```
package ca.tutortutor.codec;

public interface CodecService
{
   Encoder getEncoder(String charset);
   Decoder getDecoder(String charset);
}
```

Listing C–15 refers to `Encoder` and `Decoder` types, which represent encoders and decoders (respectively). Listing C–16 presents the `Encoder` interface (the `Decoder` interface is similar).

*Listing C–16. Encoders (and decoders) minimally providing a name*

```
package ca.tutortutor.codec;

public interface Encoder
{
   public String getName();
}
```

Codec services are obtained by passing `CodecService.class` to `ServiceLoader<S> load(Class<S> service)`. Because `ServiceLoader` provides the `Iterator<S> iterator()` method, the enhanced for loop statement can be used to iterate over the various codec services. Listing C–17 presents a `CodecManager` class that offers a demonstration.

*Listing C–17. Encoders (and decoders) minimally providing a name*

```
package ca.tutortutor.codec;

import java.util.ServiceLoader;

import ca.tutortutor.codec.CodecService;
import ca.tutortutor.codec.Decoder;
import ca.tutortutor.codec.Encoder;

public class CodecManager
{
   private static ServiceLoader<CodecService> serviceLoader =
      ServiceLoader.load(CodecService.class);

   public static Encoder getEncoder(String encodingName)
   {
      for (CodecService cs: serviceLoader)
      {
         Encoder enc = cs.getEncoder(encodingName);
         if (enc != null)
            return enc;
      }
      return null;
   }

   public static Decoder getDecoder(String encodingName)
   {
      for (CodecService cs: serviceLoader)
      {
         Decoder dec = cs.getDecoder(encodingName);
         if (dec != null)
            return dec;
      }
      return null;
   }
}
```

Listing C–17 reveals that `CodecManager` is the central point from which codec encoders and decoders are obtained, via the `getEncoder(String encodingName)` and `getDecoder(String encodingName)` class methods. The argument passed to each method is forwarded to the equivalent `CodecService` method to return the appropriate encoder or decoder that supports the argument.

Listings C–15 through C–17, and `Decoder.java`, collectively describe the Codec API that arbitrary Java applications can access. Complete the following steps to create this API:

1. Create a `ca` subdirectory of the current directory. Then create a `tutortutor` subdirectory of `ca`, followed by a `codec` subdirectory of `tutortutor`.

2. Copy Listings C–15 through C–17 to `CodecService.java`, `Encoder.java`, and `CodecManager.java` files, and store them in the previously created `codec` subdirectory. Also, create an equivalent `Decoder.java` file and store it in this subdirectory.

3. From the current directory, execute `javac ca/tutortutor/codec/*.java` to compile all of these source files.

4. Execute `jar cf codec.jar ca/tutortutor/codec/*.class` to create a `codec.jar` file to store this library.

There isn't much that you can do with this library until you have at least one codec. For example, consider Listing C–18.

*Listing C–18. Implementing the Apple Lossless Audio Codec (ALAC)*

```
package ca.tutortutor.alac;

import ca.tutortutor.codec.CodecService;
import ca.tutortutor.codec.Decoder;
import ca.tutortutor.codec.Encoder;

public class ALACCodecServiceImpl implements CodecService
{
   @Override
   public Encoder getEncoder(String name)
   {
      if (name.equalsIgnoreCase("ALAC"))
         return new ALACEncoder();
      return null;
   }

   @Override
   public Decoder getDecoder(String name)
   {
      if (name.equalsIgnoreCase("ALAC"))
         return new ALACDecoder();
      return null;
   }
}

class ALACEncoder implements Encoder
{
   @Override
   public String getName()
   {
      return "ALAC encoder";
   }
}

class ALACDecoder implements Decoder
{
   @Override
   public String getName()
   {
      return "ALAC decoder";
   }
}
```

Listing C–18's `ALACCodecServiceImpl` class implements the `CodecService` interface to implement the Apple Lossless Audio Codec (ALAC), at least as far as returning the codec encoder and decoder names are concerned.

Each of the `getEncoder(String name)` and `getDecoder(String name)` methods first interrogates its argument to determine whether or not this codec is appropriate. The codec is appropriate when the passed argument is `ALAC` (case doesn't matter).

If the argument is inappropriate, either method returns null, which prevents this codec from being used. Otherwise, an instance of the package-private `ALACEncoder` or `ALACDecoder` class is returned. It's this instance that provides the actual encoding or decoding implementation.

It's possible to store `ALACCodecServiceImpl` in a `ca.tutortutor.codec.alac` package. However, I've decided to store this class in a `ca.tutortutor.alac` package, for convenience.

Complete the following steps to create an `alac.jar` file that contains this class:

1. Create an `alac` subdirectory of the `tutortutor` subdirectory.
2. Copy Listing C–18 to an `ALACCodecServiceImpl.java` file and store it in the previously created `alac` subdirectory.
3. From the current directory, execute `javac ca/tutortutor/alac/*.java` to compile this source file.
4. Create a `META-INF` subdirectory of the `alac` subdirectory, followed by a `services` subdirectory of `META-INF`.

5. Create a `ca.tutortutor.codec.CodecService` text file containing `ca.tutortutor.alac.ALACCodecServiceImpl` in the `services` subdirectory.
6. From the current subdirectory, execute `jar cf alac.jar -C ca/tutortutor/alac META-INF ./ca/tutortutor/alac/*.class` to create an `alac.jar` file containing the ALAC codec. The `-C` option temporarily changes to the specified directory.

You need a simple application to test the codec service and the ALAC codec. Listing C–19 presents a `ServiceLoaderDemo` application that does just this.

*Listing C–19. Demoing the codec service*

```
import ca.tutortutor.codec.CodecManager;
import ca.tutortutor.codec.Decoder;
import ca.tutortutor.codec.Encoder;

public class ServiceLoaderDemo
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.out.println("usage  : java ServiceLoaderDemo codecname");
         System.out.println("example: java ServiceLoaderDemo ALAC");
         return;
      }
      Encoder enc = CodecManager.getEncoder(args[0]);
      if (enc != null)
         System.out.println("Encoder name: " + enc.getName());
      Decoder dec = CodecManager.getDecoder(args[0]);
      if (dec != null)
         System.out.println("Decoder name: " + dec.getName());
   }
}
```

Assuming that the current directory contains `ServiceLoaderDemo.java`, `codec.jar`, and `alac.jar` only, execute the following command to compile `ServiceLoaderDemo.java`:

```
javac -cp codec.jar ServiceLoaderDemo.java
```

You should now observe a `ServiceLoaderDemo.class` file in the current directory.

Now execute the following command line to run this application:

```
java -cp codec.jar;alac.jar;. ServiceLoaderDemo ALAC
```

You should observe the following output:

```
Encoder name: ALAC encoder
Decoder name: ALAC decoder
```

If you specify any value other than `ALAC` (or `alac`, or any other case combination of these four letters), you should observe no output, because codecs have yet to be created for them.

## File Revisited

Chapter 11 mentioned that Java 6 added to the `java.io.File` class `long getFreeSpace()`, `long getTotalSpace()`, and `long getUsableSpace()` methods that return space information about the *partition* (a platform-specific portion of storage for a filesystem; for example, C:\) described by the `File` instance's abstract pathname. Android also supports these methods.

■ **NOTE:** Obtaining the amount of partition free space is important to installers and other applications. Until Java 6 arrived, the only portable way to accomplish this task was to guess by creating files of different sizes.

- `long getFreeSpace()` returns the number of unallocated bytes in the partition identified by this `File` object's abstract pathname; it returns zero when the abstract pathname doesn't name a partition.
- `long getTotalSpace()` returns the size (in bytes) of the partition identified by this `File` object's abstract pathname; it returns zero when the abstract pathname doesn't name a partition.
- `long getUsableSpace()` returns the number of bytes available to the current virtual machine on the partition identified by this `File` object's abstract pathname; it returns zero when the abstract pathname doesn't name a partition.

Although `getFreeSpace()` and `getUsableSpace()` appear to be equivalent, they differ in the following respect: unlike `getFreeSpace()`, `getUsableSpace()` checks for write permissions and other platform restrictions, resulting in a more accurate estimate.

■ **NOTE:** The `getFreeSpace()` and `getUsableSpace()` methods return a hint (not a guarantee) that a Java application can use all (or most) of the unallocated or available bytes. These values are a hint because a program running outside the virtual machine can allocate partition space, resulting in actual unallocated and available values being lower than the values returned by these methods.

Listing C–20 presents the source code to an application that demonstrates these methods. After obtaining an array of all available filesystem roots, this application obtains and outputs the free, total, and usable space for each partition identified by the array.

***Listing C–20. Outputting the free, usable, and total space on all partitions***

```
import java.io.File;
```

```
public class PartitionSpace
{
   public static void main(String[] args)
   {
      File[] roots = File.listRoots();
      for (File root: roots)
      {
         System.out.printf("Partition: %s%n", root);
         System.out.printf("Free space on this partition = %d%n",
                           root.getFreeSpace());
         System.out.printf("Usable space on this partition = %d%n",
                           root.getUsableSpace());
         System.out.printf("Total space on this partition = %d%n",
                           root.getTotalSpace());
         System.out.println("***");
      }
   }
}
```

Compile Listing C–20 (`javac PartitionSpace.java`) and run the application (`java PartitionSpace`). When run on my Windows 7 machine with a harddrive designated as C:, a DVD drive designated as D:, a USB drive designated as E:, and an extra drive designated as F:, I observe the following output (usually with different free/usable space amounts on C: and E:):

```
Partition: C:\
Free space on this partition = 374407311360
Usable space on this partition = 374407311360
Total space on this partition = 499808989184
***
Partition: D:\
Free space on this partition = 0
Usable space on this partition = 0
Total space on this partition = 0
***
Partition: E:\
Free space on this partition = 2856464384
Usable space on this partition = 2856464384
Total space on this partition = 8106606592
***
Partition: F:\
Free space on this partition = 0
Usable space on this partition = 0
Total space on this partition = 0
***
```

Chapter 11 also mentioned that Java 6 added to `File` `boolean setExecutable(boolean executable)`, `boolean setExecutable(boolean executable, boolean ownerOnly)`, `boolean setReadable(boolean readable)`, `boolean setReadable(boolean readable, boolean ownerOnly)`, `boolean setWritable(boolean writable)`, and `boolean setWritable(boolean writable, boolean ownerOnly)` methods that let you set the owner's or everybody's execute, read, and write permissions) for the file identified by the `File` object's abstract pathname. Android also supports these methods.

■ **NOTE:** Java 1.2 added a `boolean setReadOnly()` method to the `File` class, to mark a file or directory as read-only. However, a method to revert the file or directory to the writable state wasn't added. More importantly, until Java 6's arrival, `File` offered no way to manage an abstract pathname's read, write, and execute permissions.

- `boolean setExecutable(boolean executable, boolean ownerOnly)` enables (pass `true` to `executable`) or disables (pass `false` to `executable`) this abstract pathname's execute permission for its owner (pass `true` to `ownerOnly`) or everyone (pass `false` to `ownerOnly`). When the filesystem doesn't differentiate between the owner and everyone, this permission always applies to everyone. It returns true when the operation succeeds. It returns false when the user doesn't have permission to change this abstract pathname's access permissions, or when `executable` is `false` and the filesystem doesn't implement an execute permission.
- `boolean setExecutable(boolean executable)` is a convenience method that invokes the previous method to set the execute permission for the owner.
- `boolean setReadable(boolean readable, boolean ownerOnly)` enables (pass `true` to `readable`) or disables (pass `false` to `readable`) this abstract pathname's read permission for its owner (pass `true` to `ownerOnly`) or everyone (pass `false` to `ownerOnly`). When the filesystem doesn't differentiate between the owner and everyone, this permission always applies to everyone. It returns true when the operation succeeds. It returns false when the user doesn't have permission to change this abstract pathname's access permissions, or when `readable` is `false` and the filesystem doesn't implement a read permission.
- `boolean setReadable(boolean readable)` is a convenience method that invokes the previous method to set the read permission for the owner.
- `boolean setWritable(boolean writable, boolean ownerOnly)` enables (pass `true` to `writable`) or disables (pass `false` to `writable`) this abstract pathname's write permission for its owner (pass `true` to `ownerOnly`) or everyone (pass `false` to `ownerOnly`). When the filesystem doesn't differentiate between the owner and everyone, this permission always applies to everyone. It returns true when the operation succeeds. It returns false when the user doesn't have permission to change this abstract pathname's access permissions.
- `boolean setWritable(boolean writable)` is a convenience method that invokes the previous method to set the write permission for the owner.

Along with these methods, Java 6 retrofitted `File`'s `boolean canRead()` and `boolean canWrite()` methods, and introduced a `boolean canExecute()` method to return an abstract pathname's access permissions. These methods return true when the filesystem object identified by the abstract pathname exists and when the appropriate permission is in effect. For example, `canWrite()` returns true when the abstract pathname exists and when the application has permission to write to the file.

The `canRead()`, `canWrite()`, and `canExecute()` methods can be used to implement a simple utility that identifies which permissions have been assigned to an arbitrary filesystem object (e.g, a file or directory). This utility's source code is presented in Listing C–21.

*Listing C–21. Checking a filesystem object's permissions*

```
import java.io.File;

public class Permissions
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java Permissions filespec");
         return;
      }
      File file = new File(args[0]);
      System.out.printf("Checking permissions for %s%n", args[0]);
      System.out.printf("  Execute = %b%n", file.canExecute());
      System.out.printf("  Read = %b%n", file.canRead());
      System.out.printf("  Write = %b%n", file.canWrite());
   }
}
```

Compile Listing C–21 (`javac Permissions.java`) and run the application (`java Permissions`). Assuming the existence of a file named `x` (in the current directory), which is only readable and executable, `java Permissions x` generates the following output:

```
Checking permissions for x
  Execute = true
  Read = true
  Write = false
```

## Internationalization APIs

We tend to write software that reflects our cultural backgrounds. For example, a Spanish developer's application might present Spanish text, an Arabic developer's application might present a Hijri (Islamic) calendar, and a Japanese developer's application might display its currencies using the Japanese Yen currency symbol. Because cultural biases restrict the size of an application's audience, you might consider internationalizing your applications to reach a larger audience (and make more money).

*Internationalization* is the process of creating an application that automatically adapts to its current user's culture (without recompilation) so that the user can read text in the user's language and otherwise interact with the application without observing cultural biases. Java simplifies internationalization by supporting *Unicode* (a universal character set that encodes the various symbols making up the world's written languages) via the `char` keyword (see Chapter 2) and the `java.lang.Character` class (see Chapter 8), and by offering the APIs discussed in this section.

**NOTE:** Related to internationalization is the concept of *localization*, which is the adaptation of internationalized software to support a new culture by adding culture-specific elements (e.g., text strings that have been translated to that culture). Java already provides much of this support via various APIs. It also lets you extend its support via locale-sensitive services, which are not discussed in this appendix (for brevity).

### Locales

The `java.util.Locale` class is the centerpiece of the various Internationalization APIs. Instances of this class represent *locales*, which are geographical, political, or cultural regions.

`Locale` declares constants (e.g., `CANADA`) that describe some common locales. This class also declares three constructors for initializing `Locale` objects, in case you cannot find an appropriate `Locale` constant for a specific locale:

- `Locale(String language)` initializes a `Locale` instance to a `language` code; for example, `"fr"` for French.
- `Locale(String language, String country)` initializes a `Locale` instance to a `language` code and a `country` code; for example, `"en"` for English and `"US"` for United States.

- `Locale(String language, String country, String variant)` initializes a `Locale` instance to a `language` code, a `country` code, and a vendor- or browser-specific `variant` code; for example, `"de"` for German, `"DE"` for Germany, and `"WIN"` for Windows (or `"MAC"` for Macintosh).

The International Standards Organization (ISO) defines language and country codes. ISO 639 (`http://en.wikipedia.org/wiki/Iso_639`) defines language codes. ISO 3166 (`http://en.wikipedia.org/wiki/Iso_3166`) defines country codes. `Locale` supports both standards.

Variant codes are useful for dealing with platform differences. For example, font differences may force you to use different characters on Windows-, Linux-, and Unix-based operating systems (e.g., Solaris). Unlike language and country codes, variant codes are not standardized.

Although applications can create their own `Locale` objects (perhaps to let users choose from similar locales), they will often call API methods that work with the *default locale*, which is the locale made available to the virtual machine at startup. An application can call `Locale`'s `Locale getDefault()` class method when it needs to access this locale.

For testing or other purposes, the application can override the default locale by calling `Locale`'s `void setDefault(Locale locale)` class method. `setDefault()` sets the default locale to `locale`. However, passing `null` to `locale` causes `setDefault()` to throw `NullPointerException`.

Listing C–22 demonstrates `getDefault()` and `setDefault()`.

*Listing C–22. Viewing and changing the default locale*

```
import java.util.Locale;

public class MyLocale
{
   public static void main(String[] args)
   {
      System.out.println(Locale.getDefault());
      Locale.setDefault(Locale.US);
      System.out.println(Locale.getDefault());
   }
}
```

Compile Listing C–22 (`javac MyLocale.java`) and run the application (`java MyLocale`). When I run `MyLocale`, I observe the following output—my default locale is Canada (en_CA):

```
en_CA
en_US
```

You can change the default locale that's made available to the virtual machine by assigning appropriate values to the `user.language` and `user.country` system properties when you launch the application via the `java` tool. For example, the following `java` command line changes the default locale to fr_FR:

```
java -Duser.language=fr -Duser.country=FR MyLocale
```

As you continue to explore `Locale`, you'll discover additional useful methods. For example, the `String[] getISOLanguages()` class method returns an array of ISO 639 language codes (including former and changed codes), and the `String[] getISOCountries()` class method returns an array of ISO 3166 country codes.

## Resource Bundles

An internationalized application contains no hard-coded text or other locale-specific elements (e.g., a specific currency format). Instead, each supported locale's version of these elements is stored outside of the application.

Java is responsible for storing each locale's version of certain elements, such as currency formats. In contrast, it's your responsibility to store each supported locale's version of other elements, such as text, audio clips, and locale-sensitive images.

Java facilitates this element storage by providing *resource bundles*, which are containers that hold one or more locale-specific elements, and which are each associated with one and only one locale.

Many applications work with one or more resource bundle families. Each family consists of resource bundles for all supported locales and typically contains one kind of element (perhaps text, or audio clips that contain language-specific verbal instructions).

Each family also shares a common *family name* (also known as a *base name*); each of its resource bundles has a unique locale designation that's appended to the family name, to differentiate one resource bundle from another within the family.

Consider an internationalized text-based game application for English and French users. After choosing `game` as the family name, and `en` and `fr` as the English and French locale designations, you end up with the following complete resource bundle names:

- `game_en` is the complete resource bundle name for English users.
- `game_fr` is the complete resource bundle name for French users.

Although you can store all of your game's English text in the `game_en` resource bundle, you might want to differentiate between American and British text (e.g., elevator versus lift). This differentiation leads to the following complete resource bundle names:

- `game_en_US` is the complete resource bundle name for users who speak the United States version of the English language.
- `game_en_GB` is the complete resource bundle name for users who speak the British version of the English language.

An application loads its resource bundles by calling the various `getBundle()` class methods that are located in the abstract `java.util.ResourceBundle` class. For example, the application might call the following `getBundle()` factory methods:

- `ResourceBundle getBundle(String baseName)` loads a resource bundle using the specified `baseName` and the default locale. For example, `ResourceBundle resources = ResourceBundle.getBundle("game");` attempts to load the resource bundle whose base name is `game`, and whose locale designation matches the default locale. When the default locale is en_US, `getBundle()` attempts to load `game_en_US`.

- `ResourceBundle getBundle(String baseName, Locale locale)` loads a resource bundle using the specified `baseName` and `locale`. For example, `ResourceBundle resources = ResourceBundle.getBundle("game", new Locale("zh", "CN", "WIN"));` attempts to load the resource bundle whose base name is `game`, and whose locale designation is Chinese with a Windows variant. In other words, `getBundle()` attempts to load `game_zh_CN_WIN`.

> ■ **NOTE:** `ResourceBundle` is an example of a pattern that you'll discover throughout the Internationalization APIs. With few exceptions, each API is architected around an abstract entry-point class whose class methods return instances of concrete subclasses. For this reason, these class methods are also known as *factory methods*.

When the resource bundle identified by the base name and locale designation doesn't exist, the `getBundle()` methods search for the next closest bundle. For example, when the locale is en_US and `game_en_US` doesn't exist, `getBundle()` looks for `game_en`.

The `getBundle()` methods first generate a sequence of candidate bundle names for the specified locale (language1, country1, and variant1) and the default locale (language2, country2, and variant2) in the following order:

- baseName + "_" + language1 + "_" + country1 + "_" + variant1
- baseName + "_" + language1 + "_" + country1
- baseName + "_" + language1
- baseName + "_" + language2 + "_" + country2 + "_" + variant2
- baseName + "_" + language2 + "_" + country2
- baseName + "_" + language2

- baseName

Candidate bundle names in which the final component is an empty string are omitted from the sequence. For example, when country1 is an empty string, the second candidate bundle name is omitted.

The `getBundle()` methods iterate over the candidate bundle names to find the first name for which they can instantiate an actual resource bundle. For each candidate bundle name, `getBundle()` attempts to create a resource bundle as follows:

- It first attempts to load a class that extends the abstract `java.util.ListResourceBundle` class using the candidate bundle name. If such a class can be found and loaded using the specified classloader, is assignment compatible with `ResourceBundle`, is accessible from `ResourceBundle`, and can be instantiated, `getBundle()` creates a new instance of this class and uses it as the result resource bundle.

- Otherwise, `getBundle()` attempts to locate a properties file. It generates a pathname from the candidate bundle name by replacing all "`.`" characters with "`/`" and appending "`.properties`." It attempts to find a "resource" with this name using `ClassLoader.getResource()`. (Note that a "resource" in the sense of `getResource()` has nothing to do with the contents of a resource bundle; it's just a container of data, such as a file.) When `getResource()` finds a "resource," it attempts to create a new `java.util.PropertyResourceBundle` instance from its contents. When successful, this instance becomes the result resource bundle.

When no result resource bundle is found, `getBundle()` throws an instance of the `java.util.MissingResourceException` class; otherwise, `getBundle()` instantiates the bundle's parent resource bundle chain.

■ **NOTE:** The parent resource bundle chain makes it possible to obtain fallback values when resources are missing. The chain is built by using `ResourceBundle`'s `protected void setParent(ResourceBundle parent)` method.

`getBundle()` builds the chain by iterating over the candidate bundle names that can be obtained by successively removing variant, country, and language (each time with the preceding "`_`") from the complete resource bundle name of the result resource bundle.

■ **NOTE:** Candidate bundle names where the final component is an empty string are omitted.

With each candidate bundle name, `getBundle()` tries to instantiate a resource bundle, as just described. When it succeeds, it calls the previously instantiated resource bundle's `setParent()` method with the new resource bundle, unless the previously instantiated resource bundle already has a nonnull parent.

■ **NOTE:** `getBundle()` caches instantiated resource bundles and may return the same resource bundle instance multiple times.

`ResourceBundle` declares various methods for accessing a resource bundle's resources. For example, `Object getObject(String key)` gets an object for the given key from this resource bundle or one of its parent bundles.

`getObject()` first tries to obtain the object from this resource bundle using the `protected abstract handleGetObject()` method, which is implemented by concrete subclasses of `ResourceBundle` (e.g., `PropertyResourceBundle`).

If `handleGetObject()` returns null, and if a nonnull parent resource bundle exists, `getObject()` calls the parent's `getObject()` method. If still not successful, it throws `MissingResourceException`.

Two other resource-access methods are `String getString(String key)` and `String[] getStringArray(String key)`. These convenience methods are wrappers for `(String) getObject(key)` and `(String[]) getObject(key)`.

## Property Resource Bundles

A *property resource bundle* is a resource bundle backed by a *properties file*, a text file (with a `.properties` extension) that stores textual elements as a series of *key=value* entries. The *key* is a nonlocalized identifier that an application uses to obtain the localized *value*.

■ **NOTE:** Properties files are accessed via instances of the `java.util.Properties` class. In Chapter 9, I mentioned that the Preferences API (discussed later in this appendix) has made `Properties` largely obsolete. Property resource bundles prove that the `Properties` class isn't entirely obsolete.

`PropertyResourceBundle`, a concrete subclass of `ResourceBundle`, manages property resource bundles. You should rarely (if ever) need to work with this subclass. Instead, for maximum portability, you should only work with `ResourceBundle`, as Listing C–23 demonstrates.

***Listing C–23. Accessing a localized `elevator` entry in `game` resource bundles***

```
import java.util.ResourceBundle;
```

```
public class PropertyResourceBundleDemo
{
   public static void main(String[] args)
   {
      ResourceBundle resources = ResourceBundle.getBundle("game");
      System.out.println("elevator = " + resources.getString("elevator"));
   }
}
```

Listing C–23 refers to `ResourceBundle` instead of `PropertyResourceBundle`, which lets you easily migrate to `ListResourceBundle` as necessary. I use `getString()` instead of `getObject()` for convenience; text resources are stored in text-based properties files.

Compile Listing C–23 (`javac PropertyResourceBundleDemo.java`) and run this application (`java PropertyResourceBundleDemo`). You'll observe the following output:

```
Exception in thread "main" java.util.MissingResourceException: Can't find bundle for base name game,
locale en_CA
        at java.util.ResourceBundle.throwMissingResourceException(Unknown Source)
        at java.util.ResourceBundle.getBundleImpl(Unknown Source)
        at java.util.ResourceBundle.getBundle(Unknown Source)
        at PropertyResourceBundleDemo.main(PropertyResourceBundleDemo.java:7)
```

This exception is thrown because no property resource bundles exist. You can easily remedy this situation by copying Listing C–24 into a `game.properties` file, which is the basis for a property resource bundle.

*Listing C–24. A fallback `game.properties` resource bundle*

```
elevator=elevator
```

Assuming that `game.properties` is located in the same directory as `PropertyResourceBundleDemo.class`, execute `java PropertyResourceBundleDemo` and you'll see the following output:

```
elevator = elevator
```

Because my locale is en_CA, `getBundle()` first tries to load `game_en_CA.properties`. Because this file doesn't exist, `getBundle()` tries to load `game_en.properties`. Because this file doesn't exist, `getBundle()` tries to load `game.properties`, and succeeds.

Copy Listing C–25 into a file named `game_en_GB.properties`.

*Listing C–25. A `game` resource bundle for the en_GB locale*

```
elevator=lift
```

Execute `java -Duser.language=en -Duser.country=GB PropertyResourceBundleDemo`. This time, you should see the following output:

```
elevator = lift
```

With the locale set to en_GB, `getBundle()` first tries to load `game_en_GB.properties`, and succeeds.

Comment out `elevator = lift` by prepending a # character to this line (as in `#elevator = lift`). Then execute `java -Duser.language=en -Duser.country=GB PropertyResourceBundleDemo` and you should see the following output:

```
elevator = elevator
```

Although `getBundle()` loaded `game_en_GB.properties`, `getString()` (via `getObject()`) could not find an `elevator` entry. As a result, `getString()/getObject()` searched the parent resource bundle chain, encountering `game.properties`' `elevator=elevator` entry, whose `elevator` value was subsequently returned.

> **NOTE:** A common reason for `getString()` throwing `MissingResourceException` in a property resource bundle context is forgetting to append `.properties` to a properties file's name.

## List Resource Bundles

A *list resource bundle* is a resource bundle backed by a classfile, which describes a concrete subclass of `ListResourceBundle` (an abstract subclass of `ResourceBundle`). List resource bundles can store binary data (e.g., images or audio) as well as text. In contrast, property resource bundles can store text only.

> **NOTE:** When a property resource bundle and a list resource bundle have the same complete resource bundle name, the list resource bundle takes precedence over the property resource bundle. For example, when `getBundle()` is confronted with `game_en.properties` and `game_en.class`, it loads `game_en.class` instead of `game_en.properties`.

Listing C–26 demonstrates a list resource bundle by presenting a `flags_en_CA` class that extends `ListResourceBundle`.

***Listing C–26.** A resource bundle containing a small Canadian flag image and English/French text*

```
import java.awt.Toolkit;

import java.util.ListResourceBundle;

public class flags_en_CA extends ListResourceBundle
{
   private byte image[] =
   {
      (byte) 137,
      (byte) 80,
      (byte) 78,
      (byte) 71,
      (byte) 13,
      (byte) 10,
      (byte) 26,
      (byte) 10,
      (byte) 0,
      (byte) 0,
// ...
      (byte) 0,
      (byte) 0,
      (byte) 73,
      (byte) 69,
      (byte) 78,
      (byte) 68,
      (byte) 174,
      (byte) 66,
      (byte) 96,
      (byte) 130
   };

   private Object[][] contents =
   {
      { "flag", Toolkit.getDefaultToolkit().createImage(image) },
      { "msg", "Welcome to Canada! | Bienvenue vers le Canada !" },
      { "title", "CANADA | LA CANADA" }
   };

   public Object[][] getContents()
   {
      return contents;
   }
}
```

Listing C–26's `flags_en_CA` class, which must be declared `public`, describes a list resource bundle whose base name is `flags` and whose locale designation is `en_CA`. This class's `image` array stores a Portable Network Graphics (PNG)-based sequence of byte integers that describes an image of the Canadian flag, `contents` stores key/value pairs, and `getContents()` returns `contents`.

**NOTE:** For brevity, Listing C–26 doesn't present the complete `image` array with Canadian flag image data. You must obtain `flags_en_CA.java` from this appendix's companion code file (see the book's introduction for instructions on how to obtain this file) to get the complete listing.

The first key/value pair consists of a key named `flag` (which will be passed to `ResourceBundle`'s `getObject()` method) and an instance of the `java.awt.Image` class. This instance represents the flag image and is obtained with the help of the `java.awt.Toolkit` class and its `createImage()` utility method.

Listing C–27 shows you how to load the default `flags_en_CA` list resource bundle (or another list resource bundle via command-line arguments) and display its flag and text.

*Listing C–27. Obtaining and displaying a list resource bundle's flag image and text*

```
import java.awt.EventQueue;
import java.awt.Image;

import java.util.Locale;
import java.util.ResourceBundle;

import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ListResourceBundleDemo
{
   public static void main(String[] args)
   {
      Locale l = Locale.CANADA;
      if (args.length == 2)
         l = new Locale(args[0], args[1]);
      final ResourceBundle resources = ResourceBundle.getBundle("flags", l);
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         Image image = (Image) resources.getObject("flag");
                         String msg = resources.getString("msg");
                         String title = resources.getString("title");
                         ImageIcon ii = new ImageIcon(image);
                         JOptionPane.showMessageDialog(null,
                                                       msg,
                                                       title,
                                                       JOptionPane.PLAIN_MESSAGE,
                                                       ii);
                      }
                   };
      EventQueue.invokeLater(r);
   }
}
```

Listing C–27's `main()` method begins by selecting `CANADA` as its default `Locale`. If it detects that two arguments were passed on the command line, `main()` assumes that the first argument is the language code and the second argument is the country code, and creates a new `Locale` object based on these arguments as its default locale.

`main()` next attempts to load a list resource bundle by passing the `flags` base name and the previously chosen `Locale` object to `ResourceBundle`'s `getBundle()` method. Assuming that `MissingResourceException` isn't thrown, `main()` creates a runnable task on which to load resources from the list resource bundle and display them graphically.

`main()` relies on a windowing toolkit known as *Swing* to present a simple user interface that displays the flag and text. Because Swing is single-threaded, where everything runs on a special thread known as the *event-dispatch thread* (EDT), it's important that all Swing operations occur on this thread. `EventQueue.invokeLater()` makes this happen.

■ **NOTE:** Swing is built on top of the AWT and provides many sophisticated features. This windowing toolkit was officially released as part of Java 1.2, and continues to be part of Java's standard class library. Swing isn't supported by Android.

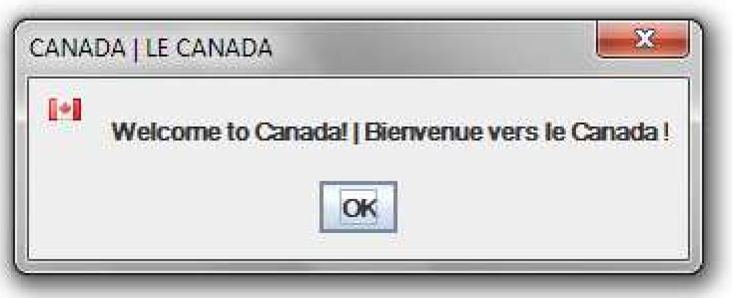
If you would like to learn more about Swing, I recommend the definitive *Java Swing, Second Edition*, by Marc Loy, Robert Eckstein, David Wood, James Elliott, and Brian Cole (O'Reilly Media, 2002; ISBN: 0596004087).

Shortly after `EventQueue.invokeLater()` is executed on the main thread, the EDT starts running and executes the runnable. This runnable first obtains the `Image` object from the list resource bundle by passing `flag` to `getObject()` and casting this method's return value to `Image`.

The runnable then obtains the `msg` and `title` strings by passing these keys to `getString()`, and converts the `Image` object to a `javax.swing.ImageIcon` instance. This instance is required by the subsequent `javax.swing.JOptionPane.showMessageDialog()` method call, which presents a simple message-oriented dialog box.

■ **NOTE:** `JOptionPane` is a Swing component that makes it easy to display a standard dialog box that prompts the user to enter a value or informs the user of something important.

Now that you understand how `ListResourceBundleDemo` works, obtain the complete `flags_en_CA.java` source file, compile its source code (`javac flags_en_CA.java`) and Listing C–27 (`javac ListResourceBundleDemo.java`), and execute the application (`java ListResourceBundleDemo`). Figure C–1 shows the resulting user interface on Windows 7.


CANADA | LE CANADA

Welcome to Canada! | Bienvenue vers le Canada !

*Figure C–1. This almost completely localized dialog box (OK isn't localized) displays Canada-specific resources on Windows 7.*

**NOTE:** I obtained the language translations for this section's examples from Yahoo! Babel Fish (`http://babelfish.yahoo.com/`), an online text translation service that no longer exists.

This book's accompanying code file also contains `flags_fr_FR.java`, which presents resources localized for the France locale. After compiling this source file, execute `java ListResourceBundleDemo fr FR` and you will see Figure C–2.

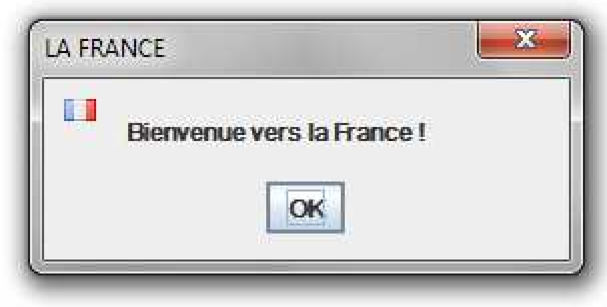


*Figure C–2. This almost completely localized dialog box displays France-specific resources on Windows 7.*

Finally, this book's accompanying code file also contains `flags_ru_RU.java`, which presents resources localized for the Russia locale. After compiling this source file (`javac -encoding Unicode flags_ru_RU.java`), execute `java ListResourceBundleDemo ru RU` and you will see Figure C–3.

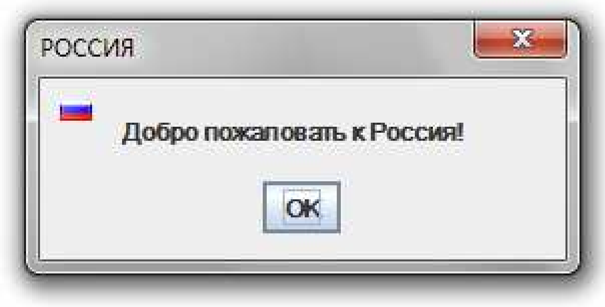


*Figure C–3. This almost completely localized dialog box displays Russia-specific resources on Windows 7.*

**NOTE:** Because I stored Russian characters verbatim (and not as Unicode escape sequences, such as `'\u0041'`), `flags_ru_RU.java` is a Unicode-encoded file and must be compiled via `javac -encoding Unicode flags_ru_RU.java`. Also, you'll need to ensure that appropriate Cyrillic fonts are installed to view the Russian text.

## Taking Advantage of Cache Clearing

Server applications are meant to run continuously; you'll probably lose customers and get a bad reputation if these applications fail often. As a result, it's preferable to change some aspect of their behavior interactively, rather than stop and restart them.

Before Java 6, you couldn't dynamically update the resource bundles for a server application that obtains localized text from these bundles and sends this text to clients. Because resource bundles are cached, a change to a resource bundle properties file, for example, would never be reflected in the cache, and ultimately not seen by the client.

Java 6 introduced `void clearCache()` and `void clearCache(ClassLoader loader)` class methods into `ResourceBundle` that make it possible to design a server application that clears out all cached resource bundles upon command. You would clear the cache after updating the appropriate resource bundle storage, which might be a file, a database table, or some other entity that stores resource data in some format.

To demonstrate cache clearing, I've created a date-server application that sends localized text and the current date (also localized) to clients. This application's source code is shown in Listing C–28.

*Listing C–28. A date server whose resource bundle cache can be cleared upon command*

```
import java.io.Console;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.PrintWriter;

import java.net.ServerSocket;
import java.net.Socket;

import java.text.MessageFormat;

import java.util.Date;
import java.util.Locale;
import java.util.MissingResourceException;
import java.util.ResourceBundle;

public class DateServer
{
   public final static int PORT = 5000;
   private ServerSocket ss;

   public DateServer(int port) throws IOException
   {
      ss = new ServerSocket(port);
   }

   public void runServer()
   {
      // This server application is console-based, as opposed to GUI-based.
      Console console = System.console();
      if (console == null)
      {
         System.err.println("unable to obtain system console");
         return;
      }
      // This would be a good place to log in the system administrator. For
      // simplicity, I've omitted this section.
      // Start a thread for handling client requests.
      Handler h = new Handler(ss);
      h.start();
      // Receive input from system administrator; respond to exit and clear
      // commands.
      while (true)
      {
         String cmd = console.readLine(">");
         if (cmd == null)
            continue;
         if (cmd.equals("exit"))
            System.exit(0);
         if (cmd.equals("clear"))
            h.clearRBCache();
      }
   }

   public static void main(String[] args) throws IOException
   {
      new DateServer(PORT).runServer();
   }
}

class Handler extends Thread
{
   private ServerSocket ss;
   private volatile boolean doClear;

   Handler(ServerSocket ss)
   {
      this.ss = ss;
   }

   void clearRBCache()
   {
      doClear = true;
   }

   @Override
   public void run()
   {
      ResourceBundle rb = null;
      while (true)
      {
         try
         {
            // Wait for a connection.
            Socket s = ss.accept();
            // Obtain the client's locale object.
            ObjectInputStream ois;
            ois = new ObjectInputStream(s.getInputStream());
            Locale l = (Locale) ois.readObject();
            // Prepare to output message back to client.
            PrintWriter pw = new PrintWriter(s.getOutputStream());
            // Clear ResourceBundle's cache upon request.
            if (doClear && rb != null)
            {
               rb.clearCache();
               doClear = false;
            }
            // Obtain a resource bundle for the specified locale. If resource
            // bundle cannot be found, the client is still waiting for
            // something, so send a ?.
            try
            {
               rb = ResourceBundle.getBundle("datemsg", l);
            }
            catch (MissingResourceException mre)
            {
               pw.println("?");
               pw.close();
               continue;
            }
            // Prepare a MessageFormat to format a locale-specific template
            // containing a reference to a locale-specific date.
            MessageFormat mf;
            mf = new MessageFormat(rb.getString("datetemplate"), l);
            Object[] args = { new Date() };
            // Format locale-specific message and send to client.
            pw.println(mf.format(args));
            // It's important to close the PrintWriter so that message is
            // flushed to the client socket's output stream.
            pw.close();
         }
         catch (Exception e)
         {
            System.err.println(e);
         }
      }
   }
}
```

After obtaining the console, the date server starts a handler thread to respond to clients requesting the current date formatted to their locale requirements—I discuss the `java.text.MessageFormat` class that's instantiated on this thread later in this appendix.

Following this thread's creation, you're repeatedly prompted to enter a command: `clear` (clear the cache) and `exit` (exit the application) are the only two possibilities. After changing a resource bundle, type `clear` to ensure that future `getBundle()` method calls initially retrieve their bundles from storage (and then the cache on subsequent method calls).

Compile Listing C–28 (`javac DateServer.java`) and run this application (`java DateServer`). You should observe a > prompt as the output.

The date server relies on resource bundles whose base name is `datetemplate`. I've created two bundles, stored in files named `datemsg_en.properties` and `datemsg_fr.properties`. The contents of the former file appear in Listing C–29.

*Listing C–29. The contents of `datemsg_en.properties`*

```
datetemplate = The date is {0, date, long}.
```

After connecting to the date server, a date-client application sends the server a `Locale` object; the client receives a `String` object in response. When the date server doesn't support the locale (a resource bundle cannot be found), it returns a string consisting of a single question mark. Otherwise, the date server returns a string consisting of localized text. Listing C–30 presents the source code to a simple date-client application.

*Listing C–30. Communicating with the date server*

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.ObjectOutputStream;

import java.net.Socket;

import java.util.Locale;

public class DateClient
{
   final static int PORT = 5000;

   public static void main(String[] args)
   {
      try
      {
         // Establish a connection to the date server. For simplicity, the
         // server is assumed to run on the same machine as the client. The
         // PORT constants of both server and client must be the same.
         Socket s = new Socket("localhost", PORT);
         // Send the default locale to the date server.
         ObjectOutputStream oos;
         oos = new ObjectOutputStream(s.getOutputStream());
         oos.writeObject(Locale.getDefault());
         // Obtain and output the server's response.
         InputStreamReader isr;
```

```
            isr = new InputStreamReader(s.getInputStream());
            BufferedReader br = new BufferedReader(isr);
            System.out.println(br.readLine());
         }
         catch (Exception e)
         {
            System.err.println(e);
         }
      }
   }
```

For simplicity, the date client sends the default locale to the server. You can override this locale via the `java` tool's `-D` option. For example, assuming that you've previously started the date server, and have compiled Listing C–30 (`javac DateClient.java`), execute `java -Duser.language=fr -Duser.country="" DateClient` to send a `Locale("fr", "")` object to the server and receive a reply in French, as demonstrated below:

```
La date est 21 décembre 2012.
```

You can verify the usefulness of cache clearing by performing a simple experiment with the date server and date client applications. Before you begin this experiment, create a second copy of Listing C–29, in which "`Thee`" replaces "`The`". Make sure that the properties file containing `Thee` is in the same directory as the date server, and then follow these steps:

1. Start the date server.
2. Run the client using `en` as the locale (via `java DateClient`, when English is the default locale, or `java -Duser.language=en DateClient` otherwise). You should see a message beginning with "Thee date is."
3. Copy the Listing C–29 properties file to the server's directory.
4. Type `clear` at the server prompt.
5. Run the client using `en` as the locale. This time, you should see a message beginning with "The date is."

**CAUTION:** It's tempting to want to always invoke `clearCache()` before invoking `getBundle()`. However, doing so negates the performance benefit that caching brings to an application. For this reason, you should use `clearCache()` sparingly, as the date server application demonstrates.

## Taking Control of the getBundle() Methods

Before Java 6, ResourceBundle's getBundle() methods were hardwired to look for resource bundles as follows:

- Look for certain kinds of bundles: properties-based or class-based.
- Look in certain places: properties files or classfiles whose directory paths are indicated by fully qualified resource bundle base names.
- Use a specific search strategy: when a search based on a specified locale fails, perform the search using the default locale.
- Use a specific loading procedure: when a class and a properties file share the same candidate bundle name, the class is always loaded while the properties file remains hidden.

Furthermore, resource bundles were always cached.

Because this lack of flexibility prevents you from performing tasks such as obtaining resource data from sources other than properties files and classfiles (e.g., an XML file or a database), Java 6 reworked ResourceBundle to depend on a nested Control class. This nested class provides several callback methods that are invoked during the resource bundle search-and-load process. By overriding specific callback methods, you can achieve the desired flexibility. When none of these methods are overridden, the getBundle() methods behave as they always have.

Control offers the following methods:

- List<Locale> getCandidateLocales(String baseName, Locale locale) returns a list of candidate locales for the specified baseName and locale. NullPointerException is thrown when baseName or locale is null.
- static ResourceBundle.Control getControl(List<String> formats) returns a ResourceBundle.Control instance whose getFormats() method returns the specified formats. NullPointerException is thrown when the formats list is null, and IllegalArgumentException is thrown when the list of formats isn't known.
- Locale getFallbackLocale(String baseName, Locale locale) returns a fallback locale for further resource bundle searches (via ResourceBundle.getBundle()). NullPointerException is thrown when baseName or locale is null.

- `List<String> getFormats(String baseName)` returns a list of strings that identify the formats to be used in loading resource bundles that share the given `baseName`. `NullPointerException` is thrown when `baseName` is `null`.

- `static final ResourceBundle.Control getNoFallbackControl(List<String> formats)` returns a `ResourceBundle.Control` instance whose `getFormats()` method returns the specified `formats`, and whose `getFallBackLocale()` method returns null. `NullPointerException` is thrown when the `formats` list is `null`, and `IllegalArgumentException` is thrown when the list of `formats` isn't known.

- `long getTimeToLive(String baseName, Locale locale)` returns the time-to-live value for resource bundles loaded via this `ResourceBundle.Control` instance. `NullPointerException` is thrown when `baseName` or `locale` is `null`.

- `boolean needsReload(String baseName, Locale locale, String format, ClassLoader loader, ResourceBundle bundle, long loadTime)` determines when the expired cached bundle needs to be reloaded by comparing the last modified time with `loadTime`. It returns a true value (the bundle needs to be reloaded) when the last modified time is more recent then the `loadTime`. `NullPointerException` is thrown when `baseName`, `locale`, `format`, `loader`, or `bundle` is `null`.

- `ResourceBundle newBundle(String baseName, Locale locale, String format, ClassLoader loader, boolean reload)` creates a new resource bundle based on a combination of `baseName` and `locale`, and takes the `format` and `loader` into consideration. `NullPointerException` is thrown when `baseName`, `locale`, `format`, or `loader` is `null` (or when `toBundleName()`, which is called by this method, returns null). `IllegalArgumentException` is thrown when `format` is unknown or when the resource identified by the given parameters contains malformed data; `java.lang.ClassCastException` is thrown when the loaded class cannot be cast to `ResourceBundle`; `IllegalAccessException` is thrown when the class or its noargument constructor isn't accessible; `InstantiationException` is thrown when the class cannot be instantiated for some other reason; `ExceptionInInitializerError` is thrown when the class's static initializer fails; and `IOException` is thrown when an I/O error occurs while reading resources using any I/O operations.

- `String toBundleName(String baseName, Locale locale)` converts the specified `baseName` and `locale` into a bundle name whose components are separated by underscore characters. For example, when `baseName` is `MyResources` and `locale` is `en`, the resulting bundle name is `MyResources_en`. `NullPointerException` is thrown when `baseName` or `locale` is `null`.

- `String toResourceName(String bundleName, String suffix)` converts the specified `bundleName` to a resource name. Forward-slash separators replace package period separators; a period followed by `suffix` is appended to the resulting name. For example, when `bundleName` is `com.company.MyResources_en` and `suffix` is `properties`, the resulting resource name is `com/company/MyResources_en.properties`. `NullPointerException` is thrown when `bundleName` or `suffix` is `null`.

The `getCandidateLocales()` method is called by a `ResourceBundle.getBundle()` class method each time the class method looks for a resource bundle for a target locale. You can override `getCandidateLocales()` to modify the target locale's parent chain. For example, when you want your Hong Kong resource bundles to share traditional Chinese strings, make Chinese/Taiwan resource bundles the parent bundles of Chinese/Hong Kong resource bundles. *The Java Tutorial*'s "Customizing Resource Bundle Loading" lesson (`http://download.oracle.com/javase/tutorial/i18n/resbundle/control.html`) shows how to accomplish this task.

The `getFallbackLocale()` method is called by a `ResourceBundle.getBundle()` class method each time the class method cannot find a resource bundle based on `getFallbackLocale()`'s `baseName` and `locale` arguments. You can override this method to return null when you don't want to continue a search using the default locale.

The `getFormats()` method is called by a `ResourceBundle.getBundle()` class method when it needs to load a resource bundle that's not found in the cache. This returned list of formats determines if the resource bundles being sought during the search are classfiles only, properties files only, both classfiles and properties files, or some other application-defined formats. When you override `getFormats()` to return application-defined formats, you'll also need to override `newBundle()` to load bundles based on these formats.

Earlier, I demonstrated using `clearCache()` to remove all resource bundles from `ResourceBundle`'s cache. Rather than explicitly clear the cache, you can control how long resource bundles remain in the cache before they need to be reloaded, by using the `getTimeToLive()` and `needsReload()` methods. The `getTimeToLive()` method returns one of the following:

- A positive value representing the number of milliseconds that resource bundles loaded under the current `ResourceBundle.Control` instance can remain in the cache without being validated against their source data

- 0 when the bundles must be validated each time they are retrieved from the cache

- `ResourceBundle.Control.TTL_DONT_CACHE` when the bundles are not cached

- The default ResourceBundle.Control.TTL_NO_EXPIRATION_CONTROL when the bundles are not to be removed from the cache under any circumstance (apart from low memory, or when you explicitly clear the cache)

When a ResourceBundle.getBundle() class method finds an expired resource bundle in the cache, it calls needsReload() to determine whether or not the resource bundle should be reloaded. When this method returns true, getBundle() removes the expired resource bundle from the cache; a false return value updates the cached resource bundle with the time-to-live value returned from getTimeToLive().

The toBundleName() method is called from the default implementations of needsReload() and newBundle() when they need to convert a base name and a locale to a bundle name. You can override this method to load resource bundles from different packages instead of the same package.

For example, assume that MyResources.properties stores your application's default (base) resource bundle, and that you also have a MyResources_de.properties file for storing your application's German language resources. The default implementation of ResourceBundle.Control organizes these bundles in the same package. By overriding toBundleName() to change how these bundles are named, you can place them into different packages.

For example, you could have a com.company.app.i18n.base.MyResources package corresponding to the com/company/app/i18n/base/MyResources.properties resource file, and a com.company.app.i18n.de.MyResources package corresponding to the com/company/app/i18n/de/MyResources.properties file. You can learn how to do this by exploring a similar example in the Oracle/Sun Developer Network "International Enhancements in Java SE 6" article (www.oracle.com/technetwork/articles/javase/i18n-enhance-137163.html).

Although you will often subclass ResourceBundle.Control and override some combination of the callback methods, this isn't always necessary. For example, when you want to restrict resource bundles to classfiles only or to properties files only, you can invoke getControl() to return a ready-made ResourceBundle.Control (thread-safe singleton) object that takes care of this task. To get this object, you will need to pass one of the following ResourceBundle.Control constants to getControl():

- FORMAT_PROPERTIES, which describes an unmodifiable List<String> containing "java.properties"

- FORMAT_CLASS, which describes an unmodifiable List<String> containing "java.class"

- FORMAT_DEFAULT, which describes an unmodifiable List<String> containing "java.class" followed by "java.properties"

The first example in `ResourceBundle.Control`'s JDK documentation uses `getControl()` to return a `ResourceBundle.Control` instance that restricts resource bundles to properties files.

You can also invoke `getNoFallbackControl()` to return a ready-made `ResourceBundle.Control` instance that (in addition to restricting resource bundles to classfiles or properties files only) tells the new `getBundle()` methods to avoid falling back to the default locale when searching for a resource bundle. The `getNoFallbackControl()` method recognizes the same `formats` argument as `getControl()`; it returns a thread-safe singleton whose `getFallbackLocale()` method returns null.

## Break Iterators

Internationalized text-processing applications (e.g., word processors) need to detect logical boundaries within the text they're manipulating. For example, a word processor needs to detect these boundaries when highlighting a character, selecting a word to cut to the clipboard, moving the *caret* (text insertion point indicator) to the start of the next sentence, and wrapping a word at the end of a line.

Java provides the Break Iterator API with its abstract `java.text.BreakIterator` entry-point class to detect text boundaries.

`BreakIterator` declares the following class methods for obtaining break iterators that detect character, word, sentence, and line boundaries:

- `BreakIterator getCharacterInstance()`
- `BreakIterator getWordInstance()`
- `BreakIterator getSentenceInstance()`
- `BreakIterator getLineInstance()`

Each of these class methods returns a break iterator for the default locale. When you need a break iterator for a specific locale, you can call the following class methods:

- `BreakIterator getCharacterInstance(Locale locale)`
- `BreakIterator getWordInstance(Locale locale)`
- `BreakIterator getSentenceInstance(Locale locale)`
- `BreakIterator getLineInstance(Locale locale)`

Each of these class methods throws `NullPointerException` when its `locale` argument is `null`.

BreakIterator's locale-sensitive class methods might not support every locale. For this reason, you should only pass Locale objects that are also stored in the array returned from this class's Locale[] getAvailableLocales() class method (which is also declared in other entry-point classes) to the aforementioned class methods—this array contains at least Locale.US. Check out the following example:

```
Locale[] supportedLocales = BreakIterator.getAvailableLocales();
BreakIterator bi = BreakIterator.getCharacterInstance(supportedLocales[0]);
```

This example obtains BreakIterator's supported locales and passes the first locale (possibly Locale.US) to getCharacterInstance(Locale).

A BreakIterator instance has an imaginary cursor that points to the current boundary within a text string. This cursor position can be interrogated and the cursor moved from boundary to boundary with the help of the following BreakIterator methods:

- int current() returns the text boundary that was most recently returned by next(), next(int), previous(), first(), last(), following(int), or preceding(int). When any of these methods returns BreakIterator.DONE because either the first or the last text boundary has been reached, current() returns the first or last text boundary depending on which one was reached.
- int first() returns the first text boundary. The iterator's current position is set to this boundary.
- int following(int offset) returns the first text boundary following the specified character offset. When offset equals the last text boundary, following(int) returns BreakIterator.DONE and the iterator's current position is unchanged. Otherwise, the iterator's current position is set to the returned text boundary. The value returned is always greater than offset or BreakIterator.DONE.
- int last() returns the last text boundary. The iterator's current position is set to this boundary.
- int next() returns the text boundary following the current boundary. When the current boundary is the last text boundary, next() returns BreakIterator.DONE and the iterator's current position is unchanged. Otherwise, the iterator's current position is set to the boundary following the current boundary.
- int next(int n) returns the nth text boundary from the current boundary. When either the first or the last text boundary has been reached, next(int) returns BreakIterator.DONE and the current position is set to either the first or last text boundary depending on which one is reached. Otherwise, the iterator's current position is set to the new text boundary.

- `int preceding(int offset)` returns the last text boundary preceding the specified character `offset`. When `offset` equals the first text boundary, `preceding(int)` returns `BreakIterator.DONE` and the iterator's current position is unchanged. Otherwise, the iterator's current position is set to the returned text boundary. The returned value is always less than `offset` or equals `BreakIterator.DONE`. (This method was added to `BreakIterator` in Java 1.2. It could not be declared `abstract` because abstract methods cannot be added to existing classes; such methods would also have to be implemented in subclasses that might be inaccessible.)

- `int previous()` returns the text boundary preceding the current boundary. When the current boundary is the first text boundary, `previous()` returns `BreakIterator.DONE` and the iterator's current position is unchanged. Otherwise, the iterator's current position is set to the boundary preceding the current boundary.

Figure C–4 reveals that characters are located between boundaries, boundaries are zero-based, and the last boundary is the length of the string.

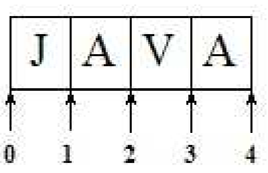


*Figure C–4. `JAVA`'s character boundaries appear as reported by the `next()` and `previous()` methods.*

`BreakIterator` also declares a `void setText(String newText)` method that identifies `newText` as the text to be iterated over. This method resets the cursor position to the beginning of this string.

Listing C–31 shows you how to use a character-based break iterator to iterate over a string's characters in a locale-independent manner.

*Listing C–31. Iterating over English/US and Arabic/Saudi Arabia strings*

```
import java.text.BreakIterator;

import java.util.Locale;

public class BreakIteratorDemo
{
   public static void main(String[] args)
   {
      BreakIterator bi = BreakIterator.getCharacterInstance(Locale.US);
      bi.setText("JAVA");
      dumpPositions(bi);
      bi = BreakIterator.getCharacterInstance(new Locale("ar", "SA"));
      bi.setText("\u0631\u0641\u0651");
      dumpPositions(bi);
   }

   static void dumpPositions(BreakIterator bi)
   {
      int boundary = bi.first();
      while (boundary != BreakIterator.DONE)
      {
         System.out.print(boundary + " ");
         boundary = bi.next();
      }
      System.out.println();
   }
}
```

Listing C–31's `main()` method first obtains a character-based break iterator for the United States locale. `main()` then calls the iterator's `setText()` method to specify `JAVA` as the text to be iterated over.

Iteration occurs in the `dumpPositions()` method. After calling `first()` to obtain the first boundary, this method uses a while loop to output the boundary and move to the next boundary (via `next()`) while the current boundary doesn't equal `BreakIterator.DONE`.

Because character iteration is straightforward for English words, `main()` next obtains a character-based break iterator for the Saudi Arabia locale, and uses this iterator to iterate over the characters in Figure C–5's Arabic version of "shelf" (as in shelf of books).

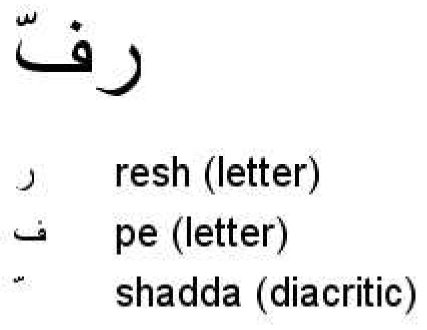


*Figure C–5. The letters and diacritic making up the Arabic equivalent of "shelf" are written from right to left.*

In Arabic, the word "shelf" consists of letters resh and pe, and diacritic shadda. A *diacritic* is an ancillary *glyph*, or mark on paper or other writing medium, added to a letter, or basic glyph. Shadda, which is shaped like a small written Latin w, indicates *gemination* (consonant doubling or extra length), which is *phonemic* (the smallest identifiable discrete unit of sound employed to form meaningful contrasts between utterances) in Arabic. Shadda is written above the consonant that's to be doubled, which happens to be pe in this example.

Compile Listing C–31 (`javac BreakIteratorDemo.java`) and run this application (`java BreakIteratorDemo`). You observe the following output:

```
0 1 2 3 4
0 1 3
```

The first output line reveals Figure C–4's character boundaries for the word `JAVA`. The second output line (0 comes before resh, 1 comes before pe) implies that you cannot move an Arabic word processor's caret on the screen once for every Unicode character. Instead, it's moved once for every *user character*, a logical character that can be composed of multiple Unicode characters, such as pe (`\u0641`) and shadda (`\u0651`).

> ■ **NOTE:** For examples of break iterators that iterate over words, sentences, and lines, check out the "Detecting Text Boundaries" section (`http://download.oracle.com/javase/tutorial/i18n/text/boundaryintro.html`) in *The Java Tutorials*.

## Collators

Applications perform string comparisons while sorting text. When an application targets English-oriented users, `String`'s `compareTo()` method is probably sufficient for comparing strings. However, this method's binary comparison of each string's Unicode characters isn't reliable for languages where the relative order of their characters doesn't correspond to the Unicode values of these characters. French is one example.

Java provides the Collator API with its abstract `java.text.Collator` entry-point class for making reliable comparisons.

`Collator` declares the following class methods for obtaining collators:

- `Collator getInstance()`
- `Collator getInstance(Locale locale)`

The first class method obtains a collator for the default locale; the second class method throws `NullPointerException` when its `locale` argument is `null`. As with `BreakIterator`, you should only pass `Locale` objects that are also stored in the array returned from `Collator`'s `Locale[] getAvailableLocales()` class method to `getInstance(Locale)`.

Listing C–32 shows how to use a collator to perform comparisons so that French words differing only in terms of accented characters are sorted into the correct order.

*Listing C–32. Using a collator to correctly order French words in the France locale*

```
import java.text.Collator;

import java.util.Arrays;
import java.util.Locale;

public class CollatorDemo
{
   public static void main(String[] args)
   {
      Collator en_USCollator = Collator.getInstance(Locale.US);
      Collator fr_FRCollator = Collator.getInstance(Locale.FRANCE);
      String[] words =
      {
         "côte", "coté", "côté", "cote"
      };
      Arrays.sort(words, en_USCollator);
      for (String word: words)
         System.out.println(word);
      System.out.println();
      Arrays.sort(words, fr_FRCollator);
      for (String word: words)
         System.out.println(word);
   }
}
```

In Listing C–32, each of the four words being sorted has a different meaning. For example, `côte` means coast and `côté` means side.

Compile Listing C–32 (`javac CollatorDemo.java`) and run this application (`java CollatorDemo`). I observe the following output in Windows Notepad:

```
cote
coté
côte
côté

cote
côte
coté
côté
```

The first four output lines show the order in which these words are sorted according to the en_US locale. This ordering isn't correct because it doesn't account for accents. In contrast, the final four output lines show the correct order when the words are sorted according to the fr_FR locale. Words are compared as if none of the characters contain accents, and then equal words are compared from right to left for accents.

■ **NOTE:** Learn about Collator's java.text.RuleBasedCollator subclass for creating custom collators when predefined collation rules don't meet your needs, and about improving collation performance via java.text.CollationKey and Collator's CollationKey getCollationKey(String source) method by reading the "Comparing Strings" section (http://download.oracle.com/javase/tutorial/i18n/text/collationintro.html) in *The Java Tutorials*.

## Time Zones and Calendars

Chapter 10 introduced the java.util.Date class whose instances record temporal moments relative to the Unix epoch. Associated with dates are *time zones* (sets of geographical regions where each set shares a common number of hours relative to Greenwich Mean Time [GMT]) and *calendars* (systems of organizing the passage of time).

■ **NOTE:** GMT identifies the standard geographical location from where all time is measured. UTC, which stands for Coordinated Universal Time, is often specified in place of GMT.

Date's toString() method reveals that a time zone is part of a date. Java provides the abstract java.util.TimeZone entry-point class for obtaining instances of TimeZone subclasses. This class declares a pair of class methods for obtaining these instances:

- TimeZone getDefault()
- TimeZone getTimeZone(String ID)

The latter method returns a TimeZone instance for the time zone whose String identifier (e.g., "CST") is passed to ID.

■ **NOTE:** Some time zones take into account *daylight saving time*, the practice of temporarily advancing clocks so that afternoons have more daylight and mornings have less; for example, Central Daylight Time (CDT). Check out Wikipedia's "Daylight saving time" entry (http://en.wikipedia.org/wiki/Daylight_saving_time) to learn more about daylight saving time.

When you need to introduce a new time zone or modify an existing time zone, perhaps to deal with changes to a time zone's daylight saving time policy, you can work directly with TimeZone's java.util.SimpleTimeZone concrete subclass. SimpleTimeZone describes a raw offset from GMT and provides rules for specifying the start and end of daylight saving time.

Java 1.1 introduced the Calendar API with its abstract `java.util.Calendar` entry-point class as a replacement for `Date`. `Calendar` is intended to represent any kind of calendar. However, time constraints meant that only the Gregorian calendar could be implemented (via the concrete `java.util.GregorianCalendar` subclass) for version 1.1.

> ■ **NOTE:** Java 1.4 introduced support for the Thai Buddhist calendar via an internal class that subclasses `Calendar`. Java 6 introduced support for the Japanese Imperial Era calendar via the package-private `java.util.JapaneseImperialCalendar` class, which also subclasses `Calendar`.

`Calendar` declares the following class methods for obtaining calendars:

- `Calendar getInstance()`
- `Calendar getInstance(Locale locale)`
- `Calendar getInstance(TimeZone zone)`
- `Calendar getInstance(TimeZone zone, Locale locale)`

The first and third methods return calendars for the default locale; the second and fourth methods take the specified `locale` into account. Also, calendars returned by the first two methods are based on the current time in the default time zone; calendars returned by the last two methods are based on the current time in the specified time `zone`.

`Calendar` declares various constants, including `YEAR`, `MONTH`, `DAY_OF_MONTH`, `DAY_OF_WEEK`, `LONG`, and `SHORT`. These constants identify the year (four digits), month (0 represents January), current month day (1 through the month's last day), and current weekday (1 represents Sunday) calendar fields, and display styles (e.g., January versus Jan).

The first four constants are used with `Calendar`'s various `set()` methods to set calendar fields to specific values (set the year field to 2012, for example). They're also used with `Calendar`'s `int get(int field)` method to return field values, along with other field-oriented methods, such as `void clear(int field)` (unset a field).

The latter two constants are used with `Calendar`'s `String getDisplayName(int field, int style, Locale locale)` and `Map<String,Integer> getDisplayNames(int field, int style, Locale locale)` methods, which return short (Jan, for example) or long (January, for example) localized `String` representations of various field values.

Listing C–33 shows how to use various `Calendar` constants and methods to output calendar pages according to the en_US and fr_FR locales.

*Listing C–33. Outputting calendar pages*

```
import java.util.Calendar;
import java.util.Iterator;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

public class CalendarDemo
{
   public static void main(String[] args)
   {
      if (args.length < 2)
      {
         System.err.println("usage: java CalendarDemo yyyy mm [f|F]");
         return;
      }
      try
      {
         int year = Integer.parseInt(args[0]);
         int month = Integer.parseInt(args[1]);
         Locale locale = Locale.US;
         if (args.length == 3 && args[2].equalsIgnoreCase("f"))
            locale = Locale.FRANCE;
         showPage(year, month, locale);
      }
      catch (NumberFormatException nfe)
      {
         System.err.print(nfe);
      }
   }

   static void showPage(int year, int month, Locale locale)
   {
      if (month < 1 || month > 12)
         throw new IllegalArgumentException("month [" + month + "] out of " +
                                            "range [1, 12]");
      Calendar cal = Calendar.getInstance(locale);
      cal.set(Calendar.YEAR, year);
      cal.set(Calendar.MONTH, --month);
      cal.set(Calendar.DAY_OF_MONTH, 1);
      displayMonthAndYear(cal, locale);
      displayWeekdayNames(cal, locale);
      int daysInMonth = cal.getActualMaximum(Calendar.DAY_OF_MONTH);
      int firstRowGap = cal.get(Calendar.DAY_OF_WEEK)-1; // 0 = Sunday
      for (int i = 0; i < firstRowGap; i++)
         System.out.print("   ");
      for (int i = 1; i <= daysInMonth; i++)
      {
         if (i < 10)
            System.out.print(' ');
         System.out.print(i);
         if ((firstRowGap + i) % 7 == 0)
            System.out.println();
         else
            System.out.print(' ');
      }
      System.out.println();
   }

   static void displayMonthAndYear(Calendar cal, Locale locale)
   {
      System.out.println(cal.getDisplayName(Calendar.MONTH, Calendar.LONG,
                                            locale) + " " +
                                            cal.get(Calendar.YEAR));
   }

   static void displayWeekdayNames(Calendar cal, Locale locale)
   {
      Map<String, Integer> weekdayNamesMap;
      weekdayNamesMap = cal.getDisplayNames(Calendar.DAY_OF_WEEK,
                                            Calendar.SHORT, locale);
      String[] names = new String[weekdayNamesMap.size()];
      int[] indexes = new int[weekdayNamesMap.size()];
      Set<Map.Entry<String, Integer>> weekdayNamesEntries;
      weekdayNamesEntries = weekdayNamesMap.entrySet();
      Iterator<Map.Entry<String, Integer>> iter;
      iter = weekdayNamesEntries.iterator();
      while (iter.hasNext())
      {
         Map.Entry<String, Integer> entry = iter.next();
         names[entry.getValue() - 1] = entry.getKey();
         indexes[entry.getValue() - 1] = entry.getValue();
      }
      for (int i = 0; i < names.length; i++)
         for (int j = i; j < names.length; j++)
            if (indexes[j] == i + 1)
            {
               System.out.print(names[j].substring(0, 2) + " ");
               continue;
            }
      System.out.println();
   }
}
```

Listing C–33 is pretty straightforward, with the exception of `displayWeekdayNames()`. This method calls `Calendar`'s `getDisplayNames()` method to return a map of localized weekday names. Instead of returning a map where the keys are `java.lang.Integers` and the values are localized `Strings`, this map's keys are the localized `Strings`.

This would be fine if the keys were ordered (as in Sunday first and Saturday last, or lundi first and dimanche last). However, they're not ordered. To output these names in order, it's necessary to obtain a set of map entries, iterate over these entries and populate parallel arrays, and then iterate over these arrays to output the weekday names.

■ **NOTE:** A French calendar begins the week on lundi (Monday) and ends it on dimanche (Sunday). However, `Calendar` doesn't take this ordering into account.

Compile Listing C–33 (`javac CalendarDemo.java`) and run this application. For example, specifying `java CalendarDemo 2012 11` causes you to see the following calendar page for the en_CA locale:

```
November 2012
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

If you would like to see this page in the fr_FR locale, specify `java CalendarDemo 2012 11 f` or `java CalendarDemo 2012 11 F`:

```
novembre 2012
di lu ma me je ve sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

■ **NOTE:** `Calendar` declares a `Date getTime()` method that returns a calendar's time representation as a `Date` instance. `Calendar` also declares a `void setTime(Date date)` method that sets a calendar's time representation to the specified `date`.

## Formatters

Internationalized applications don't present unformatted numbers (including currencies and percentages), dates, and messages to the user. These items must be formatted according to the user's locale so they appear meaningful. To help with formatting, Java provides the abstract `java.text.Format` class and various subclasses.

### Number Formatters

The abstract `java.text.NumberFormat` entry-point class (a `Format` subclass) declares the following class methods to return formatters that format numbers as currencies, integers, numbers with decimal points, and percentages (and also to parse such values):

- `NumberFormat getCurrencyInstance()`
- `NumberFormat getCurrencyInstance(Locale locale)`
- `NumberFormat getIntegerInstance()`
- `NumberFormat getIntegerInstance(Locale locale)`
- `NumberFormat getInstance()`
- `NumberFormat getInstance(Locale locale)`
- `NumberFormat getNumberInstance()`
- `NumberFormat getNumberInstance(Locale locale)`
- `NumberFormat getPercentInstance()`
- `NumberFormat getPercentInstance(Locale locale)`

The `getInstance()` and `getInstance(Locale)` class methods are equivalent to `getNumberInstance()` and `getNumberInstance(Locale)`. They're present as a shorthand convenience to the longer-named `getNumberInstance()` methods.

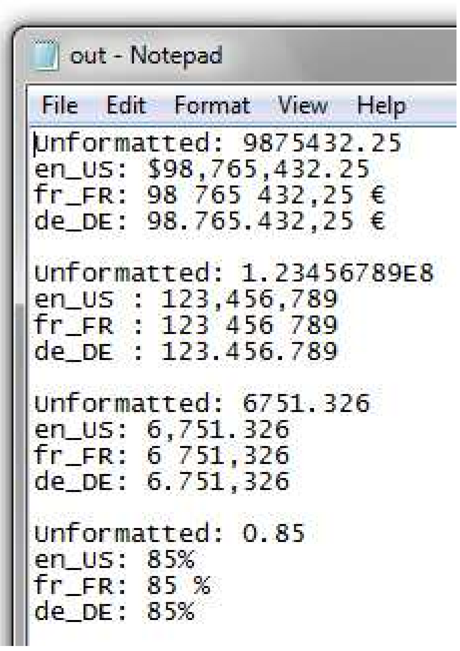
Listing C–34 shows you how to obtain and use number formatters to format numbers as currencies, integers, numbers with decimal points, and percentages for various locales.

*Listing C–34. Formatting numbers as currencies, integers, numbers with decimal points, and percentages*

```
import java.text.NumberFormat;

import java.util.Locale;

public class NumberFormatDemo
{
   public static void main(String[] args)
   {
      System.out.println("Unformatted: " + 9875432.25);
      formatCurrencies(Locale.US, 98765432.25);
      formatCurrencies(Locale.FRANCE, 98765432.25);
      formatCurrencies(Locale.GERMANY, 98765432.25);
      System.out.println();
      System.out.println("Unformatted: " + 123456789.0);
      formatIntegers(Locale.US, 123456789.0);
      formatIntegers(Locale.FRANCE, 123456789.0);
      formatIntegers(Locale.GERMANY, 123456789.0);
      System.out.println();
      System.out.println("Unformatted: " + 6751.326);
      formatNumbers(Locale.US, 6751.326);
      formatNumbers(Locale.FRANCE, 6751.326);
      formatNumbers(Locale.GERMANY, 6751.326);
      System.out.println();
      System.out.println("Unformatted: " + 0.85);
      formatPercentages(Locale.US, 0.85);
      formatPercentages(Locale.FRANCE, 0.85);
      formatPercentages(Locale.GERMANY, 0.85);
   }

   static void formatCurrencies(Locale locale, double amount)
   {
      NumberFormat nf = NumberFormat.getCurrencyInstance(locale);
      System.out.println(locale+": " + nf.format(amount));
   }

   static void formatIntegers(Locale locale, double amount)
   {
      NumberFormat nf = NumberFormat.getIntegerInstance(locale);
      System.out.println(locale+" : " + nf.format(amount));
   }

   static void formatNumbers(Locale locale, double amount)
   {
      NumberFormat nf = NumberFormat.getNumberInstance(locale);
      System.out.println(locale+": " + nf.format(amount));
   }

   static void formatPercentages(Locale locale, double amount)
   {
      NumberFormat nf = NumberFormat.getPercentInstance(locale);
      System.out.println(locale + ": " + nf.format(amount));
   }
}
```

Listing C–34 uses a `double` instead of a `java.math.BigDecimal` object to represent `9875432.25` as a currency, for simplicity and because this value can be represented exactly as a `double`.

Compile Listing C–34 (`javac NumberFormatDemo.java`) and run this application (`java NumberFormatDemo`). Figure C–6 shows the resulting output in the Windows Notepad editor.



```
Unformatted: 9875432.25
en_US: $98,765,432.25
fr_FR: 98 765 432,25 €
de_DE: 98.765.432,25 €

Unformatted: 1.23456789E8
en_US : 123,456,789
fr_FR : 123 456 789
de_DE : 123.456.789

Unformatted: 6751.326
en_US: 6,751.326
fr_FR: 6 751,326
de_DE: 6.751,326

Unformatted: 0.85
en_US: 85%
fr_FR: 85 %
de_DE: 85%
```

*Figure C–6. Windows Notepad reveals unformatted and formatted numeric output for the US, France, and Germany locales.*

`NumberFormat` declares `void setMaximumFractionDigits(int newValue)`, `void setMaximumIntegerDigits(int newValue)`, `void setMinimumFractionDigits(int newValue)`, and `void setMinimumIntegerDigits(int newValue)` methods to limit the number of digits that are allowed in a formatted number's integer or fraction. These methods are helpful for aligning numbers, as the following example demonstrates:

```
NumberFormat nf = NumberFormat.getInstance();
System.out.println(nf.format(123.4567)); // I observe 123.457
nf.setMaximumIntegerDigits(10);
nf.setMinimumIntegerDigits(6);
nf.setMaximumFractionDigits(2);
nf.setMinimumFractionDigits(2);
System.out.println(nf.format(123.4567)); // I observe 000,123.46
System.out.println(nf.format(80978.3)); // I observe 080,978.30
```

This example specifies that a number's integer portion cannot exceed ten digits, but must have a minimum of six digits. Leading zeros are output to meet the minimum. The example reveals that the fraction is rounded.

A concrete subclass of `NumberFormat` might enforce an upper limit on the value passed to `setMaximumFractionDigits(int)`, `setMaximumIntegerDigits(int)`, `setMinimumFractionDigits(int)`, or `setMinimumIntegerDigits(int)`. Call `getMaximumFractionDigits()`, `getMaximumIntegerDigits()`, `getMinimumFractionDigits()`, or `getMinimumIntegerDigits()` to find out if the value you specified has been accepted.

**NOTE:** When you need to create customized number formatters, you'll find yourself working with `NumberFormat`'s `java.text.DecimalFormat` subclass and this subclass's `java.text.DecimalFormatSymbols` companion class. The "Customizing Formats" section (`http://download.oracle.com/javase/tutorial/i18n/format/decimalFormat.html`) in *The Java Tutorials* introduces you to these classes.

## Date Formatters

The abstract `java.text.DateFormat` entry-point class (a `Format` subclass) provides access to formatters that format `Date` instances as dates or time values (and also to parse such values). This class declares the following class methods:

- `DateFormat getDateInstance()`
- `DateFormat getDateInstance(int style)`
- `DateFormat getDateInstance(int style, Locale locale)`
- `DateFormat getDateTimeInstance()`
- `DateFormat getDateTimeInstance(int dateStyle, int timeStyle)`
- `DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale locale)`
- `DateFormat getInstance()`
- `DateFormat getTimeInstance()`
- `DateFormat getTimeInstance(int style)`
- `DateFormat getTimeInstance(int style, Locale locale)`

The `getDateInstance()` class methods' formatters generate only date information, the `getTimeInstance()` class methods' formatters generate only time information, and the `getDateTimeInstance()` class methods' formatters generate date and time information.

The `dateStyle` and `timeStyle` fields determine how that information will be presented according to the following `DateFormat` constants:

- `SHORT` is completely numeric, such as 12.13.52 or 3:30pm
- `MEDIUM` is longer, such as Jan 12, 1952
- `LONG` is longer, such as January 12, 1952 or 3:30:32pm
- `FULL` is pretty completely specified, such as Tuesday, April 12, 1952 AD or 3:30:42pm PST.

Listing C–35 shows you how to format a `Date` instance that represents the Unix epoch according to the local time zone and the UTC time zone.

*Listing C–35. Formatting the Unix epoch*

```
import java.text.DateFormat;

import java.util.Date;
import java.util.Locale;
import java.util.TimeZone;

public class DateFormatDemo
{
   public static void main(String[] args)
   {
      Date d = new Date(0); // Unix epoch
      System.out.println(d);
      DateFormat df = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                     DateFormat.LONG,
                                                     Locale.US);
      System.out.println("Default format: " + df.format(d));
      df.setTimeZone(TimeZone.getTimeZone("UTC"));
      System.out.println("Taking UTC into account: " + df.format(d));
   }
}
```

Compile Listing C–35 (`javac DateFormatDemo.java`) and run this application (`java DateFormatDemo`). I observe the following output for the CST time zone:

```
Wed Dec 31 18:00:00 CST 1969
Default format: December 31, 1969 6:00:00 PM CST
Taking UTC into account: January 1, 1970 12:00:00 AM UTC
```

The Unix epoch, which is represented by passing `0` to the `Date(long)` constructor, is defined as January 1, 1970 00:00:00 UTC, but the first output line doesn't indicate this fact. Instead, it shows the epoch in my CST time zone, which is six hours away from GMT/UTC. To show the epoch correctly, I need to obtain the UTC time zone, which I accomplish by passing `"UTC"` to `TimeZone`'s `getTimeZone(String)` class method, and install this time zone instance into the date formatter with the help of `DateFormat`'s `void setTimeZone(TimeZone zone)` method.

■ **NOTE:** When you need to create customized date formatters, you'll find yourself working with `DateFormat`'s `java.text.SimpleDateFormat` subclass and this subclass's `java.text.DateFormatSymbols` companion class. The "Customizing Formats" section (`http://download.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html`) and the "Changing Date Format Symbols" section (`http://download.oracle.com/javase/tutorial/i18n/format/dateFormatSymbols.html`) in *The Java Tutorials* introduce you to these classes.

## Message Formatters

Applications often display simple and/or compound status and error messages to the user. A *simple message* consists of static (unchanging) text, whereas a *compound message* consists of static text and variable (changeable) data. For example, consider the following compound messages, where the underlined text identifies variable data:

```
10,536 visitors have visited your website since June 16, 2010.
Warning: 25 files have been modified in a suspicious manner.
Account balance is $10,567.00!
```

For a simple message, you obtain its text from a resource bundle and then display this text to the user. For a compound message, you obtain a *pattern* (template) for the message from a property resource bundle, pass this pattern along with the variable data to a message formatter to create a simple message, and display this message's text.

A *message formatter* is an instance of the concrete `MessageFormat` class (a `Format` subclass). (Unlike other APIs, `MessageFormat` doesn't have an abstract entry-point class with class methods for obtaining instances of subclasses.) Instead, this class declares the following constructors:

- `MessageFormat(String pattern)` initializes a `MessageFormat` instance to the specified `pattern` and the default locale. This constructor throws `IllegalArgumentException` when `pattern` is invalid.

- `MessageFormat(String pattern, Locale locale)` initializes a `MessageFormat` instance to the specified `pattern` and `locale`. This constructor throws `IllegalArgumentException` when `pattern` is invalid.

A pattern consists of static text and placeholders for variable data. Each placeholder is a brace-delimited sequence of a zero-based integer identifier, an optional format type, and an optional format style. Examples include `{0}` (insert text between braces), `{1, date}` (insert a date in default style), and `{2, number, currency}` (insert a currency).

For example, the previous set of compound messages can be converted into Listing C–36's patterns for the en_US locale.

*Listing C–36. Patterns in an `example_en_US.properties` file*

```
p1 = {0, number, integer} visitors have visited your website since {1, date, long}.
p2 = Warning: {0, number, integer} files have been modified in a suspicious manner.
p3 = Account balance is {0, number, currency}!
```

The same placeholders can be used in equivalent compound messages localized to another locale, such as Listing C–37's fr_FR locale.

*Listing C–37. Patterns in an `example_fr_FR.properties` file*

```
p1 = {0, number, integer} visiteurs ont visité votre site Web depuis le {1, date, long}.
p2 = Avertissement : {0, number, integer} dossiers ont été modifiés d'une façon soupçonneuse.
p3 = L''équilibre de compte est {0, number, currency} !
```

■ **NOTE:** An apostrophe (also known as a single quote) in a pattern starts a quoted string, in which, for example, `'{0, number, currency}` is treated as a literal string and isn't interpreted as a placeholder by the formatter. To ensure that this placeholder isn't treated as a literal string in the previous example, `L'équilibre`'s single quote must be doubled, which is why `L''équilibre` appears.

After creating a `MessageFormat` instance, where the pattern is obtained from a resource bundle, you typically create an array of `Objects` and call `MessageFormat`'s inherited `String format(Object obj)` method (from `Format`) with this array—passing an array of `Objects` to a method whose parameter type is `Object` works because arrays are `Objects`.

When `format()` is called, it scans the pattern, replacing each placeholder with the corresponding entry in the `Objects` array. For example, when `format()` finds a placeholder with integer identifier 0, it causes the zeroth entry in the `Objects` array to be formatted and then the formatted results to be output.

> **TIP:** You might find MessageFormat's `String format(String pattern, Object... arguments)` class method convenient for one-time formatting operations. This method is equivalent to executing `new MessageFormat(pattern).format(arguments, new StringBuffer(), null).toString()` on the default locale.

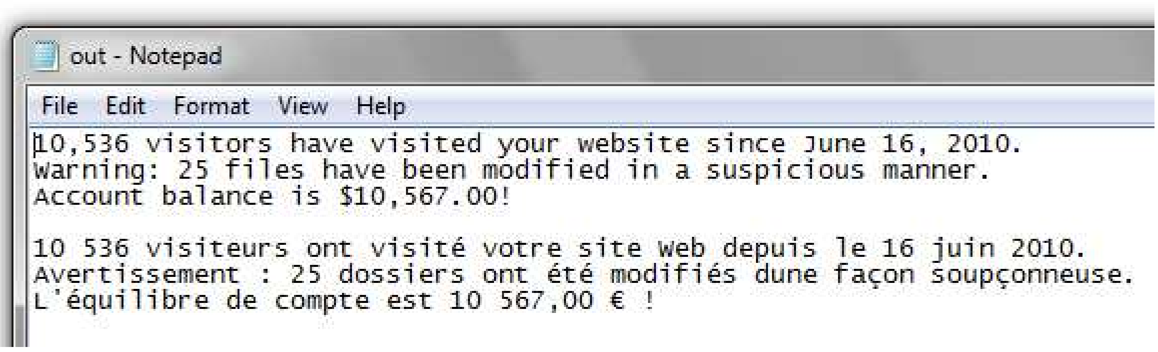
Listing C–38 demonstrates message formatting in the context of the previous examples' properties files and their localized patterns.

*Listing C–38. Formatting and outputting compound messages according to the en_US and fr_FR locales*

```java
import java.text.MessageFormat;

import java.util.Calendar;
import java.util.Locale;
import java.util.ResourceBundle;

public class MessageFormatDemo
{
   public static void main(String[] args)
   {
      dumpMessages(Locale.US);
      System.out.println();
      dumpMessages(Locale.FRANCE);
   }

   static void dumpMessages(Locale locale)
   {
      ResourceBundle rb = ResourceBundle.getBundle("example", locale);
      MessageFormat mf = new MessageFormat(rb.getString("p1"), locale);
      Calendar cal = Calendar.getInstance(locale);
      cal.set(Calendar.YEAR, 2010);
      cal.set(Calendar.MONTH, Calendar.JUNE);
      cal.set(Calendar.DAY_OF_MONTH, 16);
      Object[] args = new Object[] { 10536, cal.getTime() };
      System.out.println(mf.format(args));
      mf.applyPattern(rb.getString("p2"));
      args = new Object[] { 25 };
      System.out.println(mf.format(args));
      mf.applyPattern(rb.getString("p3"));
      args = new Object[] { 10567.0 };
      System.out.println(mf.format(args));
   }
}
```

Listing C–38 takes advantage of MessageFormat's `void applyPattern(String pattern)` method to override a previous pattern with a new pattern.

Compile Listing C–38 (`javac MessageFormatDemo.java`) and run this application (`java MessageFormatDemo`). You should observe Figure C–7's output, which I present via the Windows Notepad application.



```
10,536 visitors have visited your website since June 16, 2010.
Warning: 25 files have been modified in a suspicious manner.
Account balance is $10,567.00!

10 536 visiteurs ont visité votre site Web depuis le 16 juin 2010.
Avertissement : 25 dossiers ont été modifiés dune façon soupçonneuse.
L'équilibre de compte est 10 567,00 € !
```

*Figure C–7. Windows Notepad reveals compound messages formatted for the en_US and fr_FR locales.*

If you observe an exception instead of this output, the reason is probably due to `example_en_US.properties` and `example_fr_FR.properties` not being in the same directory as `MessageFormatDemo`.

> ■ **NOTE:** Some compound messages contain singular and plural words. For example, `Logging 1 message to x.log` and `Logging 2 messages to x.log` reveal singular and plural messages. Although you could specify pattern `Logging {0} message(s) to {1}`, it's not grammatically correct to state `Logging 2 message(s) to x.log`. The solution to this problem is to use the concrete `java.text.ChoiceFormat` class, a subclass of `NumberFormat` and a partner of `MessageFormat`, so that you can output `Logging 1 message to x.log` or `Logging 2 messages to x.log` depending on the numeric value passed to `{0}`. To learn how to use `ChoiceFormat`, check out the "Handling Plurals" section (`http://download.oracle.com/javase/tutorial/i18n/format/choiceFormat.html`) in *The Java Tutorials*.

## Parsing

The `Format` class has a dual personality in that it also declares a pair of `parseObject()` methods for parsing strings back into objects. Furthermore, it associates with a `java.text.ParseException` class whose instances are thrown when errors occur during parsing, and a `java.text.ParsePosition` class that keeps track of the current parsing position and error position indexes.

■ **NOTE:** `ParsePosition` declares `int getIndex()` and `void setIndex(int index)` methods for getting and setting the current parsing position, and `int getErrorIndex()` and `void setErrorIndex(int index)` methods for getting and setting the current error position.

`Format` declares the following `parseObject()` methods:

- `Object parseObject(String source)` parses `source` from the beginning and returns a corresponding object. Not all of `source`'s text may be parsed. This method throws `ParseException` when the beginning of this text cannot be parsed.

- `Object parseObject(String source, ParsePosition pos)` parses `source` starting at the current parsing position index stored in `pos` and returns a corresponding object. When parsing succeeds, `pos`'s current parsing position index is updated to the index after the last character used (parsing doesn't necessarily use all characters up to the end of the string), and the parsed object is returned. The updated `pos` can be used to indicate the starting point for the next call to this method. When an error occurs, `pos`'s current parsing position index isn't changed. However, its error position index is set to the index of the character where the error occurred, and null is returned. This method throws `NullPointerException` when `null` is passed to `pos`.

`parseObject(String)` invokes the abstract `parseObject(String, ParsePosition)` method as if by calling `parseObject(source, new ParsePosition(0))`.

`Format` subclasses such as `DateFormat` override `parseObject(String, ParsePosition)` to invoke one of their own `parse()` methods. For example, `MessageFormat` overrides `parseObject(String, ParsePosition)` and calls this method when necessary.

Although you should refrain from using speciality subclasses so that your application can adapt to the widest possible audience, you might find occasions to use such subclasses. For example, you might want to work directly with `SimpleDateFormat` to parse legacy date/time strings that were stored in a database according to a specific format. Listing C–39 demonstrates how you might accomplish this task with help from `SimpleDateFormat`.

***Listing C–39.** Parsing a date/time argument that must be formatted according to specific date format*

```
import java.text.ParseException;
import java.text.SimpleDateFormat;

import java.util.Date;

public class ParseDemo
{
   public static void main(String[] args) throws ParseException
   {
      if (args.length != 1)
      {
         System.err.println("usage: java ParseDemo yyyy-MM-dd HH:mm:ss z");
         return;
      }
      SimpleDateFormat sdf;
      sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss z");
      System.out.println(sdf.parse(args[0]));
   }
}
```

Listing C–39 first verifies that a single command-line argument was passed, and then instantiates `SimpleDateFormat` with a pattern string that identifies the pattern to follow for parsing. (The complete details on pattern string syntax are available in `SimpleDateFormat`'s Java documentation.)

Compile Listing C–39 (`javac ParseDemo.java`) and run this application, as follows:

```
java ParseDemo "2012-12-21 11:21:00 CST"
```

On my platform, I observe the following output:

```
Fri Dec 21 11:21:00 CST 2012
```

## Java Native Interface

The virtual machine insulates Java code from the underlying platform, to promote portability. However, there comes a point when code (primarily standard class library code) must bypass the virtual machine and communicate directly with the underlying platform to draw graphics on the screen, read/write files, and so on. Java provides the Java Native Interface to handle this communication.

**NOTE:** Java 1.1 introduced the Java Native Interface.

Your applications can also take advantage of the Java Native Interface when necessary. There are three common scenarios where you would leverage the Java Native Interface to bypass the virtual machine:

- The application must access a standard class library-unsupported platform-specific API (e.g., Win32 Joystick [`http://en.wikipedia.org/wiki/Joystick`] API).
- The application must interact with a *native library* (non-Java library) of legacy code (e.g., a library of statistics routines that predates Java).

- The application must perform complex calculations that exhibit better performance in native code (platform-specific instructions) than in Java code.

The *Java Native Interface (JNI)* is a native programming interface that lets Java code running in a virtual machine interoperate with native libraries written in other languages (e.g., C, C++, or even assembly language). The JNI is a bridge between the virtual machine and the platform.

■ **NOTE:** The decision to leverage the JNI shouldn't be made lightly. This technology restricts application execution to specific platforms (e.g., Windows only), is complex (subtle errors can crash the virtual machine), requires you to learn how to use the C language (or its C++ offspring), and causes application performance to suffer when native code is called frequently (native calls are often 2-3 times slower than Java calls).

## Creating a Hybrid Library

When working with the JNI, you're essentially creating a hybrid library consisting of a Java class and a native interface library.

### Java Class

The Java class uses the `native` reserved word to declare one or more *native methods* (methods whose bodies consist of native code stored in a native interface library). Consider the following example:

```
static native int joyGetNumDevs();
```

This example declares a native `joyGetNumDevs()` class method for returning the number of joystick devices (as a 32-bit integer) that are supported by the underlying Windows platform's joystick driver. When the Java compiler encounters `native`, it marks this method as a native method in the classfile. At runtime, the virtual machine knows that it must transfer execution to the equivalent native code when it encounters a call to `joyGetNumDevs()`.

The Java class also provides code to load the native interface library that provides a *native function* (a C-based function) counterpart for each declared native method. This code is often specified via a class initializer, as follows:

```
static
{
   System.loadLibrary("joystick");
}
```

This example's class initializer invokes System's void loadLibrary(String libname) class method, where libname is the name of the native library without a platform-specific prefix (e.g., lib) or extension (e.g., .so or .dll).

This method either loads the native library or throws an exception: NullPointerException is thrown when you pass null to libname, and java.lang.UnsatisfiedLinkError is thrown when the library doesn't exist (or doesn't contain the equivalent native function).

Because of the possibility for UnsatisfiedLinkError, which causes the application to terminate when thrown from a class initializer, I've often found it convenient to express the aforementioned code via an init() class method that returns a Boolean true/false value.

Listing C–40 presents a Joystick class that declares the joyGetNumDevs() native method and the init() class method.

*Listing C–40. The Java side of the joystick hybrid library*

```
class Joystick
{
   static boolean init()
   {
      try
      {
         System.loadLibrary("joystick");
         return true;
      }
      catch (UnsatisfiedLinkError ule)
      {
         return false;
      }
   }
   static native int joyGetNumDevs();
}
```

Listing C–40's init() method returns true when the native interface library is successfully loaded; otherwise, it returns false. You can call this method at application startup and gracefully degrade the application (or at least present a termination message to the user) when this method returns false.

## Building the Java Class

Compile Listing C–40 via the following command line:

```
javac Joystick.java
```

## Native Interface Library

The *native interface library* is a native library that contains JNI function calls and the native code that you want to execute (e.g., a Win32 function call that returns the number of supported joystick devices). This native code might serve as an interface between the native interface library and a legacy library.

Before you can code the native interface library, you need to generate a C-based include file that contains C function prototypes corresponding to the native methods. Accomplish this task with the help of the `javah` (Java header) tool. For example, the following command generates an include file for Listing C–40—you don't have to compile the source file before using `javah`:

```
javah Joystick
```

`javah` analyzes `Joystick.java` (you cannot specify the `.java` extension) and generates a file named `Joystick.h`. This include file is presented in Listing C–41.

*Listing C–41. Discovering the C language native function declaration*

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Joystick */

#ifndef _Included_Joystick
#define _Included_Joystick
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Joystick
 * Method:    joyGetNumDevs
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_Joystick_joyGetNumDevs
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

Listing C–41 begins with an `#include` C language preprocessor directive that includes the contents of a JDK file named `jni.h` into the source code. This header file declares various structures and JNI functions that a native function uses to cooperate with the Java platform, and is located in the `%JAVA_HOME%\include` directory.

The other significant item is JNIEXPORT jint JNICALL Java_Joystick_joyGetNumDevs(JNIEnv *, jclass);. This C function prototype names the native function, where the Joystick portion of the name identifies the class in which the corresponding native method is declared. It also describes the parameter list in terms of types only—C doesn't require names to be specified when declaring a function prototype.

The parameter of JNIEnv * type is a pointer to a C structure that makes it possible to call various JNI functions to perform useful work. No JNI functions are required in this example.

The parameter of jclass type identifies the class of which joyGetNumDevs() is a member. If joyGetNumDevs() was an instance method, this parameter would be of jobject type, which holds a reference to the current object on which the joyGetNumDevs() native method was invoked—think of the jobject parameter as containing this's value.

**NOTE:** Much of Listing C–41 focuses on making sure that the Java_Joystick_joyGetNumDevs(JNIEnv *, jclass) function can be invoked in the context of C++ without C++ name mangling (http://en.wikipedia.org/wiki/Name_mangling) getting in the way.

Listing C–42 presents the C code that implements Java_Joystick_joyGetNumDevs(JNIEnv *, jclass).

*Listing C–42. The C side of the joystick hybrid library*

```
#include <windows.h>
#include "Joystick.h"

JNIEXPORT jint JNICALL Java_Joystick_joyGetNumDevs(JNIEnv *pEnv, jclass clazz)
{
   return joyGetNumDevs();
}
```

Listing C–42 first specifies an #include <windows.h> directive, which provides access to Win32's joyGetNumDevs() multimedia function prototype, and an #include "Joystick.h" directive, which includes jni.h and the Java_Joystick_joyGetNumDevs(JNIEnv *, jclass) function prototype. The native function simply invokes joyGetNumDevs() and returns its value.

### Building the Native Interface Library

Various C/C++ compilers can be used to create the native interface library. I'm partial to the GNU Compiler Collection (see http://en.wikipedia.org/wiki/GNU_Compiler_Collection), which includes a C compiler.

You can obtain a version of this software for your Unix-oriented platform by pointing your browser to `http://gcc.gnu.org/` and following instructions. If your platform is Windows, you will need an environment such as MinGW (`http://en.wikipedia.org/wiki/Mingw`) to run this compiler. You can obtain a copy of MinGW by pointing your browser to `www.mingw.org` and following instructions.

Assuming a Windows platform, that Java 7 is installed into the `c:\progra~1\java\jdk1.7.0_06` home directory (`progra~1` is shorthand for `Program Files`), and that the current directory contains `Joystick.java`, `Joystick.h`, and `joystick.c` (with Listing C–42's contents), and that you're using MinGW, execute the following command (spread across two lines for readability) to compile `joystick.c` into `joystick.o`:

```
gcc -c -Ic:\progra~1\java\jdk1.7.0_06\include\ -Ic:\progra~1\java\jdk1.7.0_06\include\win32\
    joystick.c
```

The `-c` option specifies that an object file is to be created, and the `-I` option specifies the include paths for `jni.h` and `jni_md.h`—`jni_md.h` is a machine-dependent file located in the `%JAVA_HOME%\include\win32` directory. The end result is an object file named `joystick.o`.

Execute the following command line to turn this object file into a `joystick.dll` library:

```
gcc -Wl,--kill-at -shared -o joystick.dll joystick.o -lwinmm
```

The `-Wl,--kill-at` option specifies no name mangling (which leads to unsatisfied link errors), the `-shared` option specifies a Windows DLL target, the `-o` option specifies the output library name, and the `-l` option specifies a library to be included in the link step. The specified library is `winmm`, which is an import library needed by Windows to ensure that it can locate the proper DLL containing `joyGetNumDevs()`.

## Testing the Hybrid Library

Assuming that you've successfully created `Joystick.class` and the `joystick.dll` native interface library, you'll want to test this library's solitary native function. Listing C–43 presents the source code to a small application that accomplishes this task.

***Listing C–43.** Testing the joystick hybrid library*

```
class JoystickDemo
{
   public static void main(String[] args)
   {
      if (!Joystick.init())
      {
         System.err.println("unable to load joystick library");
         return;
      }
      System.out.printf("Number of joysticks = %d%n", Joystick.joyGetNumDevs());
   }
}
```

Compile Listing C–43 (`javac JoyStickDemo.java`). Then run this application (`java JoystickDemo`). On my Windows XP platform, I observed the following output:

```
Number of joysticks = 16
```

The Windows XP joystick driver supports a maximum of 16 joystick devices.

## Preferences API

Significant applications have *preferences*, which are configuration items. Examples include the location and size of the application's main window, and the locations and names of files that the application most recently accessed. Preferences are persisted to a file, to a database, or to some other storage mechanism so that they will be available to the application the next time it runs.

The simplest approach to persisting preferences is to use the Properties API, which consists of the `Properties` class. This class persists preferences as a series of *key*=*value* entries to text-based properties files. Although properties files are ideal for simple applications with few preferences, they have proven to be problematic with larger applications:

- Properties files tend to grow in size and the probability of name collisions among the various keys increases. This problem could be eliminated if properties files stored preferences in a hierarchy, but they're nonhierarchical.
- As an application grows in size and complexity, it tends to acquire numerous properties files with each part of the application associated with its own properties file. The names and locations of these properties files must be hard-coded in the application's source code.

Additionally, someone could directly modify these text-based properties files (perhaps inserting gibberish), causing the application that depends upon the modified properties file to crash unless it's properly coded to deal with this possibility. Also, properties files cannot be used on diskless computing platforms. Because of these problems, Java offers the Preferences API as a replacement for the Properties API.

The Preferences API lets you store preferences in a hierarchical manner so that you can avoid name collisions. Because this API is backend-neutral, it doesn't matter where the preferences are stored (a file, a database, or [on Windows platforms] the registry); you don't have to hardcode file names and locations. Also, there are no text-based files that can be modified, and Preferences can be used on diskless platforms.

This API uses trees of nodes to manage preferences. These nodes are the analogue of a hierarchical filesystem's directories. Also, preference name/value pairs stored under these nodes are the analogues of a directory's files. You navigate these trees in a similar manner to navigating a filesystem: specify an absolute path starting from the root node (`/`) to the node of interest, for example, `/window/location` and `/window/size`.

There are two kinds of trees: system and user. All users share the *system preference tree*, whereas the *user preference tree* is specific to a single user, which is generally the person who logged into the underlying platform. (The precise description of "user" and "system" varies from implementation to implementation of the Preferences API.)

Although the Preferences API's `java.util.prefs` package contains three interfaces (`NodeChangeListener`, `PreferencesChangeListener`, and `PreferencesFactory`), four regular classes (`AbstractPreferences`, `NodeChangeEvent`, `PreferenceChangeEvent`, and `Preferences`), and two exception classes (`BackingStoreException` and `InvalidPreferencesFormatException`), you mostly work with the `Preferences` class.

The `Preferences` class describes a node in a tree of nodes. To obtain a `Preferences` node, you must call one of the following class methods:

- `Preferences systemNodeForPackage(Class<?> c)`: Return the node whose path corresponds to the package containing the class represented by `c` from the system preference tree.

- `Preferences systemRoot()`: Return the root preference node of the system preference tree.

- `Preferences userNodeForPackage(Class<?> c)`: Return the node whose path corresponds to the package containing the class represented by `c` from the current user's preference tree.

- `Preferences userRoot()`: Return the root preference node of the current user's preference tree.

Listing C–44 demonstrates `systemNodeForPackage()`, along with `Preferences`' `void put(String key, String value)` and `String get(String key, String def)` methods for storing `String`-based preferences to and retrieving `String`-based preferences from the system preference tree. A default value must be passed to `get()` in case no value is associated with the key (which should not happen in this example).

*Listing C–44. Storing a single preference to and retrieving a single preference from the system preference tree*

```
package ca.tutortutor.examples;
```

```
import java.util.prefs.Preferences;

public class PrefsDemo
{
   public static void main(String[] args)
   {
      Preferences prefs = Preferences.systemNodeForPackage(PrefsDemo.class);
      prefs.put("version", "1.0");
      System.out.println(prefs.get("version", "unknown"));
   }
}
```

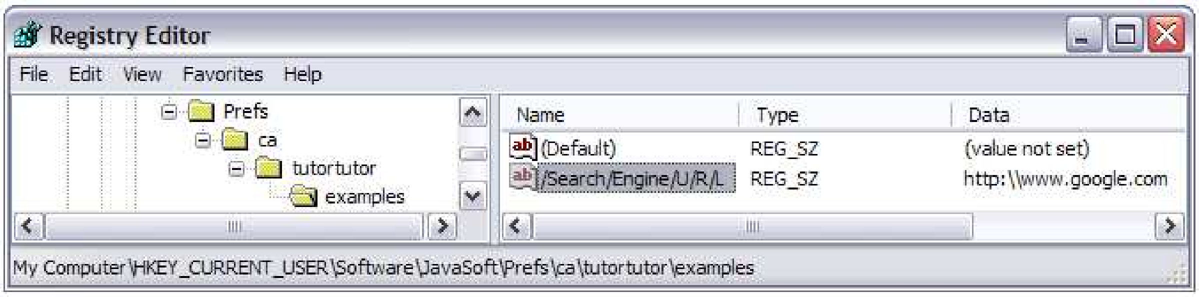
Create a `ca\tutortutor\examples` (or `ca/tutortutor/examples`) directory hierarchy under the current directory and copy `PrefsDemo.java` into `examples`.

Assuming that you haven't changed the current directory to another directory, execute `javac ca\tutortutor\examples\PrefsDemo.java` (or `javac ca/tutortutor/examples/PrefsDemo.java`) to compile the source file.

Run this application via `java ca.tutortutor.examples.PrefsDemo` and you'll observe the following output:

```
1.0
```

More interestingly, Figure C–8 reveals how this preference is stored in the Windows XP registry.



| Name | Type | Data |
|---|---|---|
| (Default) | REG_SZ | (value not set) |
| /Search/Engine/U/R/L | REG_SZ | http:\\www.google.com |

My Computer\HKEY_CURRENT_USER\Software\JavaSoft\Prefs\ca\tutortutor\examples

*Figure C–8. The Windows XP registry reveals the hierarchy for accessing the `version` key.*

`My Computer\HKEY_LOCAL_MACHINE\Software\JavaSoft\Prefs` is the path to the Windows XP registry area for storing system preferences. Under `Prefs`, you'll find a node for each package stored in this area. For example, `ca` identifies Listing C–44's `ca` package. Continuing, a hierarchy of nodes is stored under `ca`. Within the bottommost node (`examples`), you find an entry consisting of `version` (the key) and `1.0` (the value).

**NOTE:** For security reasons, the previous example doesn't work on Windows 7.

Listing C–45 provides a second example that works with the user preference tree and presents a more complex key.

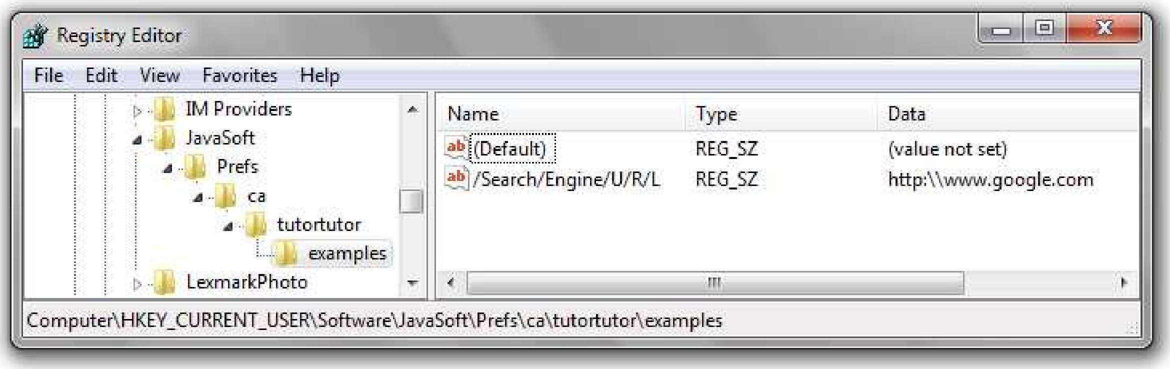
*Listing C–45. Storing a single preference to and retrieving a single preference from the current user's preference tree*

```
package ca.tutortutor.examples;

import java.util.prefs.Preferences;

public class PrefsDemo
{
   public static void main(String[] args)
   {
      Preferences prefs = Preferences.userNodeForPackage(PrefsDemo.class);
      prefs.put("SearchEngineURL", "http://www.google.com");
      System.out.println(prefs.get("SearchEngineURL", "http://www.bing.com"));
   }
}
```

When you run this application (`java ca.tutortutor.example.PrefsDemo`), it generates the following output:

```
http://www.google.com
```

More interestingly, Figure C–9 reveals how this preference is stored in the Windows 7 registry.



| Name | Type | Data |
|---|---|---|
| (Default) | REG_SZ | (value not set) |
| /Search/Engine/U/R/L | REG_SZ | http:\\www.google.com |

Computer\HKEY_CURRENT_USER\Software\JavaSoft\Prefs\ca\tutortutor\examples

*Figure C–9. The Windows 7 registry reveals the hierarchy for accessing the `SearchEngineURL` key.*

`Computer\HKEY_CURRENT_USER\Software\JavaSoft\Prefs` is the path to the Windows 7 registry area for storing user preferences. The Windows 7 registry encodes the `SearchEngineURL` key into `/Search/Engine/U/R/L`, because `Preferences` regards keys and node names as case sensitive but the Windows 7 registry doesn't.

■ **NOTE:** Learn more about the Preferences API from Ray Djajadinata's "Sir, What Is Your Preference?" article (`www.javaworld.com/javaworld/jw-08-2001/jw-0831-preferences.html`). Also, because Android defines its own preferences mechanism, check out Ilias Tsagklis's "Android Quick Preferences Tutorial" (`www.javacodegeeks.com/2011/01/android-quick-preferences-tutorial.html`).

## References and WeakHashMap

Chapter 3 introduced you to garbage collection, where you learned that the garbage collector removes an object from the heap when there are no more references to the object. This statement isn't completely true, as you will shortly discover.

Chapter 4 introduced you to `Object`'s `finalize()` method, where you learned that the garbage collector calls this method before removing an object from the heap. The `finalize()` method gives the object an opportunity to perform cleanup.

This section continues from where Chapters 3 and 4 left off by introducing you to Java's References API. After acquainting you with some basic terminology, it introduces you to the API's `Reference` and `ReferenceQueue` classes, followed by the API's `SoftReference`, `WeakReference`, and `PhantomReference` classes. These classes let applications interact with the garbage collector in limited ways.

■ **NOTE:** As well as this section, you will find Brian Goetz's "Java theory and practice: Plugging memory leaks with soft references" (`www.ibm.com/developerworks/java/library/j-jtp01246/index.html`) and "Java theory and practice: Plugging memory leaks with weak references" (`www.ibm.com/developerworks/java/library/j-jtp11225/index.html`) tutorials to be helpful in understanding the References API.

### Basic Terminology

When an application runs, its execution reveals a *root set of references*, which is a collection of local variables, parameters, class fields, and instance fields that currently exist and that contain (possibly null) references to objects. This root set changes over time as the application runs. For example, parameters disappear after a method returns.

Many garbage collectors identify this root set when they run. They use the root set to determine if an object is *reachable* (referenced, also known as *live*) or *unreachable* (not referenced). The garbage collector cannot collect reachable objects. Instead, it can only collect objects that, starting from the root set of references, cannot be reached.

■ **NOTE:** Reachable objects include objects that are indirectly reachable from root-set variables, which means objects that are reachable through live objects that are directly reachable from those variables. An object that is unreachable by any path from any root-set variable is eligible for garbage collection.

Beginning with Java 1.2, reachable objects are classified as strongly reachable, softly reachable, weakly reachable, and phantom reachable. Unlike strongly reachable objects, softly, weakly, and phantom reachable objects can be garbage collected.

Going from strongest to weakest, the different levels of reachability reflect the life cycle of an object. They are defined as follows:

- An object is *strongly reachable* when it can be reached from some thread without traversing any `Reference` objects. A newly created object (e.g., the object referenced by `d` in `Double d = new Double(1.0);`) is strongly reachable by the thread that created it.

- An object is *softly reachable* when it's not strongly reachable but can be reached by traversing a *soft reference* (a reference to the object where the reference is stored in a `SoftReference` object). The strongest reference to this object is a soft reference. When the soft references to a softly reachable object are cleared, the object becomes eligible for finalization (discussed in Chapter 4).

- An object is *weakly reachable* when it's neither strongly reachable nor softly reachable, but can be reached by traversing a *weak reference* (a reference to the object where the reference is stored in a `WeakReference` object). The strongest reference to this object is a weak reference. When the weak references to a weakly reachable object are cleared, the object becomes eligible for finalization. (Apart from the garbage collector being more eager to clean up the weakly reachable object, a weak reference is exactly like a soft reference.)

- An object is *phantom reachable* when it's neither strongly, softly, nor weakly reachable, it has been finalized, and it's referred to by some *phantom reference* (a reference to the object where the reference is stored in a `PhantomReference` object). The strongest reference to this object is a phantom reference.

- Finally, an object is unreachable, and therefore eligible for removal from memory during the next garbage collection cycle, when it's not reachable in any of the above ways.

The object whose reference is stored in a `SoftReference`, `WeakReference`, or `PhantomReference` object is known as a *referent*.

## Reference and ReferenceQueue

The References API consists of five classes located in the `java.lang.ref` package. Central to this package are `Reference` and `ReferenceQueue`:

- `Reference` is the abstract superclass of this package's concrete `SoftReference`, `WeakReference`, and `PhantomReference` subclasses.
- `ReferenceQueue` is a concrete class whose instances describe queue data structures. When you associate a `ReferenceQueue` instance with a `Reference` subclass object (`Reference` object, for short), the `Reference` object is added to the queue when the referent to which its encapsulated reference refers becomes garbage.

■ **NOTE:** You associate a `ReferenceQueue` object with a `Reference` object by passing the `ReferenceQueue` object to an appropriate `Reference` subclass constructor.

`Reference` is declared as generic type `Reference<T>`, where `T` identifies the referent's type. This class provides the following methods:

- `void clear()` assigns null to the stored reference; the `Reference` object on which this method is called isn't *enqueued* (inserted) into its associated reference queue (when there is an associated reference queue). (The garbage collector clears references directly; it doesn't call `clear()`. Instead, this method is called by applications.)
- `boolean enqueue()` adds the `Reference` object on which this method is called to the associated reference queue. This method returns true when this `Reference` object has become enqueued; otherwise, this method returns false—this `Reference` object was already enqueued or was not associated with a queue when created. (The garbage collector enqueues `Reference` objects directly; it doesn't call `enqueue()`. Instead, this method is called by applications.)
- `T get()` returns this `Reference` object's stored reference. The return value is null when the stored reference has been cleared, either by the application or by the garbage collector.

- `boolean isEnqueued()` returns true when this `Reference` object has been enqueued, either by the application or by the garbage collector. Otherwise, this method returns false—this `Reference` object was not associated with a queue when created.

■ **NOTE:** `Reference` also declares constructors. Because these constructors are package-private, only classes in the `java.lang.ref` package can subclass `Reference`. This restriction is necessary because instances of `Reference`'s subclasses must work closely with the garbage collector.

`ReferenceQueue` is declared as generic type `ReferenceQueue<T>`, where `T` identifies the referent's type. This class declares the following constructor and methods:

- `ReferenceQueue()` initializes a new `ReferenceQueue` instance.
- `Reference<? extends T> poll()` polls this queue to check for an available `Reference` object. When one is available, the object is removed from the queue and returned. Otherwise, this method returns immediately with a null value.
- `Reference<? extends T> remove()` removes the next `Reference` object from the queue and returns this object. This method waits indefinitely for a `Reference` object to become available, and throws `java.lang.InterruptedException` when this wait is interrupted.
- `Reference<? extends T> remove(long timeout)` removes the next `Reference` object from the queue and returns this object. This method waits until a `Reference` object becomes available or until `timeout` milliseconds have elapsed—passing 0 to `timeout` causes the method to wait indefinitely. If `timeout`'s value expires, the method returns null. This method throws `IllegalArgumentException` when `timeout`'s value is negative or `InterruptedException` when this wait is interrupted.

## SoftReference

The `SoftReference` class describes a `Reference` object whose referent is softly reachable. As well as inheriting `Reference`'s methods and overriding `get()`, this generic class provides the following constructors for initializing a `SoftReference` object:

- `SoftReference(T r)` encapsulates `r`'s reference. The `SoftReference` object behaves as a soft reference to `r`. No `ReferenceQueue` object is associated with this `SoftReference` object.

- `SoftReference(T r, ReferenceQueue<? super T> rq)` encapsulates `r`'s reference. The `SoftReference` object behaves as a soft reference to `r`. The `ReferenceQueue` object identified by `rq` is associated with this `SoftReference` object. Passing `null` to `rq` indicates a soft reference without a queue.

`SoftReference` is useful for implementing caches of objects that are expensive timewise to create (e.g., a database connection) and/or occupy significant amounts of heap space, such as large images. An image cache keeps images in memory (because it takes time to load them from disk) and ensures that duplicate (and possibly very large) images are not stored in memory.

The image cache contains references to image objects that are already in memory. If these references were strong, the images would remain in memory. You would then need to figure out which images are no longer needed and remove them from memory so that they can be garbage collected.

Having to manually remove images duplicates the work of a garbage collector. However, when you wrap the references to the image objects in `SoftReference` objects, the garbage collector will determine when to remove these objects (typically when heap memory runs low) and perform the removal on your behalf.

Listing C–46 presents an application that demonstrates soft references. It uses an instance of a `SoftCache` class (discussed later) to cache images, retrieves cached images before (hypothetically) drawing them, and re-caches images that are no longer cached.

*Listing C–46. Caching images*

```
import java.lang.ref.SoftReference;

class Image
{
   private byte[] image;

   private Image(String name)
   {
      image = new byte[1024 * 1024 * 100];
   }

   static Image getImage(String name)
   {
      return new Image(name);
   }
}

public class SoftReferenceDemo
{
   public static void main(String[] args)
   {
      SoftCache<Integer, Image> sc = new SoftCache<Integer, Image>();
      int i = 0;
      while (true)
      {
         System.out.printf("Putting large image %d into soft cache%n", i);
         sc.put(i, Image.getImage("large.png" + i));
         i++;
         int x = (int) (Math.random() * i);
         System.out.printf("Acquiring image %d from cache.%n", x);
         Image im = sc.get(x);
         if (im == null)
         {
            System.out.printf("Image %d no longer in cache. Re-caching.%n", x);
            sc.put(x, im = Image.getImage("large.png" + x));
         }
         System.out.printf("Drawing image %d%n", x);
         im = null; // Remove strong reference to image.
      }
   }
}
```

Listing C–46 declares an `Image` class that simulates the task of loading a large image, and declares a `SoftReferenceDemo` class that demonstrates the `SoftReference`-based caching of `Image` objects.

The `main()` method first instantiates the `SoftCache` class, which directly works with `SoftReference`. It then enters an infinite loop to continually cache new image objects and access image objects at random.

`SoftCache` provides a `get()` method that returns null when the image is no longer cached (or when no such image was ever put into the cache, which won't happen in this application). At this point, the image is re-obtained and re-cached via `SoftCache`'s `put()` method.

Listing C–47 presents `SoftCache`.

*Listing C–47. A generic class for caching arbitrary objects in a map*

```
import java.lang.ref.SoftReference;

import java.util.HashMap;

public class SoftCache<K, V>
{
   private HashMap<K, SoftReference<V>> map;

   public SoftCache()
   {
      map = new HashMap<K, SoftReference<V>>();
   }

   public V get(K key)
   {
      SoftReference<V> softRef = map.get(key);
      if (softRef == null)
         return null;
      return softRef.get();
   }

   public V put(K key, V value)
   {
      SoftReference<V> softRef = map.put(key, new SoftReference<V>(value));
      if (softRef == null)
         return null;
      V oldValue = softRef.get();
      softRef.clear();
      return oldValue;
   }

   public V remove(K key)
   {
      SoftReference<V> softRef = map.remove(key);
      if (softRef == null)
         return null;
      V oldValue = softRef.get();
      softRef.clear();
      return oldValue;
   }
}
```

Listing C–47 is pretty straightforward. One item that might be confusing is the `softRef.clear()` method call in each of the `put()` and `remove()` methods. This call makes the previously stored referent (whose `SoftReference` container instance is being overwritten, in the case of `put()`, or removed, in the case of `remove()`), which is being returned from `put()` or `remove()`, eligible for cleanup. However, the value will not be cleaned up when it's assigned to a variable, which would provide a strong reference to the value.

Compile Listing C–46 (`javac SoftReferenceDemo.java`) and run the application (`java SoftReferenceDemo`). You should discover output that's similar to the following Windows 7 output:

```
Acquiring image 6 from cache.
Image 6 no longer in cache. Re-caching.
Drawing image 6.
Putting large image 34 into soft cache.
Acquiring image 34 from cache.
Drawing image 34.
Putting large image 35 into soft cache.
Acquiring image 7 from cache.
Image 7 no longer in cache. Re-caching.
Drawing image 7.
Putting large image 36 into soft cache.
Acquiring image 1 from cache.
Image 1 no longer in cache. Re-caching.
Drawing image 1.
Putting large image 37 into soft cache.
Acquiring image 1 from cache.
Drawing image 1.
Putting large image 38 into soft cache.
Acquiring image 0 from cache.
```

```
Image 0 no longer in cache. Re-caching.
Drawing image 0.
Putting large image 39 into soft cache.
Acquiring image 34 from cache.
Image 34 no longer in cache. Re-caching.
Drawing image 34.
Putting large image 40 into soft cache.
Acquiring image 14 from cache.
Image 14 no longer in cache. Re-caching.
Drawing image 14.
Putting large image 41 into soft cache.
Acquiring image 38 from cache.
Drawing image 38.
Putting large image 42 into soft cache.
Acquiring image 39 from cache.
Drawing image 39.
Putting large image 43 into soft cache.
Acquiring image 27 from cache.
Image 27 no longer in cache. Re-caching.
Drawing image 27.
Putting large image 44 into soft cache.
Acquiring image 34 from cache.
Drawing image 34.
Putting large image 45 into soft cache.
```

## WeakReference

The WeakReference class describes a Reference object whose referent is weakly reachable. As well as inheriting Reference's methods, this generic class provides the following constructors for initializing a WeakReference object:

- WeakReference(T r) encapsulates r's reference. The WeakReference object behaves as a weak reference to r. No ReferenceQueue object is associated with this WeakReference object.

- WeakReference(T r, ReferenceQueue<? super T> rq) encapsulates r's reference. The WeakReference object behaves as a weak reference to r. The ReferenceQueue object identified by rq is associated with this WeakReference object. Passing null to rq indicates a weak reference without a queue.

WeakReference is useful for preventing memory leaks related to hashmaps. A memory leak occurs when you keep adding objects to a hashmap and never remove them. The objects remain in memory because the hashmap stores strong references to them.

Ideally, the objects should only remain in memory when they are strongly referenced from elsewhere in the application. When an object's last strong reference (apart from hashmap strong references) disappears, the object should be garbage collected.

This situation can be remedied by storing weak references to hashmap entries so they are discarded when no strong references to their keys exist. Java's `java.util.WeakHashMap` class, whose private `Entry` static member class extends `WeakReference`, accomplishes this task.

The `WeakHashMap` class provides a `java.util.Map` implementation that is based on weakly reachable keys. Because each key object is stored indirectly as the referent of a weak reference, the key is automatically removed from the map only after the garbage collector clears all weak references to the key (inside and outside of the map).

In contrast, value objects are stored via strong references (and should not strongly refer to their own keys, either directly or indirectly, because doing so prevents their associated keys from being discarded). When a key is removed from a map, its associated value object is also removed.

Listing C–48 presents an application that demonstrates weak references via the `WeakHashMap` class.

*Listing C–48. Detecting a weak hashmap entry's removal*

```
import java.util.Map;
import java.util.WeakHashMap;

class LargeObject
{
   private byte[] memory = new byte[1024 * 1024 * 50]; // 50 megabytes
}

class WeakReferenceDemo
{
   public static void main(String[] args)
   {
      Map<LargeObject, String> map = new WeakHashMap<>();
      LargeObject lo = new LargeObject();
      map.put(lo, "Large Object");
      System.out.println(map);
      lo = null;
      while (!map.isEmpty())
      {
         System.out.println("looping until map is empty");
         new LargeObject();
      }
      System.out.println(map);
   }
}
```

Listing C–48's `main()` method stores a 50MB `LargeObject` key and a `String` value in the weak hashmap, and then removes the key's strong reference by assigning `null` to `lo`. `main()` next enters a while loop that executes until the map is empty (`map.isEmpty()` returns true).

The loop first outputs a message. It then creates a `LargeObject` object, throwing away its reference. This activity should eventually cause the garbage collector to run and remove the map's solitary entry. However, if you discover an unending loop, insert `System.gc();` in this loop. This method call hints to the virtual machine that now might be a good time to run the garbage collector.

Compile Listing C–48 (`javac WeakReferenceDemo.java`) and run the application (`java WeakReferenceDemo`). You should discover output that's similar to the following Windows 7 output:

```
{LargeObject@6cc2060e=Large Object}
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
{}
```

■ **NOTE:** `WeakHashMap` is useful for avoiding memory leaks, as explained in Brian Goetz's article "Java Theory and Practice: Plugging Memory Leaks with Weak References" (`www.ibm.com/developerworks/java/library/j-jtp11225/`). Also, reference queues are more useful with `WeakReference` than they are with `SoftReference`. In the context of `WeakHashMap`, these queues provide notification of weakly referenced keys that have been removed. Code within `WeakHashMap` uses the information provided by the queue to remove all hashmap entries that no longer have valid keys so that the value objects associated with these invalid keys can be garbage collected. However, a queue associated with `SoftReference` can alert the application that heap space is beginning to run low.

## PhantomReference

The PhantomReference class describes a Reference object whose referent is phantom reachable. As well as inheriting Reference's methods and overriding get(), this generic class provides a single constructor for initializing a PhantomReference object:

- PhantomReference(T r, ReferenceQueue<? super T> rq) encapsulates r's reference. The PhantomReference object behaves as a phantom reference to r. The ReferenceQueue object identified by rq is associated with this PhantomReference object. Passing null to rq makes no sense because get() is overridden to return null and the PhantomReference object will never be enqueued.

Although you cannot access a PhantomReference object's referent (its get() method returns null), this class is useful because enqueuing the PhantomReference object signals that the referent has been finalized and its memory space has not yet been reclaimed. This signal lets you perform cleanup without using the finalize() method.

The finalize() method is problematic because the garbage collector requires at least two garbage collection cycles to determine if an object that overrides finalize() can be garbage collected. When the first cycle detects that the object is eligible for garbage collection, it calls finalize(). Because this method might perform resurrection (see Chapter 4), which makes the unreachable object reachable, a second garbage collection cycle is needed to determine if resurrection has happened. This extra cycle slows down garbage collection.

If finalize() isn't overridden, the garbage collector doesn't need to call that method, and considers the object to be finalized. Hence, the garbage collector requires only one cycle.

Although you cannot perform cleanup via finalize(), you can still perform cleanup via PhantomReference. Because there is no way to access the referent (get() returns null), resurrection cannot happen.

Listing C–49 shows how you might use PhantomReference to detect the finalization of a large object.

*Listing C–49. Detecting a large object's finalization*

```
import java.lang.ref.PhantomReference;
import java.lang.ref.ReferenceQueue;

class LargeObject
{
   private byte[] memory = new byte[1024 * 1024 * 50]; // 50 megabytes
}

public class PhantomReferenceDemo
{
   public static void main(String[] args)
   {
      ReferenceQueue<LargeObject> rq = new ReferenceQueue<LargeObject>();
      PhantomReference<LargeObject> pr;
      pr = new PhantomReference<LargeObject>(new LargeObject(), rq);
      while (rq.poll() == null)
      {
         System.out.println("waiting for first large object to be finalized");
         new LargeObject(); // Create another (unreferenced) LargeObject.
      }
      System.out.println("first large object finalized");
      System.out.println("pr.get() returns " + pr.get());
   }
}
```

Listing C–49 declares a `LargeObject` class whose private `memory` array occupies 50MB. If your virtual machine throws `java.lang.OutOfMemoryError` when you run this application, you might need to reduce the array's size.

The `main()` method first creates a `ReferenceQueue` object describing a queue onto which a `PhantomReference` object that initially contains a `LargeObject` reference will be enqueued. It then creates the `PhantomReference` object, passing a reference to a newly created `LargeObject` instance and a reference to the previously created `ReferenceQueue` object to the constructor.

`main()` now enters a polling loop, which first calls `poll()` to detect the finalization of the `LargeObject` instance. As long as this method returns null, meaning that the `LargeObject` instance is still unfinalized, the loop outputs a message and creates a new unreferenced `LargeObject` instance.

At some point, the garbage collector will run. It will clear the `PhantomReference` object's `LargeObject` reference and finalize the `LargeObject` object. The `PhantomReference` object is then enqueued onto the `rq`-referenced `ReferenceQueue`; `poll()` returns the `PhantomReference` object.

`main()` now exits the loop, outputs a message confirming the large object's finalization, and outputs `pr.get()`'s return value, which is null proving that you cannot access a `PhantomReference` object's referent.  At this point, any additional cleanup operations related to the finalized object (e.g., closing a file that was opened in the file's constructor but not otherwise closed) could be performed.

Compile Listing C–49 (`javac PhantomReferenceDemo.java`) and run the application (`java PhantomReferenceDemo`). You should discover output that's similar to the following Windows 7 output:

```
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
waiting for first large object to be finalized
first large object finalized
pr.get() returns null
```

**NOTE:** For a more useful example of `PhantomReference`, check out Keith D Gregory's "Java Reference Objects" blog post (`www.kdgregory.com/index.php?page=java.refobj`).

## StreamTokenizer

Compilers and other kinds of applications (e.g., text-based adventure games) derive meaning from human-entered text by first breaking a stream of characters into tokens, a task known as *tokenizing*. A *token* is a descriptor for a sequence of characters (e.g., an identifier), and this character sequence is also known as a *lexeme* because a compiler's tokenizing task is known as *lexical analysis*.

Although it's preferable to use Java's more powerful regular expressions capability to tokenize a character stream, a regular expressions API wasn't included in the standard class library when Java 1.0 was released, and so developers often resorted to the `java.io.StreamTokenizer` class that was included in Java 1.0—you might run across this class in legacy code.

According to its Java documentation, `StreamTokenizer` takes a reader and organizes its character stream into tokens, allowing these tokens to be read one at a time. This class performs this task by using an internal syntax table and various flags that can be set to various states. Stream tokenizers can recognize identifiers, numbers, quoted strings, and various comment styles.

Each read character (regarded as being in the range `'\u0000'` through `'\u00FF'`) is used to look up five possible character attributes: whitespace, alphabetic, numeric, string quote, and comment character. Each character can have zero or more of these attributes.

Additionally, a `StreamTokenizer` instance has the following four flags:

- Whether line terminators are to be returned as tokens or treated as whitespace that merely separates tokens.
- Whether C-style comments are to be recognized and skipped.
- Whether C++-style comments are to be recognized and skipped.
- Whether the characters of identifiers are converted to lowercase.

`StreamTokenizer` declares several nonconstant and constant fields. Each of these fields is described in Table C–4.

*Table C–4. `StreamTokenizer` Fields*

| Field | Description |
|---|---|
| `double nval` | Store a numeric token's integer or floating-point value. |
| `String sval` | Store a word token's string-of-characters value. |
| `static final int TT_EOF` | Identify end of file. |
| `static final int TT_EOL` | Identify end of line. |
| `static final int TT_NUMBER` | Identify number. Converted lexeme stored in `nval`. |
| `static final int TT_WORD` | Identify word. Lexeme stored in `sval`. |
| `int ttype` | Current token's type, which is one of `TT_EOF`, `TT_EOL`, `TT_NUMBER`, `TT_WORD`, a double-quote character when token identifies a string constant, or the value of a single character converted to an integer when none of the previous categories is appropriate. |

`StreamTokenizer` also declares several methods, which are described in Table C–5.

*Table C–5. `StreamTokenizer` Methods*

| Method | Description |
|---|---|
| `void commentChar(int ch)` | Identify `ch` as the start of a single-line comment. All characters starting with `ch` to the end of the current line are ignored. Any other attribute settings for the specified character are cleared. |

| Method | Description |
|---|---|
| `void eolIsSignificant(boolean flag)` | Determine whether (pass `true` to `flag`) or not (pass `false` to `flag`) the end-of-line character sequence is regarded as a token. A line is a sequence of characters ending with either a carriage-return character (`'\r'`) or a newline character (`'\n'`). Also, a carriage-return character followed immediately by a newline character is treated as a single end-of-line token. When you pass `true` to `flag`, `nextToken()` returns `TT_EOL` and sets `ttype` to this value when end-of-line is detected. When you pass `false` to `flag`, end-of-line characters are treated as whitespace and serve only to separate lexemes—this is the default. |
| `int lineno()` | Return the current line number. |
| `void lowerCaseMode(boolean flag)` | Determine whether (pass `true` to `flag`) or not (pass `false` to `flag`) word lexemes are automatically lowercased. When you pass `true` to `flag`, the value in the `sval` field is lowercased (i.e., a new `String` object is created with an equivalent lowercase value and assigned to `sval`) whenever a word lexeme is returned (`ttype` stores `TT_WORD`). When you pass `false` to `flag`, `sval` isn't modified—this is the default. |
| `int nextToken()` | Return the type of the next token from this tokenizer's input stream. The type is also stored in the `ttype` field. The token's value is stored in the `nval` or `sval` field. This method throws `IOException` when an I/O error occurs. |
| `void ordinaryChar(int ch)` | Specify that `ch` has no special significance (e.g., it doesn't denote the start of a comment or a word). When this tokenizer encounters this character, it treats it as a single-character lexeme and sets `ttype` to the character value. Making a line terminator character "ordinary" may interfere with the tokenizer's ability to count lines. The `lineno()` method may no longer reflect the presence of such terminator characters in its line count. |
| `void ordinaryChars(int low, int high)` | Specify that all characters from `low` to `high` have no special significance. When this tokenizer encounters this character, it treats it as a single-character lexeme and sets `ttype` to the character value. Making a line terminator character "ordinary" may interfere with the tokenizer's ability to count lines. The `lineno()` method may no longer reflect the presence of such terminator characters in its line count. |
| `void parseNumbers()` | Specify that this tokenizer should tokenize numbers. You would typically call this method after calling `resetSyntax()` when you wanted to deal with numbers or ordinary characters. |

| Method | Description |
|---|---|
| `void pushBack()` | Cause the next call to `nextToken()` to return the current value in the `ttype` field, and not to modify the value in the `nval` or `sval` field. |
| `void quoteChar(int ch)` | Specify that matching pairs of `ch`'s character value delimit string constants. When `nextToken()` encounters this character, `ttype` is set to the string delimiter and `sval` is set to the body of the string. When this character is encountered, a string is recognized, consisting of all characters after (but not including) the string quote character up to (but not including) the next occurrence of that same string quote character, or a line terminator, or end of file. The usual escape sequences (e.g., "`\n`" and "`\t`") are recognized and converted to single characters as the string is tokenized. Any other attribute settings for the specified character are cleared. |
| `void resetSyntax()` | Reset this tokenizer's internal syntax table so that all characters are regarded as ordinary. |
| `void slashSlashComments(boolean flag)` | Determine whether (pass `true` to `flag`) or not (pass `false` to `flag`) this tokenizer recognizes C++-style comments. When you pass `true` to `flag`, the tokenizer recognizes C++-style comments. Each occurrence of two consecutive slashes ("`/`") is treated as the beginning of a comment that extends to the end of the line. You would typically call this method after calling `resetSyntax()` when you want to ignore C++-style comments that are otherwise treated as ordinary characters. |
| `void slashStarComments(boolean flag)` | Determine whether (pass `true` to `flag`) or not (pass `false` to `flag`) this tokenizer recognizes C-style comments. When you pass `true` to `flag`, the tokenizer recognizes C-style comments. All text between successive occurrences of `/*` and `*/` is discarded. You would typically call this method after calling `resetSyntax()` when you want to ignore C-style comments that are otherwise treated as ordinary characters. |
| `String toString()` | Return the string representation of the current token value and the line number where it occurs. The precise string returned is unspecified, although the following example can be considered typical: `Token['a'], line 10` |
| `void whitespaceChars(int low, int high)` | Specify that all characters from `low` to `high` will be treated as whitespace. Whitespace characters serve only to separate lexemes in the character stream. Any other attribute settings for the characters in the specified range are cleared. |

| Method | Description |
|---|---|
| `void wordChars(int low, int high)` | Specify that all characters from `low` to `high` are word constituents (e.g., letters). A word lexeme consists of a word constituent followed by zero or more word constituents or number constituents. |

After invoking the `StreamTokenizer(Reader r)` constructor to create a tokenizer that tokenizes the given character stream, a typical application sets up the syntax tables via various method calls, and repeatedly loops by calling `nextToken()` in each loop iteration until this method returns `TT_EOF`. Listing C–50 demonstrates this usage pattern.

*Listing C–50. Tokenizing a file of characters*

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.StreamTokenizer;

public class Tokenizer
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java Tokenizer filename");
         return;
      }

      FileReader fr = null;
      try
      {
         fr = new FileReader(args[0]);
         StreamTokenizer st = new StreamTokenizer(fr);
         while (st.nextToken() != StreamTokenizer.TT_EOF)
            switch (st.ttype)
            {
               case '"':
                    System.out.printf("String: %s%n", st.sval);
                    break;

               case StreamTokenizer.TT_EOL:
                    System.out.println("end of line");
                    break;

               case StreamTokenizer.TT_NUMBER:
                    System.out.printf("number: %f%n", st.nval);
                    break;

               case StreamTokenizer.TT_WORD:
                    System.out.printf("word: %s%n", st.sval);
                    break;
               default:
                    System.out.printf("other: %c%n", st.ttype);
            }
      }
      catch (FileNotFoundException fnfe)
      {
         System.err.println("cannot find file: "+args[0]);
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: "+ioe.getMessage());
      }
      finally
      {
         if (fr != null)
            try
            {
               fr.close();
            }
            catch (IOException ioe)
            {
               ioe.printStackTrace();
            }
      }
   }
}
```

Compile Listing C–50 (`javac Tokenizer.java`) and run this application on its source code (`java Tokenizer Tokenizer.java`). You should observe the following output—only a short prefix of the output is shown for brevity:

```
word: import
word: java.io.FileNotFoundException
other: ;
word: import
word: java.io.FileReader
other: ;
word: import
word: java.io.IOException
other: ;
word: import
word: java.io.StreamTokenizer
other: ;
word: public
word: class
word: Tokenizer
other: {
word: public
word: static
word: void
word: main
other: (
word: String
```

```
other: [
other: ]
word: args
other: )
other: {
word: if
other: (
word: args.length
other: !
other: =
number: 1.000000
other: )
```

Although you should use regular expressions (which are far more powerful) with appropriate file I/O code in preference to `StreamTokenizer`, you might find it helpful to use this class for simple stream-based tokenizing tasks.

## StringTokenizer

Java 1.0 introduced the `java.util.StringTokenizer` class, which lets applications tokenize strings. `StringTokenizer` lets you specify a string to be tokenized and a set of delimiters that separate successive token values. This class declares three constructors for specifying these items:

- `StringTokenizer(String str)` constructs a string tokenizer for the specified string. The tokenizer uses the default delimiter set, which is " `\t\n\r\f`": the space character, the tab character, the newline character, the carriage-return character, and the form-feed character. Delimiter characters themselves won't be treated as token values. This constructor throws `NullPointerException` when you pass `null` to `str`.

- `StringTokenizer(String str, String delim)` constructs a string tokenizer for the specified string. The characters in the `delim` argument are the delimiters for separating token values. Delimiter characters themselves won't be treated as token values. This constructor throws `NullPointerException` when you pass `null` to `str`. Although it doesn't throw an exception when you pass `null` to `delim`, trying to invoke other methods on the resulting `StringTokenizer` instance may result in `NullPointerException`.

- `StringTokenizer(String str, String delim, boolean returnDelims)` constructs a string tokenizer for the specified string. All characters in the `delim` argument are the delimiters for separating token values. When the `returnDelims` flag is `true`, the delimiter characters are also returned as token values. Each delimiter is returned as a string of length one. When `returnDelims` is `false`, the delimiter characters are skipped and only serve as separators between token values. This constructor throws `NullPointerException` when you pass `null` to `str`. Although it doesn't throw an exception when you pass `null` to `delim`, invoking other methods on the resulting `StringTokenizer` instance may result in `NullPointerException`.

Additionally, `StringTokenizer` declares the following methods, which indicate that `StringTokenizer` instances maintain a current position for locating the next token value:

- `int countTokens()` returns the number of times that this tokenizer's `nextToken()` method can be called before it generates an exception. The current position is not advanced.

- `boolean hasMoreElements()` returns the same value as the `hasMoreTokens()` method. It exists because `StringTokenizer` implements the `java.util.Enumeration` interface (mentioned in Chapters 9 and 12).

- `boolean hasMoreTokens()` returns true when more token values are available from this tokenizer's string; otherwise, it returns false. When this method returns true, a subsequent call to the noargument `nextToken()` method will successfully return a token value.

- `Object nextElement()` returns the same value as the noargument `nextToken()` method, except that its return value is of type `Object` rather than `String`. It exists because `StringTokenizer` implements the `Enumeration` interface, and throws `java.util.NoSuchElementException` when there are no more token values.

- `String nextToken()` returns the next token value from this string tokenizer. It throws `NoSuchElementException` when there are no more token values.

- `String nextToken(String delim)` returns the next token value from this string tokenizer. First, the set of characters considered to be delimiters by this `StringTokenizer` object is changed to be the characters specified by `delim`. Then, the next token value in the string after the current position is returned. The current position is advanced beyond the recognized token value. The new delimiter set remains the default after this call. This method throws `NoSuchElementException` when there are no more token values and `NullPointerException` when you pass `null` to `delim`.

You would typically use `StringTokenizer` in a loop context, as follows:

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens())
   System.out.println(st.nextToken());
```

This loop generates the following output:

```
this
is
a
test
```

Alternatively, you could specify the following loop context:

```
StringTokenizer st = new StringTokenizer("this is a test");
Enumeration e = (Enumeration) st;
while (e.hasMoreElements())
   System.out.println(e.nextElement());
```

This loop generates the following output:

```
this
is
a
test
```

Of course, you would have to cast `nextElement()`'s `Object` return type to `String` before assigning the result to a `String` variable.

> ■ **NOTE:** `StringTokenizer` implements `Enumeration` so that you can create common code for enumerating token values and legacy vector/hashtable content.

Although you should be aware of `StringTokenizer` in case you encounter this class in legacy code, there's little point in discussing `StringTokenizer` further because this legacy class is only retained for compatibility reasons, and its use is discouraged in new code. Instead, you can obtain equivalent functionality by taking advantage of regular expressions.

For example, the `String` class declares `String[] split(String regex)` and `String[] split(String regex, int limit)` methods that let you specify regular expressions for returning the same token value sequences as `StringTokenizer`. The following code fragment demonstrates this alternative as a substitute for the previous code fragment:

```
String[] values = "this is a test".split("\\s");
for (String value: values)
   System.out.println(value);
```

This loop generates the following output:

```
this
is
```

```
a
test
```

## Exploring the Timer and TimerTask Classes

It's often necessary to schedule a *task* (a unit of work) for *one-shot execution* (the task runs only once) or for repeated execution at regular intervals. For example, you might schedule an alarm clock task to run only once (perhaps to wake you up in the morning) or schedule a nightly backup task to run at regular intervals. With either kind of task, you might want the task to run at a specific time in the future or after an initial delay.

You can use the Threads API (see Chapter 8) to accomplish task scheduling. However, Java 1.3 introduced a more convenient and simpler alternative in the form of `java.util.Timer` and `java.util.TimerTask` classes.

> **NOTE:** Game-oriented Android apps often use `Timer` and `TimerTask`. Check out stackoverflow's "Android timer? How?" topic (`http://stackoverflow.com/questions/4597690/android-timer-how`).

`Timer` provides a facility for threads to schedule `TimerTasks` for future execution in a background thread. Timer tasks may be scheduled for one-shot execution or for repeated execution at regular intervals.

Corresponding to each `Timer` object is a single background thread that's used to execute all of the timer tasks sequentially. This thread is known as the *task-execution thread*.

Timer tasks should complete quickly. When a timer task takes too long to complete, it "hogs" the timer's task-execution thread, delaying the execution of subsequent timer tasks, which may "bunch up" and execute in rapid succession if and when the offending timer task finally completes.

> **NOTE:** `Timer` scales to large numbers of concurrently scheduled timer tasks (thousands should present no problem). Internally, it uses a *binary heap* to represent its timer task queue so the cost to schedule a timer task is *O(log n)*, where *n* is the number of concurrently scheduled timer tasks. Check out Wikipedia's " Big O notation" topic (`http://en.wikipedia.org/wiki/Big_O_notation`) to learn more about *O()*.

`Timer` declares the following constructors:

- `Timer()` creates a new timer whose task-execution thread doesn't run as a daemon thread.

- `Timer(boolean isDaemon)` creates a new timer whose task-execution thread may be specified to run as a daemon (pass `true` to `isDaemon`). A daemon thread is called for scenarios where the timer will be used to schedule repeating “maintenance activities”, which must be performed for as long as the application is running, but shouldn’t prolong the application’s lifetime.

- `Timer(String name)` creates a new timer whose task-execution thread has the specified `name`. The task-execution thread doesn’t run as a daemon thread. This constructor throws `NullPointerException` when `name` is `null`.

- `Timer(String name, boolean isDaemon)` creates a new timer whose task-execution thread has the specified `name` and which may run as a daemon thread. This constructor throws `NullPointerException` when `name` is `null`.

`Timer` also declares the following methods:

- `void cancel()` terminates this timer, discarding any currently scheduled timer tasks. This method doesn’t interfere with a currently executing timer task (when it exists). After a timer has been terminated, its execution thread terminates gracefully and no more timer tasks may be scheduled on it. (Calling `cancel()` from within the `run()` method of a timer task that was invoked by this timer absolutely guarantees that the ongoing task execution is the last task execution that will ever be performed by this timer.) This method may be called repeatedly; the second and subsequent calls have no effect.

- `int purge()` removes all canceled timer tasks from this timer’s queue, and returns the number of timer tasks that have been removed. Calling `purge()` has no effect on the behavior of the timer, but eliminates references to the canceled timer tasks from the queue. When there are no external references to these timer tasks, they become eligible for garbage collection. (Most applications won’t need to call this method, which is designed for use by the rare application that cancels a large number of timer tasks. Calling `purge()` trades time for space: this method’s runtime may be proportional to $n + c * \log n$, where $n$ is the number of timer tasks in the queue and $c$ is the number of canceled timer tasks.) It’s permissible to call `purge()` from within a timer task scheduled on this timer.

- `void schedule(TimerTask task, Date time)` schedules `task` for execution at `time`. When `time` is in the past, `task` is scheduled for immediate execution. This method throws `IllegalArgumentException` when `time.getTime()` is negative; `java.lang.IllegalStateException` when `task` was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and `NullPointerException` when `task` or `time` is `null`.

- `void schedule(TimerTask task, Date firstTime, long period)` schedules `task` for repeated fixed-delay execution, beginning at `firstTime`. Subsequent executions take place at approximately regular intervals, separated by `period` milliseconds. In *fixed-delay execution*, each execution is scheduled relative to the actual execution time of the previous execution. When an execution is delayed for any reason (e.g., garbage collection), subsequent executions are also delayed. In the long run, the frequency of execution will generally be slightly lower than the reciprocal of `period` (assuming the system clock underlying `Object.wait(long)` is accurate). As a consequence, when the scheduled `firstTime` value is in the past, `task` is scheduled for immediate execution. Fixed-delay execution is appropriate for recurring tasks that require "smoothness." In other words, this form of execution is appropriate for tasks where it's more important to keep the frequency accurate in the short run than in the long run. This includes most animation tasks, such as blinking a cursor at regular intervals. It also includes tasks wherein regular activity is performed in response to human input, such as automatically repeating a character for as long as a key is held down. This method throws `IllegalArgumentException` when `firstTime.getTime()` is negative, or `period` is negative or zero; `IllegalStateException` when `task` was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and `NullPointerException` when `task` or `firstTime` is `null`.

- `void schedule(TimerTask task, long delay)` schedules `task` for execution after `delay` milliseconds. This method throws `IllegalArgumentException` when `delay` is negative or `delay + System.currentTimeMillis()` is negative; `IllegalStateException` when `task` was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and `NullPointerException` when `task` is `null`.

- `void schedule(TimerTask task, long delay, long period)` schedules `task` for repeated fixed-delay execution, beginning after `delay` milliseconds. Subsequent executions take place at approximately regular intervals separated by `period` milliseconds. This method throws `IllegalArgumentException` when `delay` is negative, `delay + System.currentTimeMillis()` is negative, or `period` is negative or zero; `IllegalStateException` when `task` was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and `NullPointerException` when `task` is `null`.

- `void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)` schedules `task` for repeated fixed-rate execution, beginning at `time`. Subsequent executions take place at approximately regular intervals, separated by `period` milliseconds. In *fixed-rate execution*, each execution is scheduled relative to the scheduled execution time of the initial execution. When an execution is delayed for any reason (e.g., garbage collection), two or more executions will occur in rapid succession to “catch up.” In the long run, the frequency of execution will be exactly the reciprocal of `period` (assuming the system clock underlying `Object.wait(long)` is accurate). As a consequence, when the scheduled `firstTime` is in the past, any “missed” executions will be scheduled for immediate “catch up” execution. Fixed-rate execution is appropriate for recurring activities that are sensitive to absolute time (e.g., ringing a chime every hour on the hour, or running scheduled maintenance every day at a particular time). It’s also appropriate for recurring activities where the total time to perform a fixed number of executions is important, such as a countdown timer that ticks once every second for ten seconds. Finally, fixed-rate execution is appropriate for scheduling multiple repeating timer tasks that must remain synchronized with respect to one another. This method throws `IllegalArgumentException` when `firstTime.getTime()` is negative, or `period` is negative or zero; `IllegalStateException` when `task` was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and `NullPointerException` when `task` or `firstTime` is `null`.

- `void scheduleAtFixedRate(TimerTask task, long delay, long period)` schedules `task` for repeated fixed-rate execution, beginning after `delay` milliseconds. Subsequent executions take place at approximately regular intervals, separated by `period` milliseconds. This method throws `IllegalArgumentException` when `delay` is negative, `delay + System.currentTimeMillis()` is negative, or `period` is negative or zero; `IllegalStateException` when `task` was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and `NullPointerException` when `task` is `null`.

Timer tasks are instances of classes that subclass the abstract `TimerTask` class, which implements the `Runnable` interface and offers the following methods:

- `boolean cancel()` cancels this timer task. When the timer task has been scheduled for one-shot execution and hasn't yet run, or when it hasn't yet been scheduled, it will never run. When the timer task has been scheduled for repeated execution, it will never run again. (When the timer task is running when this call occurs, the timer task will run to completion, but will never run again.) Calling `cancel()` from within the `run()` method of a repeating timer task absolutely guarantees that the timer task won't run again. This method may be called repeatedly; the second and subsequent calls have no effect. This method returns true when this timer task is scheduled for one-shot execution and hasn't yet run, or when this timer task is scheduled for repeated execution. It returns false when the timer task was scheduled for one-shot execution and has already run, or when the timer task was never scheduled, or when the timer task was already canceled. (Loosely speaking, this method returns true when it prevents one or more scheduled executions from taking place.)

- `void run()` provides the timer task's code. You must override this abstract method to perform a useful activity.

- `long scheduledExecutionTime()` returns the scheduled execution time of the most recent actual execution of this timer task. (When this method is invoked while timer task execution is in progress, the return value is the scheduled execution time of the ongoing timer task execution.) This method is typically invoked from within a task's `run()` method to determine whether the current execution of the timer task is sufficiently timely to warrant performing the scheduled activity. For example, you would specify code similar to `if (System.currentTimeMillis() - scheduledExecutionTime() >= MAX_TARDINESS) return;` at the start of the `run()` method to abort the current timer task execution when it's not timely. This method is typically not used in conjunction with fixed-delay execution repeating timer tasks because their scheduled execution times are allowed to drift over time, and are thus not terribly significant. `scheduledExecutionTime()` returns the time at which the most recent execution of this timer task was scheduled to occur, in the format returned by `Date.getTime()`. The return value is undefined when the timer task has yet to commence its first execution.

I've created a simple `TimerDemo` application that provides a basic demonstration of `Timer` and `TimerTask`. Listing C–51 presents `TimerDemo`'s source code.

*Listing C–51. Demonstrating one-shot and repeated timer tasks*

```
import java.util.Timer;
import java.util.TimerTask;

public class TimerDemo
{
   public static void main(String[] args)
   {
      Timer t = new Timer(true);
      t.schedule(new TimerTask()
                 {
                    @Override
                    public void run()
                    {
                       System.out.println("one-shot timer task executing");
                    }
                 }, 2000); // Execute one-shot timer task after 2-second delay.
      System.out.println("main thread is sleeping for 4 seconds");
      try { Thread.sleep(4000); } catch (InterruptedException ie) {}
      System.out.println("main thread has woken up");
      t = new Timer();
      t.schedule(new TimerTask()
                 {
                    int i; // initializes to 0 by default
                    @Override
                    public void run()
                    {
                       System.out.println("repeated timer task is running");
                       if (++i == 6)
                       {
                          System.out.println("canceling repeated timer task");
                          cancel();
                          System.out.println("canceled");
                          System.exit(0);
                       }
                    }
                 }, 2000, 500);
      System.out.println("main thread is terminating");
   }
}
```

Listing C–51's main thread first creates a timer whose task-execution thread is daemon, and then schedules a one-shot timer task to execute two seconds later. The main thread then sleeps for four seconds to give you an opportunity to observe the timer task's execution via an output message.

After the main thread awakens, it creates a timer whose task-execution thread isn't daemon, and then schedules a repeating timer task to run two seconds later and repeat every half-second. The main thread then terminates.

Because the timer's task-execution thread isn't daemon, the application doesn't terminate. Instead, the timer repeatedly executes its timer task six times before the final timer task execution cancels itself and invokes `System.exit(0);` to terminate the application. Otherwise, the application would never end because the timer's nondaemon task-execution thread is still running and waiting to execute subsequently scheduled timer tasks.

Compile Listing C–51 (`javac TimerDemo.java`) and run this application (`java TimerDemo`). You should observe the following output:

```
main thread is sleeping for 4 seconds
one-shot timer task executing
main thread has woken up
main thread is terminating
repeated timer task is running
repeated timer task is running
repeated timer task is running
repeated timer task is running
repeated timer task is running
repeated timer task is running
canceling repeated timer task
canceled
```

After the last live reference to a `Timer` object goes away and all outstanding timer tasks have completed execution, the timer's task-execution thread terminates gracefully (and becomes subject to garbage collection). However, this can take arbitrarily long to occur. (By default, the task-execution thread doesn't run as a daemon thread, so it's capable of preventing an application from terminating.) When an application wants to terminate a timer's task-execution thread rapidly, the application should invoke `Timer`'s `cancel()` method.

When the timer's task-execution thread terminates unexpectedly, for example, because its `stop()` method was invoked (you should never call any of `Thread`'s `stop()` methods because they're inherently unsafe), any further attempt to schedule a timer task on the timer results in `IllegalStateException`, as if `Timer`'s `cancel()` method had been invoked.

■ **NOTE:** One of the `java.util.concurrent` package's types is `ScheduledThreadPoolExecutor`, which is a thread pool executor for repeatedly executing tasks at a given rate or delay. It's a more versatile replacement for the `Timer`/`TimerTask` combination, as it allows multiple service threads, accepts various time units, and doesn't require you to subclass `TimerTask` (just implement `Runnable`). Configuring `ScheduledThreadPoolExecutor` with one thread makes it equivalent to `Timer`. However, you might find `Timer` and `TimerTask` to be conceptually simpler to work with.

## Final Exercises

Each of Chapters 1 through 14 ends with a set of exercises, and this section presents a few more. Solutions are included.

## Classify

Class declares boolean isAnnotation(), boolean isEnum(), and boolean isInterface() methods that return true when the Class object represents an annotation type, an enum, or an interface, respectively. Create a Classify application that uses Class's forName() method to load its single command-line argument, which will represent an annotation type, enum, interface, or class (the default). Use a chained if-else statement along with the aforementioned methods to output Annotation, Enum, Interface, or Class.

### Solution

Listing C–52 reveals the Classify application.

*Listing C–52. Classifying a type as an annotation type, an enum, an interface, or a class*

```
public class Classify
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java Classify pkgAndTypeName");
         return;
      }
      try
      {
         Class<?> clazz = Class.forName(args[0]);
         if (clazz.isAnnotation())
            System.out.println("Annotation");
         else
         if (clazz.isEnum())
            System.out.println("Enum");
         else
         if (clazz.isInterface())
            System.out.println("Interface");
         else
            System.out.println("Class");
      }
      catch (ClassNotFoundException cnfe)
      {
         System.err.println("could not locate " + args[0]);
      }
   }
}
```

Compile Listing C–52 (`javac Classify.java`) and run this application, as follows:

```
java Classify java.lang.Byte
```

You should observe the following output:

```
Class
```

## Media

Suppose you're creating a `Media` class whose `static` methods perform various media-oriented utility tasks; for example, a `getID3Info()` method returns an object containing information about an MP3 file (e.g., song title and artist). This information is typically stored in a 128-byte block at the end of the file according to a format known as ID3. The block begins with ASCII sequence `TAG`.

Listing C–53 reveals the `Media` class with `getID3Info()`'s skeletal contents, and begins with a description of the ID3v1.1 format that this method targets (there are several versions of this format).

*Listing C–53. The `Media` class and its skeletal `getID3Info()` method*

```
/*

In 1996, Eric Kemp devised ID3, a small file format for storing metadata in
MP3 files. This metadata consisted of song title, artist, album, year,
comments, and genre; and was organized into a 128-byte block stored at the
end of the file.

The following table describes the format of ID3 version 1 (ID3v1):

Offset (decimal) Field Name      Field Size (byte)
================ ==========      =================
0                Signature (TAG) 3
3                Song title      30
33               Artist          30
63               Album           30
93               Year            4
97               Comment         30
127              Genre           1

Each field except for Genre is a string of ASCII characters. Strings are
padded on the right with zeros or spaces. An uninitialized field is
equivalent to the empty string ("").

In 1997, Michael Mutschler made a small improvement to ID3. Because the
Comment field is too small to write anything of use, he trimmed this field by
```

two bytes and used those bytes to store the CD track number where the song is located.

If a track number is stored, the second-last byte of the Comment field is set to 0 and the subsequent byte stores the track number. The resulting format is known as ID3v1.1.

| Offset (decimal) | Field Name | Field Size (byte) |
|---|---|---|
| 0 | Signature (TAG) | 3 |
| 3 | Song title | 30 |
| 33 | Artist | 30 |
| 63 | Album | 30 |
| 93 | Year | 4 |
| 97 | Comment | 29 (last byte must be a binary 0) |
| 126 | Track (0-255) | 1 |
| 127 | Genre (0-255) | 1 (255 means no genre) |

Unlike most of the fields, Track and Genre are single-byte integer fields.
The legal values that can appear in the Genre field and their descriptions
Are described in the following table:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | Blues | 20 | Alternative | 40 | AlternRock | 60 | Top 40 |
| 1 | Classic Rock | 21 | Ska | 41 | Bass | 61 | Christian Rap |
| 2 | Country | 22 | Death Metal | 42 | Soul | 62 | Pop/Funk |
| 3 | Dance | 23 | Pranks | 43 | Punk | 63 | Jungle |
| 4 | Disco | 24 | Soundtrack | 44 | Space | 64 | Nat American |
| 5 | Funk | 25 | Euro-Techno | 45 | Meditative | 65 | Cabaret |
| 6 | Grunge | 26 | Ambient | 46 | Instrumental Pop | 66 | New Wave |
| 7 | Hip-Hop | 27 | Trip-Hop | 47 | Instrumental Rock | 67 | Psychadelic |
| 8 | Jazz | 28 | Vocal | 48 | Ethnic | 68 | Rave |
| 9 | Metal | 29 | Jazz+Funk | 49 | Gothic | 69 | Showtunes |
| 10 | New Age | 30 | Fusion | 50 | Darkwave | 70 | Trailer |
| 11 | Oldies | 31 | Trance | 51 | Techno-Industrial | 71 | Lo-Fi |
| 12 | Other | 32 | Classical | 52 | Electronic | 72 | Tribal |
| 13 | Pop | 33 | Instrumental | 53 | Pop-Folk | 73 | Acid Punk |
| 14 | R&B | 34 | Acid | 54 | Eurodance | 74 | Acid Jazz |
| 15 | Rap | 35 | House | 55 | Dream | 75 | Polka |
| 16 | Reggae | 36 | Game | 56 | Southern Rock | 76 | Retro |
| 17 | Rock | 37 | Sound Clip | 57 | Comedy | 77 | Musical |
| 18 | Techno | 38 | Gospel | 58 | Cult | 78 | Rock & Roll |
| 19 | Industrial | 39 | Noise | 59 | Gangsta | 79 | Hard Rock |

WinAmp expanded this table with the following Genre codes:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 80 | Folk | 92 | Progressive Rock | 104 | Chamber Music | 116 | Ballad |
| 81 | Folk-Rock | 93 | Psychedelic Rock | 105 | Sonata | 117 | Power Ballad |
| 82 | National-Folk | 94 | Symphonic Rock | 106 | Symphony | 118 | Rhythmic Soul |
| 83 | Swing | 95 | Slow Rock | 107 | Booty Brass | 119 | Freestyle |
| 84 | Fast Fusion | 96 | Big Band | 108 | Primus | 120 | Duet |
| 85 | Bebob | 97 | Chorus | 109 | Porn Groove | 121 | Punk Rock |
| 86 | Latin | 98 | Easy Listening | 110 | Satire | 122 | Drum Solo |
| 87 | Revival | 99 | Acoustic | 111 | Slow Jam | 123 | A cappella |
| 88 | Celtic | 100 | Humour | 112 | Club | 124 | Euro-House |

```
89 Bluegrass     101 Speech           113 Tango             125 Dance Hall
90 Avantgarde    102 Chanson          114 Samba
91 Gothic Rock   103 Opera            115 Folklore

Although there are probably additional defined codes, treat any other value
stored in the Genre field as Unknown.

To learn more about ID3 and new versions, visit the official site at id3.org.

*/

import java.io.IOException;
import java.io.RandomAccessFile;

public class Media
{
   public static class ID3
   {
      private String songTitle, artist, album, year, comment, genre;
      private int track; // -1 if track not present
      public ID3(String songTitle, String artist, String album, String year,
                 String comment, int track, String genre)
      {
         this.songTitle = songTitle;
         this.artist = artist;
         this.album = album;
         this.year = year;
         this.comment = comment;
         this.track = track;
         this.genre = genre;
      }
      String getSongTitle() { return songTitle; }
      String getArtist() { return artist; }
      String getAlbum() { return album; }
      String getYear() { return year; }
      String getComment() { return comment; }
      int getTrack() { return track; }
      String getGenre() { return genre; }
   }

   public static ID3 getID3Info(String mp3path) throws IOException
   {
      return null;
   }
}
```

Your job is to fill in the `getID3Info()` method by using the `RandomAccessFile` class to obtain the data from the 128-byte block, and then create and populate an `ID3` object with this data. `getID3Info()` subsequently returns this object.

After completing `getID3Info()`, you will want to test this method. Listing C–54 presents the source code to a `TestMedia` application that you can use for this purpose.

*Listing C–54. The `TestMedia` class for testing `getID3Info()`*

```
import java.io.IOException;

public class TestMedia
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java TestMedia mp3path");
         return;
      }
      try
      {
         Media.ID3 id3Info = Media.getID3Info(args[0]);
         if (id3Info == null)
         {
            System.err.printf("%s not MP3 or has no ID3 block%n", args[0]);
            return;
         }
         System.out.println("Song title = " + id3Info.getSongTitle());
         System.out.println("Artist = " + id3Info.getArtist());
         System.out.println("Album = " + id3Info.getAlbum());
         System.out.println("Year = " + id3Info.getYear());
         System.out.println("Comment = " +id3Info.getComment());
         System.out.println("Track = " + id3Info.getTrack());
         System.out.println("Genre = " + id3Info.getGenre());
      }
      catch (IOException ioe)
      {
         ioe.printStackTrace();
      }
   }
}
```

Suppose you've ripped a track from a CD that contains Tom Jones' "She's a Lady" into a file called `Lady.mp3`. When you execute `java TestMedia Lady.mp3`, you should see output similar to the following:

```
Song title = She's A Lady
Artist = Tom Jones
Album = Billboard Top Soft Rock Hits -
Year = 1998
Comment =
Track = 2
Genre = Rock
```

## Solution

Listing C–55 presents `Media` with an expanded `getID3Info()` method.

*Listing C–55. The `Media` class and its completed `getID3Info()` method*

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class Media
{
   public static class ID3
   {
      private String songTitle, artist, album, year, comment, genre;
      private int track; // -1 if track not present
      public ID3(String songTitle, String artist, String album, String year,
                 String comment, int track, String genre)
      {
         this.songTitle = songTitle;
         this.artist = artist;
         this.album = album;
         this.year = year;
         this.comment = comment;
         this.track = track;
         this.genre = genre;
      }
      String getSongTitle() { return songTitle; }
      String getArtist() { return artist; }
      String getAlbum() { return album; }
      String getYear() { return year; }
      String getComment() { return comment; }
      int getTrack() { return track; }
      String getGenre() { return genre; }
   }

   public static ID3 getID3Info(String mp3path) throws IOException
   {
      RandomAccessFile raf = null;
      try
      {
         raf = new RandomAccessFile(mp3path, "r");
         if (raf.length() < 128)
            return null; // Not MP3 file (way too small)
         raf.seek(raf.length() - 128);
         byte[] buffer = new byte[128];
         raf.read(buffer);
         raf.close();
         if (buffer[0] != (byte) 'T' && buffer[1] != (byte) 'A' &&
             buffer[2] != (byte) 'G')
            return null; // No ID3 block (must start with TAG)
         String songTitle = new String(buffer, 3, 30);
         String artist = new String(buffer, 33, 30);
         String album = new String(buffer, 63, 30);
         String year = new String(buffer, 93, 4);
         String comment = new String(buffer, 97, 28);
         // buffer[126] & 255 converts -128 through 127 to 0 through 255
         int track = (buffer[125] == 0) ? buffer[126] & 255 : -1;
         String[] genres = new String[]
```

```
{
   "Blues",
   "Classic Rock",
   "Country",
   "Dance",
   "Disco",
   "Funk",
   "Grunge",
   "Hip-Hop",
   "Jazz",
   "Metal",
   "New Age",
   "Oldies",
   "Other",
   "Pop",
   "R&B",
   "Rap",
   "Reggae",
   "Rock",
   "Techno",
   "Industrial",
   "Alternative",
   "Ska",
   "Death Metal",
   "Pranks",
   "Soundtrack",
   "Euro-Techno",
   "Ambient",
   "Trip-Hop",
   "Vocal",
   "Jazz+Funk",
   "Fusion",
   "Trance",
   "Classical",
   "Instrumental",
   "Acid",
   "House",
   "Game",
   "Sound Clip",
   "Gospel",
   "Noise",
   "AlternRock",
   "Bass",
   "Soul",
   "Punk",
   "Space",
   "Meditative",
   "Instrumental Pop",
   "Instrumental Rock",
   "Ethnic",
   "Gothic",
   "Darkwave",
   "Techno-Industrial",
   "Electronic",
                                    "Pop-Folk",
                                    "Eurodance",
                                    "Dream",
                                    "Southern Rock",
                                    "Comedy",
                                    "Cult",
                                    "Gangsta",
                                    "Top 40",
                                    "Christian Rap",
                                    "Pop/Funk",
                                    "Jungle",
                                    "Native American",
                                    "Cabaret",
                                    "New Wave",
                                    "Psychedelic",
                                    "Rave",
                                    "Showtunes",
                                    "Trailer",
                                    "Lo-Fi",
                                    "Tribal",
                                    "Acid Punk",
                                    "Acid Jazz",
                                    "Polka",
                                    "Retro",
                                    "Musical",
                                    "Rock & Roll",
                                    "Hard Rock",
                                    "Folk",
                                    "Folk-Rock",
                                    "National-Folk",
                                    "Swing",
                                    "Fast Fusion",
                                    "Bebob",
                                    "Latin",
                                    "Revival",
                                    "Celtic",
                                    "Bluegrass",
                                    "Avantegarde",
                                    "Gothic Rock",
                                    "Progressive Rock",
                                    "Psychedelic Rock",
                                    "Symphonic Rock",
                                    "Slow Rock",
                                    "Big Band",
                                    "Chorus",
                                    "Easy Listening",
                                    "Acoustic",
                                    "Humour",
                                    "Speech",
                                    "Chanson",
                                    "Opera",
                                    "Chamber Music",
                                    "Sonata",
                                    "Symphony",
                                    "Booty Brass",
                                    "Primus",
                                    "Porn Groove",
                                    "Satire",
                                    "Slow Jam",
                                    "Club",
                                    "Tango",
                                    "Samba",
                                    "Folklore",
                                    "Ballad",
                                    "Power Ballad",
                                    "Rhythmic Soul",
                                    "Freestyle",
                                    "Duet",
                                    "Punk Rock",
                                    "Drum Solo",
                                    "A cappella",
                                    "Euro-House",
                                    "Dance Hall"
                                 };
               assert genres.length == 126;
               String genre = (buffer[127] < 0 || buffer[127] > 125)
                              ? "Unknown" : genres[buffer[127]];
               return new ID3(songTitle, artist, album, year, comment, track, genre);
            }
            catch (IOException ioe)
            {
               if (raf != null)
                  try
                  {
                     raf.close();
                  }
                  catch (IOException ioe2)
                  {
                     ioe2.printStackTrace();
                  }
               throw ioe;
            }
         }
      }
```

## SpanishCollation

Create a `SpanishCollation` application that outputs Spanish words ñango (weak), llamado (called), lunes (Monday), champán (champagne), clamor (outcry), cerca (near), nombre (name), and chiste (joke) according to this language's current collation rules followed by its traditional collation rules. According to the current collation rules, the output order is as follows: cerca, champán, chiste, clamor, llamado, lunes, nombre, and ñango. According to the traditional collation rules, the output order is as follows: cerca, clamor, champán, chiste, lunes, llamado, nombre, and ñango. Use the `RuleBasedCollator` class to specify the rules for traditional collation. Also, construct your `Locale` object using only the es (Spanish) language code.

■ **NOTE:** The Spanish alphabet consists of 29 letters: a, b, c, ch, d, e, f, g, h, i, j, k, l, ll, m, n, ñ, o, p, q, r, s, t, u, v, w, x, y, z. (Vowels are often written with accents, as in tablón [plank or board], and u is sometimes topped with a dieresis or umlaut, as in vergüenza [bashfulness]. However, vowels with these diacritical marks are not considered separate letters.) Prior to April 1994's voting at the X Congress of the Association of Spanish Language Academies, ch was collated after c, and ll was collated after l. Because this congress adopted the standard Latin alphabet collation rules, ch is now considered a sequence of two distinct characters, and dictionaries now place words starting with ch between words starting with cg and ci. Similarly, ll is now considered a sequence of two characters.

## Solution

Listing C–56 presents `SpanishCollation`.

*Listing C–56. Outputting Spanish words according to this language's current collation rules followed by its traditional collation rules*

```
import java.text.Collator;
import java.text.ParseException;
import java.text.RuleBasedCollator;

import java.util.Arrays;
import java.util.Locale;

public class SpanishCollation
{
   public static void main(String[] args)
   {
      String[] words =
      {
         "ñango",   // weak
         "llamado", // called
         "lunes",   // monday
         "champán", // champagne
         "clamor",  // outcry
         "cerca",   // near
         "nombre",  // name
         "chiste",  // joke
      };
      Locale locale = new Locale("es", "");
      Collator c = Collator.getInstance(locale);
      Arrays.sort(words, c);
      for (String word: words)
         System.out.println(word);
      System.out.println();
      // Define the traditional Spanish sort rules.
      String upperNTilde = new String ("\u00D1");
      String lowerNTilde = new String ("\u00F1");
      String spanishRules = "< a,A < b,B < c,C < ch, cH, Ch, CH < d,D < e,E " +
                            "< f,F < g,G < h,H < i,I < j,J < k,K < l,L < ll, " +
                            "lL, Ll, LL < m,M < n,N < " + lowerNTilde + "," +
                            upperNTilde + " < o,O < p,P < q,Q < r,R < s,S < " +
                            "t,T < u,U < v,V < w,W < x,X < y,Y < z,Z";
      try
      {
         c = new RuleBasedCollator(spanishRules);
         Arrays.sort(words, c);
         for (String word: words)
            System.out.println(word);
      }
      catch (ParseException pe)
      {
         System.err.println(pe);
      }
   }
}
```

Compile Listing C–56 (`javac SpanishCollation.java`) and run this application (`java SpanishCollation`). You should observe the following output:

```
cerca
champán
chiste
clamor
llamado
lunes
nombre
ñango

cerca
clamor
champán
chiste
lunes
llamado
```

```
nombre
ñango
```

■ **NOTE:** I hope you've enjoyed this appendix along with the rest of this book. If you have any questions about the content presented in *Learn Java for Android Development Second Edition*, feel free to contact me (Jeff Friesen). Send an email to `jeff@tutortutor.ca`.