I/O

Creating a good input/output (I/O) system is one of the more difficult tasks for a language designer. This is evidenced by the number of different approaches.

The challenge seems to be in covering all possibilities. Not only are there different sources and sinks of I/O that you want to communicate with (files, the console, network connections, etc.), but you need to talk to them in a wide variety of ways (sequential, random-access, buffered, binary, character, by lines, by words, etc.).

The Java library designers attacked this problem by creating lots of classes. In fact, there are so many classes for Java's I/O system that it can be intimidating at first (ironically, the Java I/O design actually prevents an explosion of classes). There was also a significant change in the I/O library after Java 1.0, when the original byte-oriented library was supplemented with **char**-oriented, Unicode-based I/O classes. The **nio** classes (for "new I/O," a name we'll still be using years from now even though they were introduced in JDK 1.4 and so are already "old") were added for improved performance and functionality. As a result, there are a fair number of classes to learn before you understand enough of Java's I/O picture that you can use it properly. In addition, it's rather important to understand the evolution of the I/O library, even if your first reaction is "Don't bother me with history, just show me how to use it!" The problem is that without the historical perspective, you will rapidly become confused with some of the classes and when you should and shouldn't use them.

This chapter will give you an introduction to the variety of I/O classes in the standard Java library and how to use them.

The **File** class

Before getting into the classes that actually read and write data to streams, we'll look at a library utility that assists you with file directory issues.

The **File** class has a deceiving name; you might think it refers to a file, but it doesn't. In fact, "FilePath" would have been a better name for the class. It can

represent either the *name* of a particular file or the *names* of a set of files in a directory. If it's a set of files, you can ask for that set using the **list( )** method, which returns an array of **String**. It makes sense to return an array rather than one of the flexible container classes, because the number of elements is fixed, and if you want a different directory listing, you just create a different **File** object. This section shows an example of the use of this class, including the associated **FilenameFilter** interface.

A directory lister

Suppose you'd like to see a directory listing. The **File** object can be used in two ways. If you call **list( )** with no arguments, you'll get the full list that the **File** object contains. However, if you want a restricted list—for example, if you want all of the files with an extension of **.java**—then you use a "directory filter," which is a class that tells how to select the **File** objects for display.

Here's the example. Note that the result has been effortlessly sorted (alphabetically) using the **java.util.Arrays.sort( )** method and the **String.CASE_INSENSITIVE_ORDER Comparator**:

```
//: io/DirList.java
// Display a directory listing using regular expressions.
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList {
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(new DirFilter(args[0]));
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
}

class DirFilter implements FilenameFilter {
  private Pattern pattern;
```

```
  public DirFilter(String regex) {
    pattern = Pattern.compile(regex);
  }
  public boolean accept(File dir, String name) {
    return pattern.matcher(name).matches();
  }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

The **DirFilter** class implements the interface **FilenameFilter**. Notice how simple the **FilenameFilter** interface is:

```
public interface FilenameFilter {
  boolean accept(File dir, String name);
}
```

DirFilter's sole reason for existence is to provide the **accept( )** method to the **list( )** method so that **list( )** can "call back" **accept( )** to determine which file names should be included in the list. Thus, this structure is often referred to as a *callback*. More specifically, this is an example of the *Strategy* design pattern, because **list( )** implements basic functionality, and you provide the Strategy in the form of a **FilenameFilter** in order to complete the algorithm necessary for **list( )** to provide its service. Because **list( )** takes a **FilenameFilter** object as its argument, it means that you can pass an object of any class that implements **FilenameFilter** to choose (even at run time) how the **list( )** method will behave. The purpose of a Strategy is to provide flexibility in the behavior of code.

The **accept( )** method must accept a **File** object representing the directory that a particular file is found in, and a **String** containing the name of that file. Remember that the **list( )** method is calling **accept( )** for each of the file names in the directory object to see which one should be included; this is indicated by the **boolean** result returned by **accept( )**.

accept( ) uses a regular expression **matcher** object to see if the regular expression **regex** matches the name of the file. Using **accept( )**, the **list( )** method returns an array.

Anonymous inner classes

This example is ideal for rewriting using an anonymous inner class (described in *Inner Classes*). As a first cut, a method **filter( )** is created that returns a reference to a **FilenameFilter**:

```
//: io/DirList2.java
// Uses anonymous inner classes.
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList2 {
  public static FilenameFilter filter(final String regex) {
    // Creation of anonymous inner class:
    return new FilenameFilter() {
      private Pattern pattern = Pattern.compile(regex);
      public boolean accept(File dir, String name) {
        return pattern.matcher(name).matches();
      }
    }; // End of anonymous inner class
  }
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(filter(args[0]));
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

Note that the argument to **filter( )** must be **final**. This is required by the anonymous inner class so that it can use an object from outside its scope.

This design is an improvement because the **FilenameFilter** class is now tightly bound to **DirList2**. However, you can take this approach one step further and define the anonymous inner class as an argument to **list( )**, in which case it's even smaller:

```
//: io/DirList3.java
// Building the anonymous inner class "in-place."
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList3 {
  public static void main(final String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(new FilenameFilter() {
        private Pattern pattern = Pattern.compile(args[0]);
        public boolean accept(File dir, String name) {
          return pattern.matcher(name).matches();
        }
      });
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

The argument to **main( )** is now **final**, since the anonymous inner class uses **args[0]** directly.

This shows you how anonymous inner classes allow the creation of specific, one-off classes to solve problems. One benefit of this approach is that it keeps the code that solves a particular problem isolated in one spot. On the other hand, it is not always as easy to read, so you must use it judiciously.

Exercise 1: (3) Modify **DirList.java** (or one of its variants) so that the **FilenameFilter** opens and reads each file (using the **net.mindview.util.TextFile** utility) and accepts the file based on whether any of the trailing arguments on the command line exist in that file.

Exercise 2: (2) Create a class called **SortedDirList** with a constructor that takes a **File** object and builds a sorted directory list from the files at that **File**. Add to this class two overloaded **list( )** methods: the first produces the whole list, and the second produces the subset of the list that matches its argument (which is a regular expression).

Exercise 3: (3) Modify **DirList.java** (or one of its variants) so that it sums up the file sizes of the selected files.

Directory utilities

A common task in programming is to perform operations on sets of files, either in the local directory or by walking the entire directory tree. It is useful to have a tool that will produce the set of files for you. The following utility class produces either an array of **File** objects in the local directory using the **local( )** method, or a **List<File>** of the entire directory tree starting at the given directory using **walk( )** (**File** objects are more useful than file names because **File** objects contain more information). The files are chosen based on the regular expression that you provide:

```
//: net/mindview/util/Directory.java
// Produce a sequence of File objects that match a
// regular expression in either a local directory,
// or by walking a directory tree.
package net.mindview.util;
import java.util.regex.*;
import java.io.*;
import java.util.*;

public final class Directory {
  public static File[]
  local(File dir, final String regex) {
    return dir.listFiles(new FilenameFilter() {
      private Pattern pattern = Pattern.compile(regex);
      public boolean accept(File dir, String name) {
        return pattern.matcher(
          new File(name).getName()).matches();
      }
    });
```

```
  }
  public static File[]
  local(String path, final String regex) { // Overloaded
    return local(new File(path), regex);
  }
  // A two-tuple for returning a pair of objects:
  public static class TreeInfo implements Iterable<File> {
    public List<File> files = new ArrayList<File>();
    public List<File> dirs = new ArrayList<File>();
    // The default iterable element is the file list:
    public Iterator<File> iterator() {
      return files.iterator();
    }
    void addAll(TreeInfo other) {
      files.addAll(other.files);
      dirs.addAll(other.dirs);
    }
    public String toString() {
      return "dirs: " + PPrint.pformat(dirs) +
        "\n\nfiles: " + PPrint.pformat(files);
    }
  }
  public static TreeInfo
  walk(String start, String regex) { // Begin recursion
    return recurseDirs(new File(start), regex);
  }
  public static TreeInfo
  walk(File start, String regex) { // Overloaded
    return recurseDirs(start, regex);
  }
  public static TreeInfo walk(File start) { // Everything
    return recurseDirs(start, ".*");
  }
  public static TreeInfo walk(String start) {
    return recurseDirs(new File(start), ".*");
  }
  static TreeInfo recurseDirs(File startDir, String regex){
    TreeInfo result = new TreeInfo();
    for(File item : startDir.listFiles()) {
      if(item.isDirectory()) {
        result.dirs.add(item);
        result.addAll(recurseDirs(item, regex));
      } else // Regular file
        if(item.getName().matches(regex))
```

```
            result.files.add(item);
      }
      return result;
    }
    // Simple validation test:
    public static void main(String[] args) {
      if(args.length == 0)
        System.out.println(walk("."));
      else
        for(String arg : args)
          System.out.println(walk(arg));
    }
} ///:~
```

The **local( )** method uses a variant of **File.list( )** called **listFiles( )** that produces an array of **File**. You can see that it also uses a **FilenameFilter**. If you need a **List** instead of an array, you can convert the result yourself using **Arrays.asList( )**.

The **walk( )** method converts the name of the starting directory into a **File** object and calls **recurseDirs( )**, which performs a recursive directory walk, collecting more information with each recursion. To distinguish ordinary files from directories, the return value is effectively a "tuple" of objects—a **List** holding ordinary files, and another holding directories. The fields are intentionally made **public** here, because the point of **TreeInfo** is simply to collect the objects together—if you were just returning a **List**, you wouldn't make it **private**, so just because you are returning a pair of objects, it doesn't mean you need to make them **private**. Note that **TreeInfo** implements **Iterable<File>**, which produces the files, so that you have a "default iteration" over the file list, whereas you can specify directories by saying "**.dirs**".

The **TreeInfo.toString( )** method uses a "pretty printer" class so that the output is easer to view. The default **toString( )** methods for containers print all the elements for a container on a single line. For large collections this can become difficult to read, so you may want to use an alternate formatting. Here's a tool that adds newlines and indents each element:

```
//: net/mindview/util/PPrint.java
// Pretty-printer for collections
package net.mindview.util;
import java.util.*;
```

```
public class PPrint {
  public static String pformat(Collection<?> c) {
    if(c.size() == 0) return "[]";
    StringBuilder result = new StringBuilder("[");
    for(Object elem : c) {
      if(c.size() != 1)
        result.append("\n  ");
      result.append(elem);
    }
    if(c.size() != 1)
      result.append("\n");
    result.append("]");
    return result.toString();
  }
  public static void pprint(Collection<?> c) {
    System.out.println(pformat(c));
  }
  public static void pprint(Object[] c) {
    System.out.println(pformat(Arrays.asList(c)));
  }
} ///:~
```

The **pformat( )** method produces a formatted **String** from a **Collection**, and the **pprint( )** method uses **pformat( )** to do its job. Note that the special cases of no elements and a single element are handled differently. There's also a version of **pprint( )** for arrays.

The **Directory** utility is placed in the **net.mindview.util** package so that it is easily available. Here's a sample of how you can use it:

```
//: io/DirectoryDemo.java
// Sample use of Directory utilities.
import java.io.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DirectoryDemo {
  public static void main(String[] args) {
    // All directories:
    PPrint.pprint(Directory.walk(".").dirs);
    // All files beginning with 'T'
    for(File file : Directory.local(".", "T.*"))
      print(file);
    print("----------------------");
    // All Java files beginning with 'T':
```

```
    for(File file : Directory.walk(".", "T.*\\.java"))
      print(file);
    print("======================");
    // Class files containing "Z" or "z":
    for(File file : Directory.walk(".",".*[Zz].*\\.class"))
      print(file);
  }
} /* Output: (Sample)
[.\xfiles]
.\TestEOF.class
.\TestEOF.java
.\TransferTo.class
.\TransferTo.java
----------------------
.\TestEOF.java
.\TransferTo.java
.\xfiles\ThawAlien.java
======================
.\FreezeAlien.class
.\GZIPcompress.class
.\ZipCompress.class
*///:~
```

You may need to refresh your knowledge of regular expressions from the *Strings* chapter in order to understand the second arguments in **local( )** and **walk( )**.

We can take this a step further and create a tool that will walk directories *and* process the files within them according to a **Strategy** object (this is another example of the *Strategy* design pattern):

```
//: net/mindview/util/ProcessFiles.java
package net.mindview.util;
import java.io.*;

public class ProcessFiles {
  public interface Strategy {
    void process(File file);
  }
  private Strategy strategy;
  private String ext;
  public ProcessFiles(Strategy strategy, String ext) {
    this.strategy = strategy;
    this.ext = ext;
  }
```

```
  public void start(String[] args) {
    try {
      if(args.length == 0)
        processDirectoryTree(new File("."));
      else
        for(String arg : args) {
          File fileArg = new File(arg);
          if(fileArg.isDirectory())
            processDirectoryTree(fileArg);
          else {
            // Allow user to leave off extension:
            if(!arg.endsWith("." + ext))
              arg += "." + ext;
            strategy.process(
              new File(arg).getCanonicalFile());
          }
        }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public void
  processDirectoryTree(File root) throws IOException {
    for(File file : Directory.walk(
        root.getAbsolutePath(), ".*\\." + ext))
      strategy.process(file.getCanonicalFile());
  }
  // Demonstration of how to use it:
  public static void main(String[] args) {
    new ProcessFiles(new ProcessFiles.Strategy() {
      public void process(File file) {
        System.out.println(file);
      }
    }, "java").start(args);
  }
} /* (Execute to see output) *///:~
```

The **Strategy** interface is nested within **ProcessFiles**, so that if you want to implement it you must **implement ProcessFiles.Strategy**, which provides more context for the reader. **ProcessFiles** does all the work of finding the files that have a particular extension (the **ext** argument to the constructor), and when it finds a matching file, it simply hands it to the **Strategy** object (which is also an argument to the constructor).

If you don't give it any arguments, **ProcessFiles** assumes that you want to traverse all the directories off of the current directory. You can also specify a particular file, with or without the extension (it will add the extension if necessary), or one or more directories.

In **main( )** you see a basic example of how to use the tool; it prints the names of all the Java source files according to the command line that you provide.

Exercise 4: (2) Use **Directory.walk( )** to sum the sizes of all files in a directory tree whose names match a particular regular expression.

Exercise 5: (1) Modify **ProcessFiles.java** so that it matches a regular expression rather than a fixed extension.

Checking for and creating directories

The **File** class is more than just a representation for an existing file or directory. You can also use a **File** object to create a new directory or an entire directory path if it doesn't exist. You can also look at the characteristics of files (size, last modification date, read/write), see whether a **File** object represents a file or a directory, and delete a file. The following example shows some of the other methods available with the **File** class (see the JDK documentation from *http://java.sun.com* for the full set):

```
//: io/MakeDirectories.java
// Demonstrates the use of the File class to
// create directories and manipulate files.
// {Args: MakeDirectoriesTest}
import java.io.*;

public class MakeDirectories {
  private static void usage() {
    System.err.println(
      "Usage:MakeDirectories path1 ...\n" +
      "Creates each path\n" +
      "Usage:MakeDirectories -d path1 ...\n" +
      "Deletes each path\n" +
      "Usage:MakeDirectories -r path1 path2\n" +
      "Renames from path1 to path2");
    System.exit(1);
  }
  private static void fileData(File f) {
    System.out.println(
      "Absolute path: " + f.getAbsolutePath() +
```

```
      "\n Can read: " + f.canRead() +
      "\n Can write: " + f.canWrite() +
      "\n getName: " + f.getName() +
      "\n getParent: " + f.getParent() +
      "\n getPath: " + f.getPath() +
      "\n length: " + f.length() +
      "\n lastModified: " + f.lastModified());
    if(f.isFile())
      System.out.println("It's a file");
    else if(f.isDirectory())
      System.out.println("It's a directory");
  }
  public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
      if(args.length != 3) usage();
      File
        old = new File(args[1]),
        rname = new File(args[2]);
      old.renameTo(rname);
      fileData(old);
      fileData(rname);
      return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
      count++;
      del = true;
    }
    count--;
    while(++count < args.length) {
      File f = new File(args[count]);
      if(f.exists()) {
        System.out.println(f + " exists");
        if(del) {
          System.out.println("deleting..." + f);
          f.delete();
        }
      }
      else { // Doesn't exist
        if(!del) {
          f.mkdirs();
          System.out.println("created " + f);
```

```
        }
      }
      fileData(f);
    }
  }
} /* Output: (80% match)
created MakeDirectoriesTest
Absolute path: d:\aaa-TIJ4\code\io\MakeDirectoriesTest
 Can read: true
 Can write: true
 getName: MakeDirectoriesTest
 getParent: null
 getPath: MakeDirectoriesTest
 length: 0
 lastModified: 1101690308831
It's a directory
*///:~
```

In **fileData( )** you can see various file investigation methods used to display information about the file or directory path.

The first method that's exercised by **main( )** is **renameTo( )**, which allows you to rename (or move) a file to an entirely new path represented by the argument, which is another **File** object. This also works with directories of any length.

If you experiment with the preceding program, you'll find that you can make a directory path of any complexity, because **mkdirs( )** will do all the work for you.

Exercise 6: (5) Use **ProcessFiles** to find all the Java source-code files in a particular directory subtree that have been modified after a particular date.

Input and output

Programming language I/O libraries often use the abstraction of a *stream*, which represents any data source or sink as an object capable of producing or receiving pieces of data. The stream hides the details of what happens to the data inside the actual I/O device.

The Java library classes for I/O are divided by input and output, as you can see by looking at the class hierarchy in the JDK documentation. Through inheritance, everything derived from the **InputStream** or **Reader** classes

has basic methods called **read( )** for reading a single **byte** or an array of **byte**s. Likewise, everything derived from **OutputStream** or **Writer** classes has basic methods called **write( )** for writing a single **byte** or an array of **byte**s. However, you won't generally use these methods; they exist so that other classes can use them—these other classes provide a more useful interface. Thus, you'll rarely create your stream object by using a single class, but instead will layer multiple objects together to provide your desired functionality (this is the *Decorator* design pattern, as you shall see in this section). The fact that you create more than one object to produce a single stream is the primary reason that Java's I/O library is confusing.

It's helpful to categorize the classes by their functionality. In Java 1.0, the library designers started by deciding that all classes that had anything to do with input would be inherited from **InputStream**, and all classes that were associated with output would be inherited from **OutputStream**.

As is the practice in this book, I will attempt to provide an overview of the classes, but assume that you will use the JDK documentation to determine all the details, such as the exhaustive list of methods of a particular class.

Types of **InputStream**

InputStream's job is to represent classes that produce input from different sources. These sources can be:

1. An array of bytes.
2. A **String** object.
3. A file.
4. A "pipe," which works like a physical pipe: You put things in at one end and they come out the other.
5. A sequence of other streams, so you can collect them together into a single stream.
6. Other sources, such as an Internet connection. (This is covered in *Thinking in Enterprise Java*, available at *www.MindView.net*.)

Each of these has an associated subclass of **InputStream**. In addition, the **FilterInputStream** is also a type of **InputStream**, to provide a base class

for "decorator" classes that attach attributes or useful interfaces to input streams. This is discussed later.

Table I/O-1. Types of InputStream

Class	Function	Constructor arguments How to use it
ByteArray-InputStream	Allows a buffer in memory to be used as an **InputStream**.	The buffer from which to extract the bytes.
		As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
StringBuffer-InputStream	Converts a **String** into an **InputStream**.	A **String**. The underlying implementation actually uses a **StringBuffer**.
		As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
File-InputStream	For reading information from a file.	A **String** representing the file name, or a **File** or **FileDescriptor** object.
		As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
Piped-InputStream	Produces the data that's being written to the associated **PipedOutput-Stream**. Implements the "piping" concept.	**PipedOutputStream**
		As a source of data in multithreading: Connect it to a **FilterInputStream** object to provide a useful interface.
Sequence-InputStream	Converts two or more **InputStream** objects into a single	Two **InputStream** objects or an **Enumeration** for a container of **InputStream** objects.

Class	Function	Constructor arguments
		How to use it
	InputStream.	As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
Filter-InputStream	Abstract class that is an interface for decorators that provide useful functionality to the other **InputStream** classes. See Table I/O-3.	See Table I/O-3.
		See Table I/O-3.

Types of **OutputStream**

This category includes the classes that decide where your output will go: an array of bytes (but not a **String**—presumably, you can create one using the array of bytes), a file, or a "pipe."

In addition, the **FilterOutputStream** provides a base class for "decorator" classes that attach attributes or useful interfaces to output streams. This is discussed later.

Table I/O-2. Types of OutputStream

Class	Function	Constructor arguments
		How to use it
ByteArray-OutputStream	Creates a buffer in memory. All the data that you send to the stream is placed in this buffer.	Optional initial size of the buffer.
		To designate the destination of your data: Connect it to a **FilterOutputStream** object to provide a useful interface.

Class	Function	Constructor arguments
		How to use it
File-OutputStream	For sending information to a file.	A **String** representing the file name, or a **File** or **FileDescriptor** object.
		To designate the destination of your data: Connect it to a **FilterOutputStream** object to provide a useful interface.
Piped-OutputStream	Any information you write to this automatically ends up as input for the associated **PipedInput-Stream**. Implements the "piping" concept.	**PipedInputStream**
		To designate the destination of your data for multithreading: Connect it to a **FilterOutputStream** object to provide a useful interface.
Filter-OutputStream	Abstract class that is an interface for decorators that provide useful functionality to the other **OutputStream** classes. See Table I/O-4.	See Table I/O-4.
		See Table I/O-4.

Adding attributes and useful interfaces

Decorators were introduced in the *Generics* chapter, on page 717. The Java I/O library requires many different combinations of features, and this is the justification for using the Decorator design pattern.[1] The reason for the

[1] It's not clear that this was a good design decision, especially compared to the simplicity of I/O libraries in other languages. But it's the justification for the decision.

existence of the "filter" classes in the Java I/O library is that the abstract "filter" class is the base class for all the decorators. A decorator must have the same interface as the object it decorates, but the decorator can also extend the interface, which occurs in several of the "filter" classes.

There is a drawback to Decorator, however. Decorators give you much more flexibility while you're writing a program (since you can easily mix and match attributes), but they add complexity to your code. The reason that the Java I/O library is awkward to use is that you must create many classes—the "core" I/O type plus all the decorators—in order to get the single I/O object that you want.

The classes that provide the decorator interface to control a particular **InputStream** or **OutputStream** are the **FilterInputStream** and **FilterOutputStream**, which don't have very intuitive names. **FilterInputStream** and **FilterOutputStream** are derived from the base classes of the I/O library, **InputStream** and **OutputStream**, which is a key requirement of the decorator (so that it provides the common interface to all the objects that are being decorated).

Reading from an **InputStream** with **FilterInputStream**

The **FilterInputStream** classes accomplish two significantly different things. **DataInputStream** allows you to read different types of primitive data as well as **String** objects. (All the methods start with "read," such as **readByte( )**, **readFloat( )**, etc.) This, along with its companion **DataOutputStream**, allows you to move primitive data from one place to another via a stream. These "places" are determined by the classes in Table I/O-1.

The remaining **FilterInputStream** classes modify the way an **InputStream** behaves internally: whether it's buffered or unbuffered, whether it keeps track of the lines it's reading (allowing you to ask for line numbers or set the line number), and whether you can push back a single character. The last two classes look a lot like support for building a compiler (they were probably added to support the experiment of "building a Java compiler in Java"), so you probably won't use them in general programming.

You'll need to buffer your input almost every time, regardless of the I/O device you're connecting to, so it would have made more sense for the I/O

library to have a special case (or simply a method call) for unbuffered input rather than buffered input.

Table I/O-3. Types of FilterInputStream

Class	Function	Constructor arguments
		How to use it
Data-InputStream	Used in concert with **DataOutputStream**, so you can read primitives (**int**, **char**, **long**, etc.) from a stream in a portable fashion.	**InputStream**
		Contains a full interface to allow you to read primitive types.
Buffered-InputStream	Use this to prevent a physical read every time you want more data. You're saying, "Use a buffer."	**InputStream**, with optional buffer size.
		This doesn't provide an interface per se. It just adds buffering to the process. Attach an interface object.
LineNumber-InputStream	Keeps track of line numbers in the input stream; you can call **getLineNumber( )** and **setLineNumber(int)**.	**InputStream**
		This just adds line numbering, so you'll probably attach an interface object.
Pushback-InputStream	Has a one-byte push-back buffer so that you can push back the last character read.	**InputStream**
		Generally used in the scanner for a compiler. You probably won't use this.

Writing to an **OutputStream** with **FilterOutputStream**

The complement to **DataInputStream** is **DataOutputStream**, which formats each of the primitive types and **String** objects onto a stream in such a way that any **DataInputStream**, on any machine, can read them. All the methods start with "write," such as **writeByte( )**, **writeFloat( )**, etc.

The original intent of **PrintStream** was to print all of the primitive data types and **String** objects in a viewable format. This is different from **DataOutputStream**, whose goal is to put data elements on a stream in a way that **DataInputStream** can portably reconstruct them.

The two important methods in **PrintStream** are **print( )** and **println( )**, which are overloaded to print all the various types. The difference between **print( )** and **println( )** is that the latter adds a newline when it's done.

PrintStream can be problematic because it traps all **IOException**s (you must explicitly test the error status with **checkError( )**, which returns **true** if an error has occurred). Also, **PrintStream** doesn't internationalize properly and doesn't handle line breaks in a platform-independent way. These problems are solved with **PrintWriter**, described later.

BufferedOutputStream is a modifier and tells the stream to use buffering so you don't get a physical write every time you write to the stream. You'll probably always want to use this when doing output.

Table I/O-4. Types of FilterOutputStream

Class	**Function**	**Constructor arguments** **How to use it**
Data-OutputStream	Used in concert with **DataInputStream** so you can write primitives (**int**, **char**, **long**, etc.) to a stream in a portable fashion.	**OutputStream** Contains a full interface to allow you to write primitive types.

Class	Function	Constructor arguments How to use it
PrintStream	For producing formatted output. While **DataOutputStream** handles the *storage* of data, **PrintStream** handles *display*.	**OutputStream**, with optional **boolean** indicating that the buffer is flushed with every newline.
		Should be the "final" wrapping for your **OutputStream** object. You'll probably use this a lot.
Buffered-OutputStream	Use this to prevent a physical write every time you send a piece of data. You're saying, "Use a buffer." You can call **flush( )** to flush the buffer.	**OutputStream**, with optional buffer size.
		This doesn't provide an interface per se. It just adds buffering to the process. Attach an interface object.

Readers & Writers

Java 1.1 made significant modifications to the fundamental I/O stream library. When you see the **Reader** and **Writer** classes, your first thought (like mine) might be that these were meant to replace the **InputStream** and **OutputStream** classes. But that's not the case. Although some aspects of the original streams library are deprecated (if you use them you will receive a warning from the compiler), the **InputStream** and **OutputStream** classes still provide valuable functionality in the form of byte-oriented I/O, whereas the **Reader** and **Writer** classes provide Unicode-compliant, character-based I/O. In addition:

1. Java 1.1 added new classes into the **InputStream** and **OutputStream** hierarchy, so it's obvious those hierarchies weren't being replaced.

2. There are times when you must use classes from the "byte" hierarchy *in combination* with classes in the "character" hierarchy. To accomplish this, there are "adapter" classes: **InputStreamReader** converts an **InputStream** to a **Reader**, and **OutputStreamWriter** converts an **OutputStream** to a **Writer**.

The most important reason for the **Reader** and **Writer** hierarchies is for internationalization. The old I/O stream hierarchy supports only 8-bit byte streams and doesn't handle the 16-bit Unicode characters well. Since Unicode is used for internationalization (and Java's native **char** is 16-bit Unicode), the **Reader** and **Writer** hierarchies were added to support Unicode in all I/O operations. In addition, the new libraries are designed for faster operations than the old.

Sources and sinks of data

Almost all of the original Java I/O stream classes have corresponding **Reader** and **Writer** classes to provide native Unicode manipulation. However, there are some places where the byte-oriented **InputStream**s and **OutputStream**s are the correct solution; in particular, the **java.util.zip** libraries are byte-oriented rather than **char**-oriented. So the most sensible approach to take is to *try* to use the **Reader** and **Writer** classes whenever you can. You'll discover the situations when you have to use the byte-oriented libraries because your code won't compile.

Here is a table that shows the correspondence between the sources and sinks of information (that is, where the data physically comes from or goes to) in the two hierarchies.

Sources & sinks: Java 1.0 class	**Corresponding Java 1.1 class**
InputStream	**Reader** adapter: **InputStreamReader**
OutputStream	**Writer** adapter: **OutputStreamWriter**
FileInputStream	**FileReader**
FileOutputStream	**FileWriter**

Sources & sinks: Java 1.0 class	Corresponding Java 1.1 class
StringBufferInputStream (deprecated)	**StringReader**
(no corresponding class)	**StringWriter**
ByteArrayInputStream	**CharArrayReader**
ByteArrayOutputStream	**CharArrayWriter**
PipedInputStream	**PipedReader**
PipedOutputStream	**PipedWriter**

In general, you'll find that the interfaces for the two different hierarchies are similar, if not identical.

Modifying stream behavior

For **InputStream**s and **OutputStream**s, streams were adapted for particular needs using "decorator" subclasses of **FilterInputStream** and **FilterOutputStream**. The **Reader** and **Writer** class hierarchies continue the use of this idea—but not exactly.

In the following table, the correspondence is a rougher approximation than in the previous table. The difference is because of the class organization; although **BufferedOutputStream** is a subclass of **FilterOutputStream**, **BufferedWriter** is *not* a subclass of **FilterWriter** (which, even though it is **abstract**, has no subclasses and so appears to have been put in either as a placeholder or simply so you don't wonder where it is). However, the interfaces to the classes are quite a close match.

Filters: Java 1.0 class	Corresponding Java 1.1 class
FilterInputStream	**FilterReader**
FilterOutputStream	**FilterWriter** (abstract class with no subclasses)
BufferedInputStream	**BufferedReader** (also has **readLine( )**)
BufferedOutputStream	**BufferedWriter**
DataInputStream	Use **DataInputStream** (except when you need to use

Filters: Java 1.0 class	Corresponding Java 1.1 class
	readLine( ), when you should use a **BufferedReader**)
PrintStream	**PrintWriter**
LineNumberInputStream (deprecated)	**LineNumberReader**
StreamTokenizer	**StreamTokenizer** (Use the constructor that takes a **Reader** instead)
PushbackInputStream	**PushbackReader**

There's one direction that's quite clear: Whenever you want to use **readLine( )**, you shouldn't do it with a **DataInputStream** (this is met with a deprecation message at compile time), but instead use a **BufferedReader**. Other than this, **DataInputStream** is still a "preferred" member of the I/O library.

To make the transition to using a **PrintWriter** easier, it has constructors that take any **OutputStream** object as well as **Writer** objects. **PrintWriter**'s formatting interface is virtually the same as **PrintStream**.

In Java SE5, **PrintWriter** constructors were added to simplify the creation of files when writing output, as you shall see shortly.

One **PrintWriter** constructor also has an option to perform automatic flushing, which happens after every **println( )** if the constructor flag is set.

Unchanged classes

Some classes were left unchanged between Java 1.0 and Java 1.1:

Java 1.0 classes without corresponding Java 1.1 classes
DataOutputStream
File
RandomAccessFile
SequenceInputStream

DataOutputStream, in particular, is used without change, so for storing and retrieving data in a transportable format, you use the **InputStream** and **OutputStream** hierarchies.

Off by itself: RandomAccessFile

RandomAccessFile is used for files containing records of known size so that you can move from one record to another using **seek( )**, then read or change the records. The records don't have to be the same size; you just have to determine how big they are and where they are placed in the file.

At first it's a little bit hard to believe that **RandomAccessFile** is not part of the **InputStream** or **OutputStream** hierarchy. However, it has no association with those hierarchies other than that it happens to implement the **DataInput** and **DataOutput** interfaces (which are also implemented by **DataInputStream** and **DataOutputStream**). It doesn't even use any of the functionality of the existing **InputStream** or **OutputStream** classes; it's a completely separate class, written from scratch, with all of its own (mostly native) methods. The reason for this may be that **RandomAccessFile** has essentially different behavior than the other I/O types, since you can move forward and backward within a file. In any event, it stands alone, as a direct descendant of **Object**.

Essentially, a **RandomAccessFile** works like a **DataInputStream** pasted together with a **DataOutputStream**, along with the methods **getFilePointer( )** to find out where you are in the file, **seek( )** to move to a new point in the file, and **length( )** to determine the maximum size of the file. In addition, the constructors require a second argument (identical to **fopen( )** in C) indicating whether you are just randomly reading (**"r"**) or reading and writing (**"rw"**). There's no support for write-only files, which could suggest that **RandomAccessFile** might have worked well if it were inherited from **DataInputStream**.

The seeking methods are available only in **RandomAccessFile**, which works for files only. **BufferedInputStream** does allow you to **mark( )** a position (whose value is held in a single internal variable) and **reset( )** to that position, but this is limited and not very useful.

Most, if not all, of the **RandomAccessFile** functionality is superseded as of JDK 1.4 with the **nio** *memory-mapped files*, which will be described later in this chapter.

Typical uses of I/O streams

Although you can combine the I/O stream classes in many different ways, you'll probably just use a few combinations. The following examples can be used as a basic reference for typical I/O usage.

In these examples, exception handing will be simplified by passing exceptions out to the console, but this is appropriate only in small examples and utilities. In your code you'll want to consider more sophisticated error-handling approaches.

Buffered input file

To open a file for character input, you use a **FileInputReader** with a **String** or a **File** object as the file name. For speed, you'll want that file to be buffered so you give the resulting reference to the constructor for a **BufferedReader**. Since **BufferedReader** also provides the **readLine( )** method, this is your final object and the interface you read from. When **readLine( )** returns **null**, you're at the end of the file.

```
//: io/BufferedInputFile.java
import java.io.*;

public class BufferedInputFile {
  // Throw exceptions to console:
  public static String
  read(String filename) throws IOException {
    // Reading input by lines:
    BufferedReader in = new BufferedReader(
      new FileReader(filename));
    String s;
    StringBuilder sb = new StringBuilder();
    while((s = in.readLine())!= null)
      sb.append(s + "\n");
    in.close();
    return sb.toString();
  }
  public static void main(String[] args)
  throws IOException {
```

```
    System.out.print(read("BufferedInputFile.java"));
  }
} /* (Execute to see output) *///:~
```

The **StringBuilder sb** is used to accumulate the entire contents of the file (including newlines that must be added since **readLine( )** strips them off). Finally, **close( )** is called to close the file.[2]

Exercise 7: (2) Open a text file so that you can read the file one line at a time. Read each line as a **String** and place that **String** object into a **LinkedList**. Print all of the lines in the **LinkedList** in reverse order.

Exercise 8: (1) Modify Exercise 7 so that the name of the file you read is provided as a command-line argument.

Exercise 9: (1) Modify Exercise 8 to force all the lines in the **LinkedList** to uppercase and send the results to **System.out**.

Exercise 10: (2) Modify Exercise 8 to take additional command-line arguments of words to find in the file. Print all lines in which any of the words match.

Exercise 11: (2) In the **innerclasses/GreenhouseController.java** example, **GreenhouseController** contains a hard-coded set of events. Change the program so that it reads the events and their relative times from a text file. ((difficulty level 8): Use a *Factory Method* design pattern to build the events—see *Thinking in Patterns (with Java)* at *www.MindView.net*.)

Input from memory

Here, the **String** result from **BufferedInputFile.read( )** is used to create a **StringReader**. Then **read( )** is used to read each character one at a time and send it out to the console:

```
//: io/MemoryInput.java
import java.io.*;

public class MemoryInput {
```

[2] In the original design, **close( )** was supposed to be called when **finalize( )** ran, and you will see **finalize( )** defined this way for I/O classes. However, as is discussed elsewhere in this book, the **finalize( )** feature didn't work out the way the Java designers originally envisioned it (that is to say, it's irreparably broken), so the only safe approach is to explicitly call **close( )** for files.

```
  public static void main(String[] args)
  throws IOException {
    StringReader in = new StringReader(
      BufferedInputFile.read("MemoryInput.java"));
    int c;
    while((c = in.read()) != -1)
      System.out.print((char)c);
  }
} /* (Execute to see output) *///:~
```

Note that **read( )** returns the next character as an **int** and thus it must be cast to a **char** to print properly.

Formatted memory input

To read "formatted" data, you use a **DataInputStream**, which is a byte-oriented I/O class (rather than **char**-oriented). Thus you must use all **InputStream** classes rather than **Reader** classes. Of course, you can read anything (such as a file) as bytes using **InputStream** classes, but here a **String** is used:

```
//: io/FormattedMemoryInput.java
import java.io.*;

public class FormattedMemoryInput {
  public static void main(String[] args)
  throws IOException {
    try {
      DataInputStream in = new DataInputStream(
        new ByteArrayInputStream(
          BufferedInputFile.read(
            "FormattedMemoryInput.java").getBytes()));
      while(true)
        System.out.print((char)in.readByte());
    } catch(EOFException e) {
      System.err.println("End of stream");
    }
  }
} /* (Execute to see output) *///:~
```

A **ByteArrayInputStream** must be given an array of bytes. To produce this, **String** has a **getBytes( )** method. The resulting **ByteArrayInputStream** is an appropriate **InputStream** to hand to **DataInputStream**.

If you read the characters from a **DataInputStream** one **byte** at a time using **readByte( )**, any **byte** value is a legitimate result, so the return value cannot be used to detect the end of input. Instead, you can use the **available( )** method to find out how many more characters are available. Here's an example that shows how to read a file one **byte** at a time:

```
//: io/TestEOF.java
// Testing for end of file while reading a byte at a time.
import java.io.*;

public class TestEOF {
  public static void main(String[] args)
  throws IOException {
    DataInputStream in = new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("TestEOF.java")));
    while(in.available() != 0)
      System.out.print((char)in.readByte());
  }
} /* (Execute to see output) *///:~
```

Note that **available( )** works differently depending on what sort of medium you're reading from; it's literally "the number of bytes that can be read *without blocking*." With a file, this means the whole file, but with a different kind of stream this might not be true, so use it thoughtfully.

You could also detect the end of input in cases like these by catching an exception. However, the use of exceptions for control flow is considered a misuse of that feature.

Basic file output

A **FileWriter** object writes data to a file. You'll virtually always want to buffer the output by wrapping it in a **BufferedWriter** (try removing this wrapping to see the impact on the performance—buffering tends to dramatically increase performance of I/O operations). In this example, it's decorated as a **PrintWriter** to provide formatting. The data file created this way is readable as an ordinary text file:

```
//: io/BasicFileOutput.java
import java.io.*;

public class BasicFileOutput {
```

```
  static String file = "BasicFileOutput.out";
  public static void main(String[] args)
  throws IOException {
    BufferedReader in = new BufferedReader(
      new StringReader(
        BufferedInputFile.read("BasicFileOutput.java")));
    PrintWriter out = new PrintWriter(
      new BufferedWriter(new FileWriter(file)));
    int lineCount = 1;
    String s;
    while((s = in.readLine()) != null )
      out.println(lineCount++ + ": " + s);
    out.close();
    // Show the stored file:
    System.out.println(BufferedInputFile.read(file));
  }
} /* (Execute to see output) *///:~
```

As the lines are written to the file, line numbers are added. Note that **LineNumberReader** is *not* used, because it's a silly class and you don't need it. You can see from this example that it's trivial to keep track of your own line numbers.

When the input stream is exhausted, **readLine( )** returns **null**. You'll see an explicit **close( )** for **out**, because if you don't call **close( )** for all your output files, you might discover that the buffers don't get flushed, so the file will be incomplete.

Text file output shortcut

Java SE5 added a helper constructor to **PrintWriter** so that you don't have to do all the decoration by hand every time you want to create a text file and write to it. Here's **BasicFileOutput.java** rewritten to use this shortcut:

```
//: io/FileOutputShortcut.java
import java.io.*;

public class FileOutputShortcut {
  static String file = "FileOutputShortcut.out";
  public static void main(String[] args)
  throws IOException {
    BufferedReader in = new BufferedReader(
      new StringReader(
       BufferedInputFile.read("FileOutputShortcut.java")));
    // Here's the shortcut:
```

```
    PrintWriter out = new PrintWriter(file);
    int lineCount = 1;
    String s;
    while((s = in.readLine()) != null )
      out.println(lineCount++ + ": " + s);
    out.close();
    // Show the stored file:
    System.out.println(BufferedInputFile.read(file));
  }
} /* (Execute to see output) *///:~
```

You still get buffering, you just don't have to do it yourself. Unfortunately, other commonly written tasks were not given shortcuts, so typical I/O will still involve a lot of redundant text. However, the **TextFile** utility that is used in this book, and which will be defined a little later in this chapter, does simplify these common tasks.

Exercise 12: (3) Modify Exercise 8 to also open a text file so you can write text into it. Write the lines in the **LinkedList**, along with line numbers (do not attempt to use the "LineNumber" classes), out to the file.

Exercise 13: (3) Modify **BasicFileOutput.java** so that it uses **LineNumberReader** to keep track of the line count. Note that it's much easier to just keep track programmatically.

Exercise 14: (2) Starting with **BasicFileOutput.java**, write a program that compares the performance of writing to a file when using buffered and unbuffered I/O.

Storing and recovering data

A **PrintWriter** formats data so that it's readable by a human. However, to output data for recovery by another stream, you use a **DataOutputStream** to write the data and a **DataInputStream** to recover the data. Of course, these streams can be anything, but the following example uses a file, buffered for both reading and writing. **DataOutputStream** and **DataInputStream** are byte-oriented and thus require **InputStream**s and **OutputStream**s:

```
//: io/StoringAndRecoveringData.java
import java.io.*;

public class StoringAndRecoveringData {
  public static void main(String[] args)
  throws IOException {
```

```
    DataOutputStream out = new DataOutputStream(
      new BufferedOutputStream(
        new FileOutputStream("Data.txt")));
    out.writeDouble(3.14159);
    out.writeUTF("That was pi");
    out.writeDouble(1.41413);
    out.writeUTF("Square root of 2");
    out.close();
    DataInputStream in = new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("Data.txt")));
    System.out.println(in.readDouble());
    // Only readUTF() will recover the
    // Java-UTF String properly:
    System.out.println(in.readUTF());
    System.out.println(in.readDouble());
    System.out.println(in.readUTF());
  }
} /* Output:
3.14159
That was pi
1.41413
Square root of 2
*///:~
```

If you use a **DataOutputStream** to write the data, then Java guarantees that you can accurately recover the data using a **DataInputStream**—regardless of what different platforms write and read the data. This is incredibly valuable, as anyone knows who has spent time worrying about platform-specific data issues. That problem vanishes if you have Java on both platforms.[3]

When you are using a **DataOutputStream**, the only reliable way to write a **String** so that it can be recovered by a **DataInputStream** is to use UTF-8 encoding, accomplished in this example using **writeUTF( )** and **readUTF( )**. UTF-8 is a multi-byte format, and the length of encoding varies according to the actual character set in use. If you're working with ASCII or mostly ASCII characters (which occupy only seven bits), Unicode is a

---

[3] XML is another way to solve the problem of moving data across different computing platforms, and does not depend on having Java on all platforms. XML is introduced later in this chapter.

tremendous waste of space and/or bandwidth, so UTF-8 encodes ASCII characters in a single byte, and non-ASCII characters in two or three bytes. In addition, the length of the string is stored in the first two bytes of the UTF-8 string. However, **writeUTF( )** and **readUTF( )** use a special variation of UTF-8 for Java (which is completely described in the JDK documentation for those methods), so if you read a string written with **writeUTF( )** using a non-Java program, you must write special code in order to read the string properly.

With **writeUTF( )** and **readUTF( )**, you can intermingle **String**s and other types of data using a **DataOutputStream**, with the knowledge that the **String**s will be properly stored as Unicode and will be easily recoverable with a **DataInputStream**.

The **writeDouble( )** method stores the **double** number to the stream, and the complementary **readDouble( )** method recovers it (there are similar methods for reading and writing the other types). But for any of the reading methods to work correctly, you must know the exact placement of the data item in the stream, since it would be equally possible to read the stored **double** as a simple sequence of bytes, or as a **char**, etc. So you must either have a fixed format for the data in the file, or extra information must be stored in the file that you parse to determine where the data is located. Note that object serialization or XML (both described later in this chapter) may be easier ways to store and retrieve complex data structures.

Exercise 15: (4) Look up **DataOutputStream** and **DataInputStream** in the JDK documentation. Starting with **StoringAndRecoveringData.java**, create a program that stores and then retrieves all the different possible types provided by the **DataOutputStream** and **DataInputStream** classes. Verify that the values are stored and retrieved accurately.

Reading and writing random-access files

Using a **RandomAccessFile** is like using a combined **DataInputStream** and **DataOutputStream** (because it implements the same interfaces: **DataInput** and **DataOutput**). In addition, you can use **seek( )** to move about in the file and change the values.

When using **RandomAccessFile**, you must know the layout of the file so that you can manipulate it properly. **RandomAccessFile** has specific methods to read and write primitives and UTF-8 strings. Here's an example:

```
//: io/UsingRandomAccessFile.java
import java.io.*;

public class UsingRandomAccessFile {
  static String file = "rtest.dat";
  static void display() throws IOException {
    RandomAccessFile rf = new RandomAccessFile(file, "r");
    for(int i = 0; i < 7; i++)
      System.out.println(
        "Value " + i + ": " + rf.readDouble());
    System.out.println(rf.readUTF());
    rf.close();
  }
  public static void main(String[] args)
  throws IOException {
    RandomAccessFile rf = new RandomAccessFile(file, "rw");
    for(int i = 0; i < 7; i++)
      rf.writeDouble(i*1.414);
    rf.writeUTF("The end of the file");
    rf.close();
    display();
    rf = new RandomAccessFile(file, "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();
    display();
  }
} /* Output:
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 7.069999999999999
Value 6: 8.484
The end of the file
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
```

```
Value 4: 5.656
Value 5: 47.0001
Value 6: 8.484
The end of the file
*///:~
```

The **display( )** method opens a file and displays seven elements within as **double** values. In **main( )**, the file is created, then opened and modified. Since a **double** is always eight bytes long, to **seek( )** to double number 5 you just multiply **5*8** to produce the seek value.

As previously noted, **RandomAccessFile** is effectively separate from the rest of the I/O hierarchy, save for the fact that it implements the **DataInput** and **DataOutput** interfaces. It doesn't support decoration, so you cannot combine it with any of the aspects of the **InputStream** and **OutputStream** subclasses. You must assume that a **RandomAccessFile** is properly buffered since you cannot add that.

The one option you have is in the second constructor argument: You can open a **RandomAccessFile** to read (**"r"**) or read and write (**"rw"**).

You may want to consider using **nio** memory-mapped files instead of **RandomAccessFile**.

Exercise 16: (2) Look up **RandomAccessFile** in the JDK documentation. Starting with **UsingRandomAccessFile.java**, create a program that stores and then retrieves all the different possible types provided by the **RandomAccessFile** class. Verify that the values are stored and retrieved accurately.

Piped streams

The **PipedInputStream**, **PipedOutputStream**, **PipedReader** and **PipedWriter** have been mentioned only briefly in this chapter. This is not to suggest that they aren't useful, but their value is not apparent until you begin to understand concurrency, since the piped streams are used to communicate between tasks. This is covered along with an example in the *Concurrency* chapter.

File reading & writing utilities

A very common programming task is to read a file into memory, modify it, and then write it out again. One of the problems with the Java I/O library is

that it requires you to write quite a bit of code in order to perform these common operations—there are no basic helper functions to do them for you. What's worse, the decorators make it rather hard to remember how to open files. Thus, it makes sense to add helper classes to your library that will easily perform these basic tasks for you. Java SE5 has added a convenience constructor to **PrintWriter** so you can easily open a text file for writing. However, there are many other common tasks that you will want to do over and over, and it makes sense to eliminate the redundant code associated with those tasks.

Here's the **TextFile** class that has been used in previous examples in this book to simplify reading and writing files. It contains **static** methods to read and write text files as a single string, and you can create a **TextFile** object that holds the lines of the file in an **ArrayList** (so you have all the **ArrayList** functionality while manipulating the file contents):

```
//: net/mindview/util/TextFile.java
// Static functions for reading and writing text files as
// a single string, and treating a file as an ArrayList.
package net.mindview.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList<String> {
  // Read a file as a single string:
  public static String read(String fileName) {
    StringBuilder sb = new StringBuilder();
    try {
      BufferedReader in= new BufferedReader(new FileReader(
        new File(fileName).getAbsoluteFile()));
      try {
        String s;
        while((s = in.readLine()) != null) {
          sb.append(s);
          sb.append("\n");
        }
      } finally {
        in.close();
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return sb.toString();
```

```
  }
  // Write a single file in one method call:
  public static void write(String fileName, String text) {
    try {
      PrintWriter out = new PrintWriter(
        new File(fileName).getAbsoluteFile());
      try {
        out.print(text);
      } finally {
        out.close();
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  // Read a file, split by any regular expression:
  public TextFile(String fileName, String splitter) {
    super(Arrays.asList(read(fileName).split(splitter)));
    // Regular expression split() often leaves an empty
    // String at the first position:
    if(get(0).equals("")) remove(0);
  }
  // Normally read by lines:
  public TextFile(String fileName) {
    this(fileName, "\n");
  }
  public void write(String fileName) {
    try {
      PrintWriter out = new PrintWriter(
        new File(fileName).getAbsoluteFile());
      try {
        for(String item : this)
          out.println(item);
      } finally {
        out.close();
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  // Simple test:
  public static void main(String[] args) {
    String file = read("TextFile.java");
    write("test.txt", file);
```

```
    TextFile text = new TextFile("test.txt");
    text.write("test2.txt");
    // Break into unique sorted list of words:
    TreeSet<String> words = new TreeSet<String>(
      new TextFile("TextFile.java", "\\W+"));
    // Display the capitalized words:
    System.out.println(words.headSet("a"));
  }
} /* Output:
[0, ArrayList, Arrays, Break, BufferedReader,
BufferedWriter, Clean, Display, File, FileReader,
FileWriter, IOException, Normally, Output, PrintWriter,
Read, Regular, RuntimeException, Simple, Static, String,
StringBuilder, System, TextFile, Tools, TreeSet, W, Write]
*///:~
```

read( ) appends each line to a **StringBuilder**, followed by a newline, because that is stripped out during reading. Then it returns a **String** containing the whole file. **write( )** opens and writes the text **String** to the file.

Notice that any code that opens a file guards the file's **close( )** call in a **finally** clause to guarantee that the file will be properly closed.

The constructor uses the **read( )** method to turn the file into a **String**, then uses **String.split( )** to divide the result into lines along newline boundaries (if you use this class a lot, you may want to rewrite this constructor to improve efficiency). Alas, there is no corresponding "join" method, so the non-**static write( )** method must write the lines out by hand.

Because this class is intended to trivialize the process of reading and writing files, all **IOException**s are converted to **RuntimeException**s, so the user doesn't have to use **try-catch** blocks. However, you may need to create another version that passes **IOException**s out to the caller.

In **main( )**, a basic test is performed to ensure that the methods work.

Although this utility did not require much code to create, using it can save a lot of time and make your life easier, as you'll see in some of the examples later in this chapter.

Another way to solve the problem of reading text files is to use the **java.util.Scanner** class introduced in Java SE5. However, this is only for reading files, not writing them, and that tool (which you'll notice is *not* in

java.io) is primarily designed for creating programming-language scanners or "little languages."

Exercise 17: (4) Using **TextFile** and a **Map<Character,Integer>**, create a program that counts the occurrence of all the different characters in a file. (So if there are 12 occurrences of the letter 'a' in the file, the **Integer** associated with the **Character** containing 'a' in the **Map** contains '12').

Exercise 18: (1) Modify **TextFile.java** so that it passes **IOExceptions** out to the caller.

Reading binary files

This utility is similar to **TextFile.java** in that it simplifies the process of reading binary files:

```
//: net/mindview/util/BinaryFile.java
// Utility for reading files in binary form.
package net.mindview.util;
import java.io.*;

public class BinaryFile {
  public static byte[] read(File bFile) throws IOException{
    BufferedInputStream bf = new BufferedInputStream(
      new FileInputStream(bFile));
    try {
      byte[] data = new byte[bf.available()];
      bf.read(data);
      return data;
    } finally {
      bf.close();
    }
  }
  public static byte[]
  read(String bFile) throws IOException {
    return read(new File(bFile).getAbsoluteFile());
  }
} ///:~
```

One overloaded method takes a **File** argument; the second takes a **String** argument, which is the file name. Both return the resulting **byte** array.

The **available( )** method is used to produce the appropriate array size, and this particular version of the overloaded **read( )** method fills the array.

Exercise 19: (2) Using **BinaryFile** and a **Map<Byte,Integer>**, create a program that counts the occurrence of all the different bytes in a file.

Exercise 20: (4) Using **Directory.walk( )** and **BinaryFile**, verify that all **.class** files in a directory tree begin with the hex characters '**CAFEBABE**'.

Standard I/O

The term *standard I/O* refers to the Unix concept of a single stream of information that is used by a program (this idea is reproduced in some form in Windows and many other operating systems). All of the program's input can come from *standard input*, all of its output can go to *standard output*, and all of its error messages can be sent to *standard error*. The value of standard I/O is that programs can easily be chained together, and one program's standard output can become the standard input for another program. This is a powerful tool.

Reading from standard input

Following the standard I/O model, Java has **System.in**, **System.out**, and **System.err**. Throughout this book, you've seen how to write to standard output using **System.out**, which is already pre-wrapped as a **PrintStream** object. **System.err** is likewise a **PrintStream**, but **System.in** is a raw **InputStream** with no wrapping. This means that although you can use **System.out** and **System.err** right away, **System.in** must be wrapped before you can read from it.

You'll typically read input a line at a time using **readLine( )**. To do this, wrap **System.in** in a **BufferedReader**, which requires you to convert **System.in** to a **Reader** using **InputStreamReader**. Here's an example that simply echoes each line that you type in:

```
//: io/Echo.java
// How to read from standard input.
// {RunByHand}
import java.io.*;

public class Echo {
  public static void main(String[] args)
  throws IOException {
    BufferedReader stdin = new BufferedReader(
      new InputStreamReader(System.in));
    String s;
```

```
    while((s = stdin.readLine()) != null && s.length()!= 0)
      System.out.println(s);
    // An empty line or Ctrl-Z terminates the program
  }
} ///:~
```

The reason for the exception specification is that **readLine( )** can throw an **IOException**. Note that **System.in** should usually be buffered, as with most streams.

Exercise 21: (1) Write a program that takes standard input and capitalizes all characters, then puts the results on standard output. Redirect the contents of a file into this program (the process of redirection will vary depending on your operating system).

Changing **System.out** to a **PrintWriter**

System.out is a **PrintStream**, which is an **OutputStream**. **PrintWriter** has a constructor that takes an **OutputStream** as an argument. Thus, if you want, you can convert **System.out** into a **PrintWriter** using that constructor:

```
//: io/ChangeSystemOut.java
// Turn System.out into a PrintWriter.
import java.io.*;

public class ChangeSystemOut {
  public static void main(String[] args) {
    PrintWriter out = new PrintWriter(System.out, true);
    out.println("Hello, world");
  }
} /* Output:
Hello, world
*///:~
```

It's important to use the two-argument version of the **PrintWriter** constructor and to set the second argument to **true** in order to enable automatic flushing; otherwise, you may not see the output.

Redirecting standard I/O

The Java **System** class allows you to redirect the standard input, output, and error I/O streams using simple **static** method calls:

setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)

Redirecting output is especially useful if you suddenly start creating a large amount of output on your screen, and it's scrolling past faster than you can read it.[4] Redirecting input is valuable for a command-line program in which you want to test a particular user-input sequence repeatedly. Here's a simple example that shows the use of these methods:

```
//: io/Redirecting.java
// Demonstrates standard I/O redirection.
import java.io.*;

public class Redirecting {
  public static void main(String[] args)
  throws IOException {
    PrintStream console = System.out;
    BufferedInputStream in = new BufferedInputStream(
      new FileInputStream("Redirecting.java"));
    PrintStream out = new PrintStream(
      new BufferedOutputStream(
        new FileOutputStream("test.out")));
    System.setIn(in);
    System.setOut(out);
    System.setErr(out);
    BufferedReader br = new BufferedReader(
      new InputStreamReader(System.in));
    String s;
    while((s = br.readLine()) != null)
      System.out.println(s);
    out.close(); // Remember this!
    System.setOut(console);
  }
} ///:~
```

This program attaches standard input to a file and redirects standard output and standard error to another file. Notice that it stores a reference to the

---

[4] The *Graphical User Interfaces* chapter shows an even more convenient solution for this: a GUI program with a scrolling text area.

original **System.out** object at the beginning of the program, and restores the system output to that object at the end.

I/O redirection manipulates streams of bytes, not streams of characters; thus, **InputStream**s and **OutputStream**s are used rather than **Reader**s and **Writer**s.

Process control

You will often need to execute other operating system programs from inside Java, and to control the input and output from such programs. The Java library provides classes to perform such operations.

A common task is to run a program and send the resulting output to the console. This section contains a utility to simplify this task.

Two types of errors can occur with this utility: the normal errors that result in exceptions—for these we will just rethrow a runtime exception—and errors from the execution of the process itself. We want to report these errors with a separate exception:

```
//: net/mindview/util/OSExecuteException.java
package net.mindview.util;

public class OSExecuteException extends RuntimeException {
  public OSExecuteException(String why) { super(why); }
} ///:~
```

To run a program, you pass **OSExecute.command( )** a **command** string, which is the same command that you would type to run the program on the console. This command is passed to the **java.lang.ProcessBuilder** constructor (which requires it as a sequence of **String** objects), and the resulting **ProcessBuilder** object is started:

```
//: net/mindview/util/OSExecute.java
// Run an operating system command
// and send the output to the console.
package net.mindview.util;
import java.io.*;

public class OSExecute {
  public static void command(String command) {
    boolean err = false;
    try {
```

```
      Process process =
        new ProcessBuilder(command.split(" ")).start();
      BufferedReader results = new BufferedReader(
        new InputStreamReader(process.getInputStream()));
      String s;
      while((s = results.readLine())!= null)
        System.out.println(s);
      BufferedReader errors = new BufferedReader(
        new InputStreamReader(process.getErrorStream()));
      // Report errors and return nonzero value
      // to calling process if there are problems:
      while((s = errors.readLine())!= null) {
        System.err.println(s);
        err = true;
      }
    } catch(Exception e) {
      // Compensate for Windows 2000, which throws an
      // exception for the default command line:
      if(!command.startsWith("CMD /C"))
        command("CMD /C " + command);
      else
        throw new RuntimeException(e);
    }
    if(err)
      throw new OSExecuteException("Errors executing " +
        command);
  }
} ///:~
```

To capture the standard output stream from the program as it executes, you call **getInputStream( )**. This is because an **InputStream** is something we can read from.

The results from the program arrive a line at a time, so they are read using **readLine( )**. Here the lines are simply printed, but you may also want to capture and return them from **command( )**.

The program's errors are sent to the standard error stream, and are captured by calling **getErrorStream( )**. If there are any errors, they are printed and an **OSExecuteException** is thrown so the calling program will handle the problem.

Here's an example that shows how to use **OSExecute**:

```
//: io/OSExecuteDemo.java
// Demonstrates standard I/O redirection.
import net.mindview.util.*;

public class OSExecuteDemo {
  public static void main(String[] args) {
    OSExecute.command("javap OSExecuteDemo");
  }
} /* Output:
Compiled from "OSExecuteDemo.java"
public class OSExecuteDemo extends java.lang.Object{
    public OSExecuteDemo();
    public static void main(java.lang.String[]);
}
*///:~
```

This uses the **javap** decompiler (that comes with the JDK) to decompile the program.

Exercise 22: (5) Modify **OSExecute.java** so that, instead of printing the standard output stream, it returns the results of executing the program as a **List** of **String**s. Demonstrate the use of this new version of the utility.

New I/O

The Java "new" I/O library, introduced in JDK 1.4 in the **java.nio.*** packages, has one goal: speed. In fact, the "old" I/O packages have been reimplemented using **nio** in order to take advantage of this speed increase, so you will benefit even if you don't explicitly write code with **nio**. The speed increase occurs both in file I/O, which is explored here, and in network I/O, which is covered in *Thinking in Enterprise Java*.

The speed comes from using structures that are closer to the operating system's way of performing I/O: *channels* and *buffers*. You could think of it as a coal mine; the channel is the mine containing the seam of coal (the data), and the buffer is the cart that you send into the mine. The cart comes back full of coal, and you get the coal from the cart. That is, you don't interact directly with the channel; you interact with the buffer and send the buffer into the channel. The channel either pulls data from the buffer, or puts data into the buffer.

The only kind of buffer that communicates directly with a channel is a **ByteBuffer**—that is, a buffer that holds raw bytes. If you look at the JDK

documentation for **java.nio.ByteBuffer**, you'll see that it's fairly basic: You create one by telling it how much storage to allocate, and there are methods to put and get data, in either raw byte form or as primitive data types. But there's no way to put or get an object, or even a **String**. It's fairly low-level, precisely because this makes a more efficient mapping with most operating systems.

Three of the classes in the "old" I/O have been modified so that they produce a **FileChannel**: **FileInputStream**, **FileOutputStream**, and, for both reading and writing, **RandomAccessFile**. Notice that these are the byte manipulation streams, in keeping with the low-level nature of **nio**. The **Reader** and **Writer** character-mode classes do not produce channels, but the **java.nio.channels.Channels** class has utility methods to produce **Reader**s and **Writer**s from channels.

Here's a simple example that exercises all three types of stream to produce channels that are writeable, read/writeable, and readable:

```
//: io/GetChannel.java
// Getting channels from streams
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class GetChannel {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    // Write a file:
    FileChannel fc =
      new FileOutputStream("data.txt").getChannel();
    fc.write(ByteBuffer.wrap("Some text ".getBytes()));
    fc.close();
    // Add to the end of the file:
    fc =
      new RandomAccessFile("data.txt", "rw").getChannel();
    fc.position(fc.size()); // Move to the end
    fc.write(ByteBuffer.wrap("Some more".getBytes()));
    fc.close();
    // Read the file:
    fc = new FileInputStream("data.txt").getChannel();
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    fc.read(buff);
    buff.flip();
```

```
    while(buff.hasRemaining())
      System.out.print((char)buff.get());
  }
} /* Output:
Some text Some more
*///:~
```

For any of the stream classes shown here, **getChannel( )** will produce a **FileChannel**. A channel is fairly basic: You can hand it a **ByteBuffer** for reading or writing, and you can lock regions of the file for exclusive access (this will be described later).

One way to put bytes into a **ByteBuffer** is to stuff them in directly using one of the "put" methods, to put one or more bytes, or values of primitive types. However, as seen here, you can also "wrap" an existing **byte** array in a **ByteBuffer** using the **wrap( )** method. When you do this, the underlying array is not copied, but instead is used as the storage for the generated **ByteBuffer**. We say that the **ByteBuffer** is "backed by" the array.

The **data.txt** file is reopened using a **RandomAccessFile**. Notice that you can move the **FileChannel** around in the file; here, it is moved to the end so that additional writes will be appended.

For read-only access, you must explicitly allocate a **ByteBuffer** using the **static allocate( )** method. The goal of **nio** is to rapidly move large amounts of data, so the size of the **ByteBuffer** should be significant—in fact, the 1K used here is probably quite a bit smaller than you'd normally want to use (you'll have to experiment with your working application to find the best size).

It's also possible to go for even more speed by using **allocateDirect( )** instead of **allocate( )** to produce a "direct" buffer that may have an even higher coupling with the operating system. However, the overhead in such an allocation is greater, and the actual implementation varies from one operating system to another, so again, you must experiment with your working application to discover whether direct buffers will buy you any advantage in speed.

Once you call **read( )** to tell the **FileChannel** to store bytes into the **ByteBuffer**, you must call **flip( )** on the buffer to tell it to get ready to have its bytes extracted (yes, this seems a bit crude, but remember that it's very low-level and is done for maximum speed). And if we were to use the buffer

for further **read( )** operations, we'd also have to call **clear( )** to prepare it for each **read( )**. You can see this in a simple file-copying program:

```
//: io/ChannelCopy.java
// Copying a file using channels and buffers
// {Args: ChannelCopy.java test.txt}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ChannelCopy {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    if(args.length != 2) {
      System.out.println("arguments: sourcefile destfile");
      System.exit(1);
    }
    FileChannel
      in = new FileInputStream(args[0]).getChannel(),
      out = new FileOutputStream(args[1]).getChannel();
    ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
    while(in.read(buffer) != -1) {
      buffer.flip(); // Prepare for writing
      out.write(buffer);
      buffer.clear();  // Prepare for reading
    }
  }
} ///:~
```

You can see that one **FileChannel** is opened for reading, and one for writing. A **ByteBuffer** is allocated, and when **FileChannel.read( )** returns **-1** (a holdover, no doubt, from Unix and C), it means that you've reached the end of the input. After each **read( )**, which puts data into the buffer, **flip( )** prepares the buffer so that its information can be extracted by the **write( )**. After the **write( )**, the information is still in the buffer, and **clear( )** resets all the internal pointers so that it's ready to accept data during another **read( )**.

The preceding program is not the ideal way to handle this kind of operation, however. Special methods **transferTo( )** and **transferFrom( )** allow you to connect one channel directly to another:

```
//: io/TransferTo.java
// Using transferTo() between channels
// {Args: TransferTo.java TransferTo.txt}
```

```
import java.nio.channels.*;
import java.io.*;

public class TransferTo {
  public static void main(String[] args) throws Exception {
    if(args.length != 2) {
      System.out.println("arguments: sourcefile destfile");
      System.exit(1);
    }
    FileChannel
      in = new FileInputStream(args[0]).getChannel(),
      out = new FileOutputStream(args[1]).getChannel();
    in.transferTo(0, in.size(), out);
    // Or:
    // out.transferFrom(in, 0, in.size());
  }
} ///:~
```

You won't do this kind of thing very often, but it's good to know about.

Converting data

If you look back at **GetChannel.java**, you'll notice that, to print the information in the file, we are pulling the data out one **byte** at a time and casting each **byte** to a **char**. This seems a bit primitive—if you look at the **java.nio.CharBuffer** class, you'll see that it has a **toString( )** method that says, "Returns a string containing the characters in this buffer." Since a **ByteBuffer** can be viewed as a **CharBuffer** with the **asCharBuffer( )** method, why not use that? As you can see from the first line in the output statement below, this doesn't work out:

```
//: io/BufferToText.java
// Converting text to and from ByteBuffers
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.io.*;

public class BufferToText {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    FileChannel fc =
      new FileOutputStream("data2.txt").getChannel();
    fc.write(ByteBuffer.wrap("Some text".getBytes()));
```

```
    fc.close();
    fc = new FileInputStream("data2.txt").getChannel();
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    fc.read(buff);
    buff.flip();
    // Doesn't work:
    System.out.println(buff.asCharBuffer());
    // Decode using this system's default Charset:
    buff.rewind();
    String encoding = System.getProperty("file.encoding");
    System.out.println("Decoded using " + encoding + ": "
      + Charset.forName(encoding).decode(buff));
    // Or, we could encode with something that will print:
    fc = new FileOutputStream("data2.txt").getChannel();
    fc.write(ByteBuffer.wrap(
      "Some text".getBytes("UTF-16BE")));
    fc.close();
    // Now try reading again:
    fc = new FileInputStream("data2.txt").getChannel();
    buff.clear();
    fc.read(buff);
    buff.flip();
    System.out.println(buff.asCharBuffer());
    // Use a CharBuffer to write through:
    fc = new FileOutputStream("data2.txt").getChannel();
    buff = ByteBuffer.allocate(24); // More than needed
    buff.asCharBuffer().put("Some text");
    fc.write(buff);
    fc.close();
    // Read and display:
    fc = new FileInputStream("data2.txt").getChannel();
    buff.clear();
    fc.read(buff);
    buff.flip();
    System.out.println(buff.asCharBuffer());
  }
} /* Output:
????
Decoded using Cp1252: Some text
Some text
Some text
*///:~
```

The buffer contains plain bytes, and to turn these into characters, we must either *encode* them as we put them in (so that they will be meaningful when

they come out) or *decode* them as they come out of the buffer. This can be accomplished using the **java.nio.charset.Charset** class, which provides tools for encoding into many different types of character sets:

```
//: io/AvailableCharSets.java
// Displays Charsets and aliases
import java.nio.charset.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class AvailableCharSets {
  public static void main(String[] args) {
    SortedMap<String,Charset> charSets =
      Charset.availableCharsets();
    Iterator<String> it = charSets.keySet().iterator();
    while(it.hasNext()) {
      String csName = it.next();
      printnb(csName);
      Iterator aliases =
        charSets.get(csName).aliases().iterator();
      if(aliases.hasNext())
        printnb(": ");
      while(aliases.hasNext()) {
        printnb(aliases.next());
        if(aliases.hasNext())
          printnb(", ");
      }
      print();
    }
  }
} /* Output:
Big5: csBig5
Big5-HKSCS: big5-hkscs, big5hk, big5-hkscs:unicode3.0,
big5hkscs, Big5_HKSCS
EUC-JP: eucjis, x-eucjp, csEUCPkdFmtjapanese, eucjp,
Extended_UNIX_Code_Packed_Format_for_Japanese, x-euc-jp,
euc_jp
EUC-KR: ksc5601, 5601, ksc5601_1987, ksc_5601, ksc5601-
1987, euc_kr, ks_c_5601-1987, euckr, csEUCKR
GB18030: gb18030-2000
GB2312: gb2312-1980, gb2312, EUC_CN, gb2312-80, euc-cn,
euccn, x-EUC-CN
GBK: windows-936, CP936
...
```

```
*///:~
```

So, returning to **BufferToText.java**, if you **rewind( )** the buffer (to go back to the beginning of the data) and then use that platform's default character set to **decode( )** the data, the resulting **CharBuffer** will print to the console just fine. To discover the default character set, use **System.getProperty("file.encoding")**, which produces the string that names the character set. Passing this to **Charset.forName( )** produces the **Charset** object that can be used to decode the string.

Another alternative is to **encode( )** using a character set that will result in something printable when the file is read, as you see in the third part of **BufferToText.java**. Here, UTF-16BE is used to write the text into the file, and when it is read, all you must do is convert it to a **CharBuffer**, and it produces the expected text.

Finally, you see what happens if you *write* to the **ByteBuffer** through a **CharBuffer** (you'll learn more about this later). Note that 24 bytes are allocated for the **ByteBuffer**. Since each **char** requires two bytes, this is enough for 12 **char**s, but "Some text" only has 9. The remaining zero bytes still appear in the representation of the **CharBuffer** produced by its **toString( )**, as you can see in the output.

Exercise 23: (6) Create and test a utility method to print the contents of a **CharBuffer** up to the point where the characters are no longer printable.

Fetching primitives

Although a **ByteBuffer** only holds bytes, it contains methods to produce each of the different types of primitive values from the bytes it contains. This example shows the insertion and extraction of various values using these methods:

```
//: io/GetData.java
// Getting different representations from a ByteBuffer
import java.nio.*;
import static net.mindview.util.Print.*;

public class GetData {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    // Allocation automatically zeroes the ByteBuffer:
```

```
    int i = 0;
    while(i++ < bb.limit())
      if(bb.get() != 0)
        print("nonzero");
    print("i = " + i);
    bb.rewind();
    // Store and read a char array:
    bb.asCharBuffer().put("Howdy!");
    char c;
    while((c = bb.getChar()) != 0)
      printnb(c + " ");
    print();
    bb.rewind();
    // Store and read a short:
    bb.asShortBuffer().put((short)471142);
    print(bb.getShort());
    bb.rewind();
    // Store and read an int:
    bb.asIntBuffer().put(99471142);
    print(bb.getInt());
    bb.rewind();
    // Store and read a long:
    bb.asLongBuffer().put(99471142);
    print(bb.getLong());
    bb.rewind();
    // Store and read a float:
    bb.asFloatBuffer().put(99471142);
    print(bb.getFloat());
    bb.rewind();
    // Store and read a double:
    bb.asDoubleBuffer().put(99471142);
    print(bb.getDouble());
    bb.rewind();
  }
} /* Output:
i = 1025
H o w d y !
12390
99471142
99471142
9.9471144E7
9.9471142E7
*///:~
```

After a **ByteBuffer** is allocated, its values are checked to see whether buffer allocation automatically zeroes the contents—and it does. All 1,024 values are checked (up to the **limit( )** of the buffer), and all are zero.

The easiest way to insert primitive values into a **ByteBuffer** is to get the appropriate "view" on that buffer using **asCharBuffer( )**, **asShortBuffer( )**, etc., and then to use that view's **put( )** method. You can see this is the process used for each of the primitive data types. The only one of these that is a little odd is the **put( )** for the **ShortBuffer**, which requires a cast (note that the cast truncates and changes the resulting value). All the other view buffers do not require casting in their **put( )** methods.

View buffers

A "view buffer" allows you to look at an underlying **ByteBuffer** through the window of a particular primitive type. The **ByteBuffer** is still the actual storage that's "backing" the view, so any changes you make to the view are reflected in modifications to the data in the **ByteBuffer**. As seen in the previous example, this allows you to conveniently insert primitive types into a **ByteBuffer**. A view also allows you to read primitive values from a **ByteBuffer**, either one at a time (as **ByteBuffer** allows) or in batches (into arrays). Here's an example that manipulates **int**s in a **ByteBuffer** via an **IntBuffer**:

```
//: io/IntBufferDemo.java
// Manipulating ints in a ByteBuffer with an IntBuffer
import java.nio.*;

public class IntBufferDemo {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    IntBuffer ib = bb.asIntBuffer();
    // Store an array of int:
    ib.put(new int[]{ 11, 42, 47, 99, 143, 811, 1016 });
    // Absolute location read and write:
    System.out.println(ib.get(3));
    ib.put(3, 1811);
    // Setting a new limit before rewinding the buffer.
    ib.flip();
    while(ib.hasRemaining()) {
      int i = ib.get();
      System.out.println(i);
```

```
    }
  }
} /* Output:
99
11
42
47
1811
143
811
1016
*///:~
```

The overloaded **put( )** method is first used to store an array of **int**. The following **get( )** and **put( )** method calls directly access an **int** location in the underlying **ByteBuffer**. Note that these absolute location accesses are available for primitive types by talking directly to a **ByteBuffer**, as well.

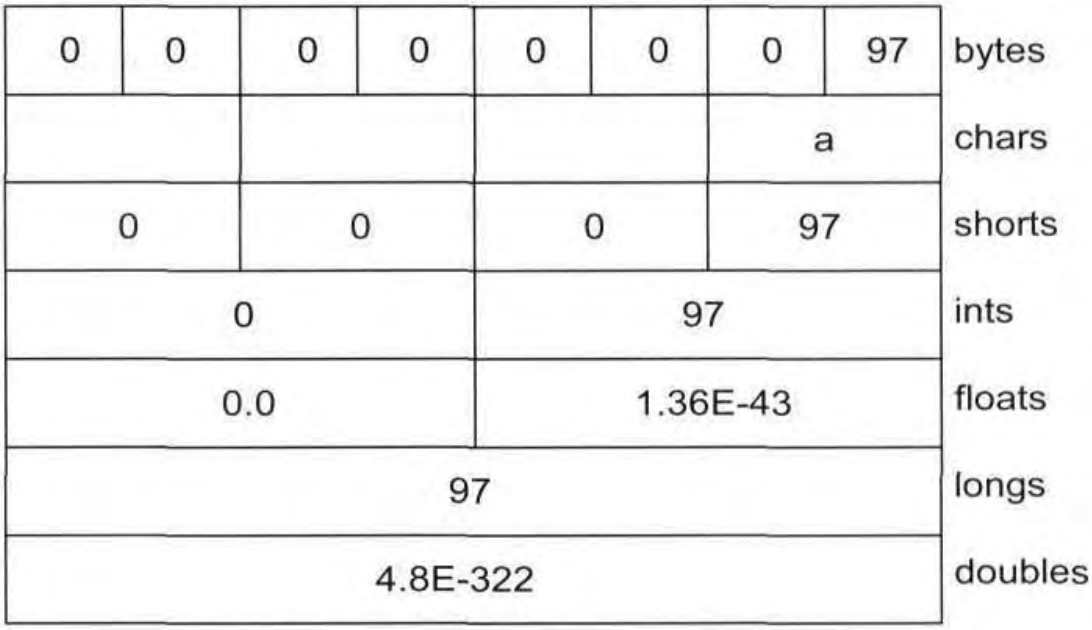
Once the underlying **ByteBuffer** is filled with **int**s or some other primitive type via a view buffer, then that **ByteBuffer** can be written directly to a channel. You can just as easily read from a channel and use a view buffer to convert everything to a particular type of primitive. Here's an example that interprets the same sequence of bytes as **short**, **int**, **float**, **long**, and **double** by producing different view buffers on the same **ByteBuffer**:

```
//: io/ViewBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class ViewBuffers {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(
      new byte[]{ 0, 0, 0, 0, 0, 0, 0, 'a' });
    bb.rewind();
    printnb("Byte Buffer ");
    while(bb.hasRemaining())
      printnb(bb.position()+ " -> " + bb.get() + ", ");
    print();
    CharBuffer cb =
      ((ByteBuffer)bb.rewind()).asCharBuffer();
    printnb("Char Buffer ");
    while(cb.hasRemaining())
      printnb(cb.position() + " -> " + cb.get() + ", ");
    print();
```

```
    FloatBuffer fb =
      ((ByteBuffer)bb.rewind()).asFloatBuffer();
    printnb("Float Buffer ");
    while(fb.hasRemaining())
      printnb(fb.position()+ " -> " + fb.get() + ", ");
    print();
    IntBuffer ib =
      ((ByteBuffer)bb.rewind()).asIntBuffer();
    printnb("Int Buffer ");
    while(ib.hasRemaining())
      printnb(ib.position()+ " -> " + ib.get() + ", ");
    print();
    LongBuffer lb =
      ((ByteBuffer)bb.rewind()).asLongBuffer();
    printnb("Long Buffer ");
    while(lb.hasRemaining())
      printnb(lb.position()+ " -> " + lb.get() + ", ");
    print();
    ShortBuffer sb =
      ((ByteBuffer)bb.rewind()).asShortBuffer();
    printnb("Short Buffer ");
    while(sb.hasRemaining())
      printnb(sb.position()+ " -> " + sb.get() + ", ");
    print();
    DoubleBuffer db =
      ((ByteBuffer)bb.rewind()).asDoubleBuffer();
    printnb("Double Buffer ");
    while(db.hasRemaining())
      printnb(db.position()+ " -> " + db.get() + ", ");
  }
} /* Output:
Byte Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5 -> 0,
6 -> 0, 7 -> 97,
Char Buffer 0 ->  , 1 ->  , 2 ->  , 3 -> a,
Float Buffer 0 -> 0.0, 1 -> 1.36E-43,
Int Buffer 0 -> 0, 1 -> 97,
Long Buffer 0 -> 97,
Short Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
Double Buffer 0 -> 4.8E-322,
*///:~
```

The **ByteBuffer** is produced by "wrapping" an eight-**byte** array, which is then displayed via view buffers of all the different primitive types. You can see

in the following diagram the way the data appears differently when read from the different types of buffers:



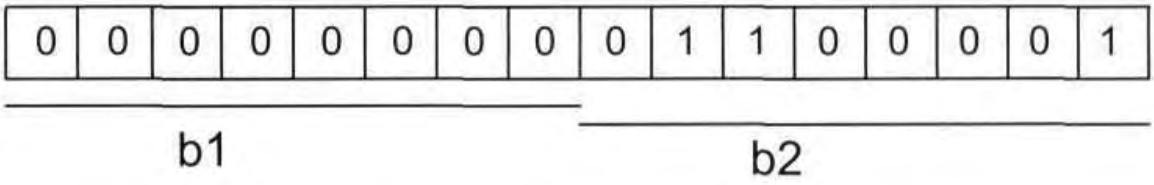
This corresponds to the output from the program.

Exercise 24: (1) Modify **IntBufferDemo.java** to use **double**s.

Endians

Different machines may use different byte-ordering approaches to store data. "Big endian" places the most significant byte in the lowest memory address, and "little endian" places the most significant byte in the highest memory address. When storing a quantity that is greater than one **byte**, like **int**, **float**, etc., you may need to consider the byte ordering. A **ByteBuffer** stores data in big endian form, and data sent over a network always uses big endian order. You can change the endian-ness of a **ByteBuffer** using **order( )** with an argument of **ByteOrder.BIG_ENDIAN** or **ByteOrder.LITTLE_ENDIAN**.

Consider a **ByteBuffer** containing the following two bytes:



If you read the data as a **short** (**ByteBuffer.asShortBuffer( )**), you will get the number 97 (00000000 01100001), but if you change to little endian, you will get the number 24832 (01100001 00000000).

Here's an example that shows how byte ordering is changed in characters depending on the endian setting:

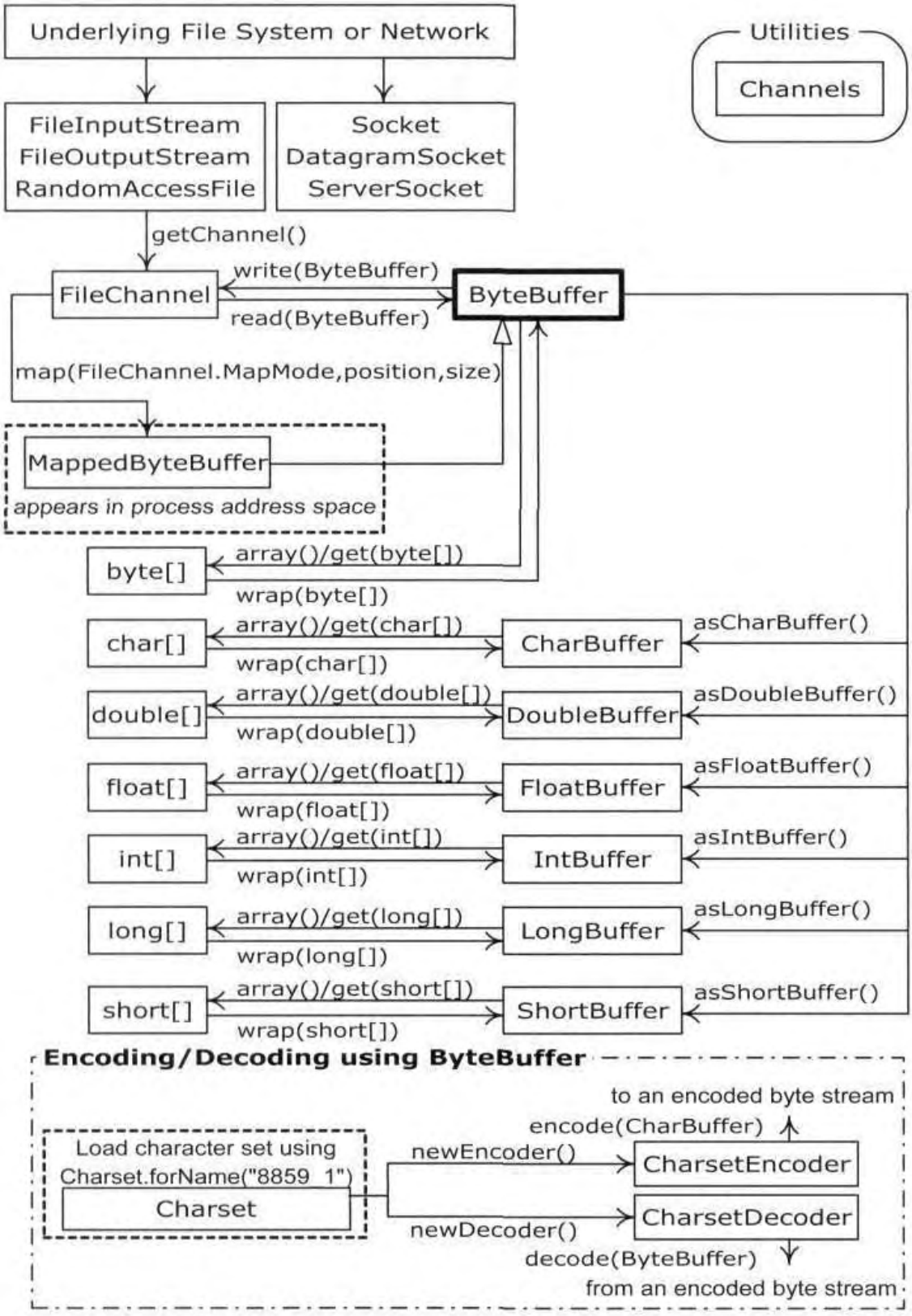
```
//: io/Endians.java
// Endian differences and data storage.
import java.nio.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Endians {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.BIG_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.LITTLE_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
  }
} /* Output:
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]
*///:~
```

The **ByteBuffer** is given enough space to hold all the bytes in **charArray** as an external buffer so that the **array( )** method can be called to display the underlying bytes. The **array( )** method is "optional," and you can only call it on a buffer that is backed by an array; otherwise, you'll get an **UnsupportedOperationException**.

charArray is inserted into the **ByteBuffer** via a **CharBuffer** view. When the underlying bytes are displayed, you can see that the default ordering is the same as the subsequent big endian order, whereas the little endian order swaps the bytes.

Data manipulation with buffers

The following diagram illustrates the relationships between the **nio** classes, so that you can see how to move and convert data. For example, if you wish to write a **byte** array to a file, then you wrap the **byte** array using the **ByteBuffer.wrap( )** method, open a channel on the **FileOutputStream** using the **getChannel( )** method, and then write data into **FileChannel** from this **ByteBuffer**.



Note that **ByteBuffer** is the only way to move data into and out of channels, and that you can only create a standalone primitive-typed buffer, or get one

from a **ByteBuffer** using an "as" method. That is, you cannot convert a primitive-typed buffer *to* a **ByteBuffer**. However, since you are able to move primitive data into and out of a **ByteBuffer** via a view buffer, this is not really a restriction.

Buffer details

A **Buffer** consists of data and four indexes to access and manipulate this data efficiently: *mark*, *position*, *limit* and *capacity*. There are methods to set and reset these indexes and to query their value.

capacity( )	Returns the buffer's *capacity*.
clear( )	Clears the buffer, sets the *position* to zero, and *limit* to *capacity*. You call this method to overwrite an existing buffer.
flip( )	Sets *limit* to *position* and *position* to zero. This method is used to prepare the buffer for a read after data has been written into it.
limit( )	Returns the value of *limit*.
limit(int lim)	Sets the value of *limit*.
mark( )	Sets *mark* at *position*.
position( )	Returns the value of *position*.
position(int pos)	Sets the value of *position*.
remaining( )	Returns (*limit* - *position*).
hasRemaining( )	Returns **true** if there are any elements between *position* and *limit*.

Methods that insert and extract data from the buffer update these indexes to reflect the changes.

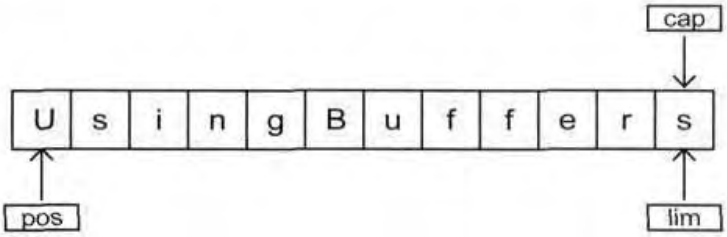
This example uses a very simple algorithm (swapping adjacent characters) to scramble and unscramble characters in a **CharBuffer**:

```
//: io/UsingBuffers.java
```

```
import java.nio.*;
import static net.mindview.util.Print.*;

public class UsingBuffers {
  private static void symmetricScramble(CharBuffer buffer){
    while(buffer.hasRemaining()) {
      buffer.mark();
      char c1 = buffer.get();
      char c2 = buffer.get();
      buffer.reset();
      buffer.put(c2).put(c1);
    }
  }
  public static void main(String[] args) {
    char[] data = "UsingBuffers".toCharArray();
    ByteBuffer bb = ByteBuffer.allocate(data.length * 2);
    CharBuffer cb = bb.asCharBuffer();
    cb.put(data);
    print(cb.rewind());
    symmetricScramble(cb);
    print(cb.rewind());
    symmetricScramble(cb);
    print(cb.rewind());
  }
} /* Output:
UsingBuffers
sUniBgfuefsr
UsingBuffers
*///:~
```

Although you could produce a **CharBuffer** directly by calling **wrap( )** with a **char** array, an underlying **ByteBuffer** is allocated instead, and a **CharBuffer** is produced as a view on the **ByteBuffer**. This emphasizes that the goal is always to manipulate a **ByteBuffer**, since that is what interacts with a channel.

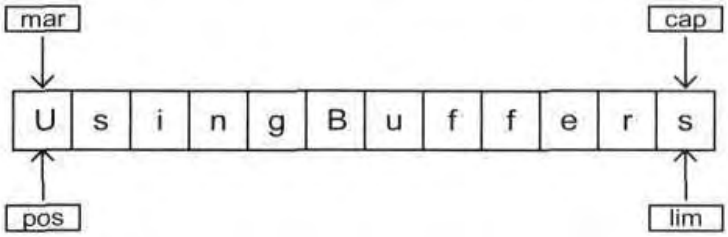
Here's what the buffer looks like at the entrance of the **symmetricScramble( )** method:



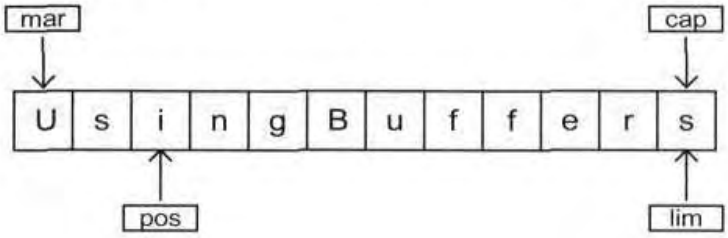
The *position* points to the first element in the buffer, and the *capacity* and *limit* point to the last element.

In **symmetricScramble( )**, the **while** loop iterates until *position* is equivalent to *limit*. The *position* of the buffer changes when a relative **get( )** or **put( )** function is called on it. You can also call absolute **get( )** and **put( )** methods that include an index argument, which is the location where the **get( )** or **put( )** takes place. These methods do not modify the value of the buffer's *position*.

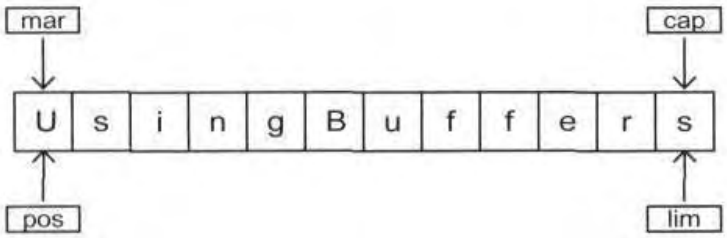
When the control enters the **while** loop, the value of *mark* is set using a **mark( )** call. The state of the buffer is then:



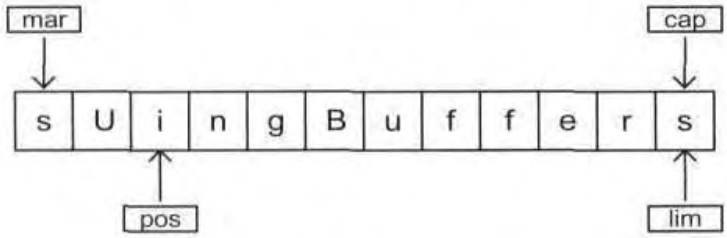
The two relative **get( )** calls save the value of the first two characters in variables **c1** and **c2**. After these two calls, the buffer looks like this:



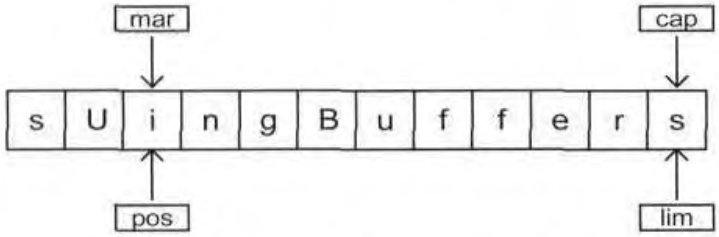
To perform the swap, we need to write **c2** at *position* = 0 and **c1** at *position* = 1. We can either use the absolute put method to achieve this, or set the value of *position* to *mark*, which is what **reset( )** does:



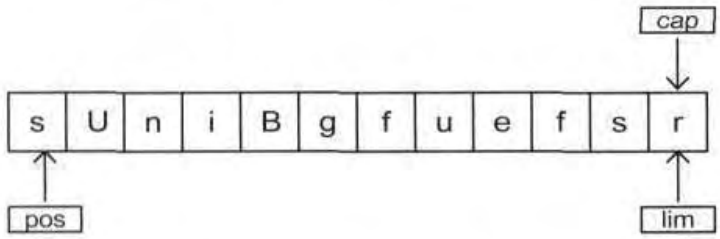
The two **put( )** methods write **c2** and then **c1**:



During the next iteration of the loop, *mark* is set to the current value of *position*:



The process continues until the entire buffer is traversed. At the end of the **while** loop, *position* is at the end of the buffer. If you print the buffer, only the characters between the *position* and *limit* are printed. Thus, if you want to show the entire contents of the buffer, you must set *position* to the start of the buffer using **rewind( )**. Here is the state of buffer after the **rewind( )** call (the value of *mark* becomes undefined):



When the function **symmetricScramble( )** is called again, the **CharBuffer** undergoes the same process and is restored to its original state.

Memory-mapped files

Memory-mapped files allow you to create and modify files that are too big to bring into memory. With a memory-mapped file, you can pretend that the entire file is in memory and that you can access it by simply treating it as a very large array. This approach greatly simplifies the code you write in order to modify the file. Here's a small example:

```
//: io/LargeMappedFiles.java
// Creating a very large file using mapping.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class LargeMappedFiles {
  static int length = 0x8FFFFFF; // 128 MB
  public static void main(String[] args) throws Exception {
    MappedByteBuffer out =
      new RandomAccessFile("test.dat", "rw").getChannel()
      .map(FileChannel.MapMode.READ_WRITE, 0, length);
    for(int i = 0; i < length; i++)
      out.put((byte)'x');
    print("Finished writing");
    for(int i = length/2; i < length/2 + 6; i++)
      printnb((char)out.get(i));
  }
} ///:~
```

To do both writing and reading, we start with a **RandomAccessFile**, get a channel for that file, and then call **map( )** to produce a **MappedByteBuffer**, which is a particular kind of direct buffer. Note that you must specify the starting point and the length of the region that you want to map in the file; this means that you have the option to map smaller regions of a large file.

MappedByteBuffer is inherited from **ByteBuffer**, so it has all of **ByteBuffer**'s methods. Only the very simple uses of **put( )** and **get( )** are shown here, but you can also use methods like **asCharBuffer( )**, etc.

The file created with the preceding program is 128 MB long, which is probably larger than your OS will allow in memory at one time. The file

appears to be accessible all at once because only portions of it are brought into memory, and other parts are swapped out. This way a very large file (up to 2 GB) can easily be modified. Note that the file-mapping facilities of the underlying operating system are used to maximize performance.

Performance

Although the performance of "old" stream I/O has been improved by implementing it with **nio**, mapped file access tends to be dramatically faster. This program does a simple performance comparison:

```
//: io/MappedIO.java
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class MappedIO {
  private static int numOfInts = 4000000;
  private static int numOfUbuffInts = 200000;
  private abstract static class Tester {
    private String name;
    public Tester(String name) { this.name = name; }
    public void runTest() {
      System.out.print(name + ": ");
      try {
        long start = System.nanoTime();
        test();
        double duration = System.nanoTime() - start;
        System.out.format("%.2f\n", duration/1.0e9);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
    public abstract void test() throws IOException;
  }
  private static Tester[] tests = {
    new Tester("Stream Write") {
      public void test() throws IOException {
        DataOutputStream dos = new DataOutputStream(
          new BufferedOutputStream(
            new FileOutputStream(new File("temp.tmp"))));
        for(int i = 0; i < numOfInts; i++)
          dos.writeInt(i);
        dos.close();
```

```
    }
  },
  new Tester("Mapped Write") {
    public void test() throws IOException {
      FileChannel fc =
        new RandomAccessFile("temp.tmp", "rw")
        .getChannel();
      IntBuffer ib = fc.map(
        FileChannel.MapMode.READ_WRITE, 0, fc.size())
        .asIntBuffer();
      for(int i = 0; i < numOfInts; i++)
        ib.put(i);
      fc.close();
    }
  },
  new Tester("Stream Read") {
    public void test() throws IOException {
      DataInputStream dis = new DataInputStream(
        new BufferedInputStream(
          new FileInputStream("temp.tmp")));
      for(int i = 0; i < numOfInts; i++)
        dis.readInt();
      dis.close();
    }
  },
  new Tester("Mapped Read") {
    public void test() throws IOException {
      FileChannel fc = new FileInputStream(
        new File("temp.tmp")).getChannel();
      IntBuffer ib = fc.map(
        FileChannel.MapMode.READ_ONLY, 0, fc.size())
        .asIntBuffer();
      while(ib.hasRemaining())
        ib.get();
      fc.close();
    }
  },
  new Tester("Stream Read/Write") {
    public void test() throws IOException {
      RandomAccessFile raf = new RandomAccessFile(
        new File("temp.tmp"), "rw");
      raf.writeInt(1);
      for(int i = 0; i < numOfUbuffInts; i++) {
        raf.seek(raf.length() - 4);
```

```
            raf.writeInt(raf.readInt());
          }
          raf.close();
        }
      },
      new Tester("Mapped Read/Write") {
        public void test() throws IOException {
          FileChannel fc = new RandomAccessFile(
            new File("temp.tmp"), "rw").getChannel();
          IntBuffer ib = fc.map(
            FileChannel.MapMode.READ_WRITE, 0, fc.size())
            .asIntBuffer();
          ib.put(0);
          for(int i = 1; i < numOfUbuffInts; i++)
            ib.put(ib.get(i - 1));
          fc.close();
        }
      }
    };
    public static void main(String[] args) {
      for(Tester test : tests)
        test.runTest();
    }
} /* Output: (90% match)
Stream Write: 0.56
Mapped Write: 0.12
Stream Read: 0.80
Mapped Read: 0.07
Stream Read/Write: 5.32
Mapped Read/Write: 0.02
*///:~
```

As seen in earlier examples in this book, **runTest( )** is used by the *Template Method* to create a testing framework for various implementations of **test( )** defined in anonymous inner subclasses. Each of these subclasses performs one kind of test, so the **test( )** methods also give you a prototype for performing the various I/O activities.

Although a mapped write would seem to use a **FileOutputStream**, all output in file mapping must use a **RandomAccessFile**, just as read/write does in the preceding code.

Note that the **test( )** methods include the time for initialization of the various I/O objects, so even though the setup for mapped files can be expensive, the overall gain compared to stream I/O is significant.

Exercise 25: (6) Experiment with changing the **ByteBuffer.allocate( )** statements in the examples in this chapter to **ByteBuffer.allocateDirect( )**. Demonstrate performance differences, but also notice whether the startup time of the programs noticeably changes.

Exercise 26: (3) Modify **strings/JGrep.java** to use Java **nio** memory-mapped files.

File locking

File locking allows you to synchronize access to a file as a shared resource. However, two threads that contend for the same file may be in different JVMs, or one may be a Java thread and the other some native thread in the operating system. The file locks are visible to other operating system processes because Java file locking maps directly to the native operating system locking facility.

Here is a simple example of file locking.

```
//: io/FileLocking.java
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class FileLocking {
  public static void main(String[] args) throws Exception {
    FileOutputStream fos= new FileOutputStream("file.txt");
    FileLock fl = fos.getChannel().tryLock();
    if(fl != null) {
      System.out.println("Locked File");
      TimeUnit.MILLISECONDS.sleep(100);
      fl.release();
      System.out.println("Released Lock");
    }
    fos.close();
  }
} /* Output:
Locked File
Released Lock
*///:~
```

You get a **FileLock** on the entire file by calling either **tryLock( )** or **lock( )** on a **FileChannel**. (**SocketChannel**, **DatagramChannel**, and **ServerSocketChannel** do not need locking since they are inherently single-process entities; you don't generally share a network socket between two processes.) **tryLock( )** is non-blocking. It tries to grab the lock, but if it cannot (when some other process already holds the same lock and it is not shared), it simply returns from the method call. **lock( )** blocks until the lock is acquired, or the thread that invoked **lock( )** is interrupted, or the channel on which the **lock( )** method is called is closed. A lock is released using **FileLock.release( )**.

It is also possible to lock a part of the file by using

```
tryLock(long position, long size, boolean shared)
```

or

```
lock(long position, long size, boolean shared)
```

which locks the region **(size - position)**. The third argument specifies whether this lock is shared.

Although the zero-argument locking methods adapt to changes in the size of a file, locks with a fixed size do not change if the file size changes. If a lock is acquired for a region from **position** to **position+size** and the file increases beyond **position+size**, then the section beyond **position+size** is not locked. The zero-argument locking methods lock the entire file, even if it grows.

Support for exclusive or shared locks must be provided by the underlying operating system. If the operating system does not support shared locks and a request is made for one, an exclusive lock is used instead. The type of lock (shared or exclusive) can be queried using **FileLock.isShared( )**.

Locking portions of a mapped file

As mentioned earlier, file mapping is typically used for very large files. You may need to lock portions of such a large file so that other processes may modify unlocked parts of the file. This is something that happens, for example, with a database, so that it can be available to many users at once.

Here's an example that has two threads, each of which locks a distinct portion of a file:

```
//: io/LockingMappedFiles.java
// Locking portions of a mapped file.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LockingMappedFiles {
  static final int LENGTH = 0x8FFFFFF; // 128 MB
  static FileChannel fc;
  public static void main(String[] args) throws Exception {
    fc =
      new RandomAccessFile("test.dat", "rw").getChannel();
    MappedByteBuffer out =
      fc.map(FileChannel.MapMode.READ_WRITE, 0, LENGTH);
    for(int i = 0; i < LENGTH; i++)
      out.put((byte)'x');
    new LockAndModify(out, 0, 0 + LENGTH/3);
    new LockAndModify(out, LENGTH/2, LENGTH/2 + LENGTH/4);
  }
  private static class LockAndModify extends Thread {
    private ByteBuffer buff;
    private int start, end;
    LockAndModify(ByteBuffer mbb, int start, int end) {
      this.start = start;
      this.end = end;
      mbb.limit(end);
      mbb.position(start);
      buff = mbb.slice();
      start();
    }
    public void run() {
      try {
        // Exclusive lock with no overlap:
        FileLock fl = fc.lock(start, end, false);
        System.out.println("Locked: "+ start +" to "+ end);
        // Perform modification:
        while(buff.position() < buff.limit() - 1)
          buff.put((byte)(buff.get() + 1));
        fl.release();
        System.out.println("Released: "+start+" to "+ end);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
```

```
    }
  }
} ///:~
```

The **LockAndModify** thread class sets up the buffer region and creates a **slice( )** to be modified, and in **run( )**, the lock is acquired on the file channel (you can't acquire a lock on the buffer—only the channel). The call to **lock( )** is very similar to acquiring a threading lock on an object—you now have a "critical section" with exclusive access to that portion of the file.[5]

The locks are automatically released when the JVM exits, or the channel on which it was acquired is closed, but you can also explicitly call **release( )** on the **FileLock** object, as shown here.

Compression

The Java I/O library contains classes to support reading and writing streams in a compressed format. You wrap these around other I/O classes to provide compression functionality.

These classes are not derived from the **Reader** and **Writer** classes, but instead are part of the **InputStream** and **OutputStream** hierarchies. This is because the compression library works with bytes, not characters. However, you might sometimes be forced to mix the two types of streams. (Remember that you can use **InputStreamReader** and **OutputStreamWriter** to provide easy conversion between one type and another.)

Compression class	Function
CheckedInputStream	**GetCheckSum( )** produces checksum for any **InputStream** (not just decompression).
CheckedOutputStream	**GetCheckSum( )** produces checksum for any **OutputStream** (not just compression).
DeflaterOutputStream	Base class for compression classes.
ZipOutputStream	A **DeflaterOutputStream** that

[5] More details about threads will be found in the *Concurrency* chapter.

Compression class	Function
	compresses data into the Zip file format.
GZIPOutputStream	A **DeflaterOutputStream** that compresses data into the GZIP file format.
InflaterInputStream	Base class for decompression classes.
ZipInputStream	An **InflaterInputStream** that decompresses data that has been stored in the Zip file format.
GZIPInputStream	An **InflaterInputStream** that decompresses data that has been stored in the GZIP file format.

Although there are many compression algorithms, Zip and GZIP are possibly the most commonly used. Thus you can easily manipulate your compressed data with the many tools available for reading and writing these formats.

Simple compression with GZIP

The GZIP interface is simple and thus is probably more appropriate when you have a single stream of data that you want to compress (rather than a container of dissimilar pieces of data). Here's an example that compresses a single file:

```
//: io/GZIPcompress.java
// {Args: GZIPcompress.java}
import java.util.zip.*;
import java.io.*;

public class GZIPcompress {
  public static void main(String[] args)
  throws IOException {
    if(args.length == 0) {
      System.out.println(
        "Usage: \nGZIPcompress file\n" +
        "\tUses GZIP compression to compress " +
        "the file to test.gz");
      System.exit(1);
    }
    BufferedReader in = new BufferedReader(
      new FileReader(args[0]));
    BufferedOutputStream out = new BufferedOutputStream(
      new GZIPOutputStream(
```

```
        new FileOutputStream("test.gz")));
    System.out.println("Writing file");
    int c;
    while((c = in.read()) != -1)
      out.write(c);
    in.close();
    out.close();
    System.out.println("Reading file");
    BufferedReader in2 = new BufferedReader(
      new InputStreamReader(new GZIPInputStream(
        new FileInputStream("test.gz"))));
    String s;
    while((s = in2.readLine()) != null)
      System.out.println(s);
  }
} /* (Execute to see output) *///:~
```

The use of the compression classes is straightforward; you simply wrap your output stream in a **GZIPOutputStream** or **ZipOutputStream**, and your input stream in a **GZIPInputStream** or **ZipInputStream**. All else is ordinary I/O reading and writing. This is an example of mixing the **char**-oriented streams with the byte-oriented streams; **in** uses the **Reader** classes, whereas **GZIPOutputStream**'s constructor can accept only an **OutputStream** object, not a **Writer** object. When the file is opened, the **GZIPInputStream** is converted to a **Reader**.

Multifile storage with Zip

The library that supports the Zip format is more extensive. With it you can easily store multiple files, and there's even a separate class to make the process of reading a Zip file easy. The library uses the standard Zip format so that it works seamlessly with all the Zip tools currently downloadable on the Internet. The following example has the same form as the previous example, but it handles as many command-line arguments as you want. In addition, it shows the use of the **Checksum** classes to calculate and verify the checksum for the file. There are two **Checksum** types: **Adler32** (which is faster) and **CRC32** (which is slower but slightly more accurate).

```
//: io/ZipCompress.java
// Uses Zip compression to compress any
// number of files given on the command line.
// {Args: ZipCompress.java}
import java.util.zip.*;
```

```
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ZipCompress {
  public static void main(String[] args)
  throws IOException {
    FileOutputStream f = new FileOutputStream("test.zip");
    CheckedOutputStream csum =
      new CheckedOutputStream(f, new Adler32());
     ZipOutputStream zos = new ZipOutputStream(csum);
     BufferedOutputStream out =
      new BufferedOutputStream(zos);
    zos.setComment("A test of Java Zipping");
    // No corresponding getComment(), though.
    for(String arg : args) {
      print("Writing file " + arg);
      BufferedReader in =
        new BufferedReader(new FileReader(arg));
      zos.putNextEntry(new ZipEntry(arg));
      int c;
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
      out.flush();
    }
    out.close();
    // Checksum valid only after the file has been closed!
    print("Checksum: " + csum.getChecksum().getValue());
    // Now extract the files:
    print("Reading file");
    FileInputStream fi = new FileInputStream("test.zip");
    CheckedInputStream csumi =
      new CheckedInputStream(fi, new Adler32());
    ZipInputStream in2 = new ZipInputStream(csumi);
    BufferedInputStream bis = new BufferedInputStream(in2);
    ZipEntry ze;
    while((ze = in2.getNextEntry()) != null) {
      print("Reading file " + ze);
      int x;
      while((x = bis.read()) != -1)
        System.out.write(x);
    }
    if(args.length == 1)
```

```
    print("Checksum: " + csumi.getChecksum().getValue());
    bis.close();
    // Alternative way to open and read Zip files:
    ZipFile zf = new ZipFile("test.zip");
    Enumeration e = zf.entries();
    while(e.hasMoreElements()) {
      ZipEntry ze2 = (ZipEntry)e.nextElement();
      print("File: " + ze2);
      // ... and extract the data as before
    }
    /* if(args.length == 1) */
  }
} /* (Execute to see output) *///:~
```

For each file to add to the archive, you must call **putNextEntry( )** and pass it a **ZipEntry** object. The **ZipEntry** object contains an extensive interface that allows you to get and set all the data available on that particular entry in your Zip file: name, compressed and uncompressed sizes, date, CRC checksum, extra field data, comment, compression method, and whether it's a directory entry. However, even though the Zip format has a way to set a password, this is not supported in Java's Zip library. And although **CheckedInputStream** and **CheckedOutputStream** support both **Adler32** and **CRC32** checksums, the **ZipEntry** class supports only an interface for CRC. This is a restriction of the underlying Zip format, but it might limit you from using the faster **Adler32**.

To extract files, **ZipInputStream** has a **getNextEntry( )** method that returns the next **ZipEntry** if there is one. As a more succinct alternative, you can read the file using a **ZipFile** object, which has a method **entries( )** to return an **Enumeration** to the **ZipEntries**.

In order to read the checksum, you must somehow have access to the associated **Checksum** object. Here, a reference to the **CheckedOutputStream** and **CheckedInputStream** objects is retained, but you could also just hold on to a reference to the **Checksum** object.

A baffling method in Zip streams is **setComment( )**. As shown in **ZipCompress.java**, you can set a comment when you're writing a file, but there's no way to recover the comment in the **ZipInputStream**. Comments appear to be supported fully on an entry-by-entry basis only via **ZipEntry**.

Of course, you are not limited to files when using the **GZIP** or **Zip** libraries—you can compress anything, including data to be sent through a network connection.

Java ARchives (JARs)

The Zip format is also used in the JAR (Java ARchive) file format, which is a way to collect a group of files into a single compressed file, just like Zip. However, like everything else in Java, JAR files are cross-platform, so you don't need to worry about platform issues. You can also include audio and image files as well as class files.

JAR files are particularly helpful when you deal with the Internet. Before JAR files, your Web browser would have to make repeated requests of a Web server in order to download all the files that made up an applet. In addition, each of these files was uncompressed. By combining all of the files for a particular applet into a single JAR file, only one server request is necessary and the transfer is faster because of compression. And each entry in a JAR file can be digitally signed for security.

A JAR file consists of a single file containing a collection of zipped files along with a "manifest" that describes them. (You can create your own manifest file; otherwise, the **jar** program will do it for you.) You can find out more about JAR manifests in the JDK documentation.

The **jar** utility that comes with Sun's JDK automatically compresses the files of your choice. You invoke it on the command line:

```
jar [options] destination [manifest] inputfile(s)
```

The options are simply a collection of letters (no hyphen or any other indicator is necessary). Unix/Linux users will note the similarity to the **tar** options. These are:

c	Creates a new or empty archive.
t	Lists the table of contents.
x	Extracts all files.
x file	Extracts the named file.
f	Says, "I'm going to give you the name of the file." If you don't use this, **jar** assumes that its input will come from standard input, or, if it is creating a file, its output will go to

	standard output.
m	Says that the first argument will be the name of the user-created manifest file.
v	Generates verbose output describing what **jar** is doing.
0	Only stores the files; doesn't compress the files (use to create a JAR file that you can put in your classpath).
M	Doesn't automatically create a manifest file.

If a subdirectory is included in the files to be put into the JAR file, that subdirectory is automatically added, including all of its subdirectories, etc. Path information is also preserved.

Here are some typical ways to invoke **jar**. The following command creates a JAR file called **myJarFile.jar** that contains all of the class files in the current directory, along with an automatically generated manifest file:

```
jar cf myJarFile.jar *.class
```

The next command is like the previous example, but it adds a user-created manifest file called **myManifestFile.mf**:

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

This produces a table of contents of the files in **myJarFile.jar**:

```
jar tf myJarFile.jar
```

This adds the "verbose" flag to give more detailed information about the files in **myJarFile.jar**:

```
jar tvf myJarFile.jar
```

Assuming **audio**, **classes**, and **image** are subdirectories, this combines all of the subdirectories into the file **myApp.jar**. The "verbose" flag is also included to give extra feedback while the **jar** program is working:

```
jar cvf myApp.jar audio classes image
```

If you create a JAR file using the **0** (zero) option, that file can be placed in your CLASSPATH:

```
CLASSPATH="lib1.jar;lib2.jar;"
```

Then Java can search **lib1.jar** and **lib2.jar** for class files.

The **jar** tool isn't as general-purpose as a **Zip** utility. For example, you can't add or update files to an existing JAR file; you can create JAR files only from scratch. Also, you can't move files into a JAR file, erasing them as they are moved. However, a JAR file created on one platform will be transparently readable by the **jar** tool on any other platform (a problem that sometimes plagues **Zip** utilities).

As you will see in the *Graphical User Interfaces* chapter, JAR files are also used to package JavaBeans.

Object serialization

When you create an object, it exists for as long as you need it, but under no circumstances does it exist when the program terminates. While this makes sense at first, there are situations in which it would be incredibly useful if an object could exist and hold its information even while the program wasn't running. Then, the next time you started the program, the object would be there and it would have the same information it had the previous time the program was running. Of course, you can get a similar effect by writing the information to a file or to a database, but in the spirit of making everything an object, it would be quite convenient to declare an object to be "persistent," and have all the details taken care of for you.

Java's *object serialization* allows you to take any object that implements the **Serializable** interface and turn it into a sequence of bytes that can later be fully restored to regenerate the original object. This is even true across a network, which means that the serialization mechanism automatically compensates for differences in operating systems. That is, you can create an object on a Windows machine, serialize it, and send it across the network to a Unix machine, where it will be correctly reconstructed. You don't have to worry about the data representations on the different machines, the byte ordering, or any other details.

By itself, object serialization is interesting because it allows you to implement *lightweight persistence*. Persistence means that an object's lifetime is not determined by whether a program is executing; the object lives *in between* invocations of the program. By taking a serializable object and writing it to disk, then restoring that object when the program is reinvoked, you're able to produce the effect of persistence. The reason it's called "lightweight" is that you can't simply define an object using some kind of "persistent" keyword and let the system take care of the details (perhaps this will happen in the

future). Instead, you must explicitly serialize and deserialize the objects in your program. If you need a more serious persistence mechanism, consider a tool like Hibernate (*http://hibernate.sourceforge.net*). For details, see *Thinking in Enterprise Java*, downloadable from *www.MindView.net*.

Object serialization was added to the language to support two major features. Java's *Remote Method Invocation* (RMI) allows objects that live on other machines to behave as if they live on your machine. When messages are sent to remote objects, object serialization is necessary to transport the arguments and return values. RMI is discussed in *Thinking in Enterprise Java*.

Object serialization is also necessary for JavaBeans, described in the *Graphical User Interfaces* chapter. When a Bean is used, its state information is generally configured at design time. This state information must be stored and later recovered when the program is started; object serialization performs this task.

Serializing an object is quite simple as long as the object implements the **Serializable** interface (this is a tagging interface and has no methods). When serialization was added to the language, many standard library classes were changed to make them serializable, including all of the wrappers for the primitive types, all of the container classes, and many others. Even **Class** objects can be serialized.

To serialize an object, you create some sort of **OutputStream** object and then wrap it inside an **ObjectOutputStream** object. At this point you need only call **writeObject( )**, and your object is serialized and sent to the **OutputStream** (object serialization is byte-oriented, and thus uses the **InputStream** and **OutputStream** hierarchies). To reverse the process, you wrap an **InputStream** inside an **ObjectInputStream** and call **readObject( )**. What comes back is, as usual, a reference to an upcast **Object**, so you must downcast to set things straight.

A particularly clever aspect of object serialization is that it not only saves an image of your object, but it also follows all the references contained in your object and saves *those* objects, and follows all the references in each of those objects, etc. This is sometimes referred to as the "web of objects" that a single object can be connected to, and it includes arrays of references to objects as well as member objects. If you had to maintain your own object serialization scheme, maintaining the code to follow all these links could be mind-boggling. However, Java object serialization seems to pull it off flawlessly, no

doubt using an optimized algorithm that traverses the web of objects. The following example tests the serialization mechanism by making a "worm" of linked objects, each of which has a link to the next segment in the worm as well as an array of references to objects of a different class, **Data**:

```
//: io/Worm.java
// Demonstrates object serialization.
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Data implements Serializable {
  private int n;
  public Data(int n) { this.n = n; }
  public String toString() { return Integer.toString(n); }
}

public class Worm implements Serializable {
  private static Random rand = new Random(47);
  private Data[] d = {
    new Data(rand.nextInt(10)),
    new Data(rand.nextInt(10)),
    new Data(rand.nextInt(10))
  };
  private Worm next;
  private char c;
  // Value of i == number of segments
  public Worm(int i, char x) {
    print("Worm constructor: " + i);
    c = x;
    if(--i > 0)
      next = new Worm(i, (char)(x + 1));
  }
  public Worm() {
    print("Default constructor");
  }
  public String toString() {
    StringBuilder result = new StringBuilder(":");
    result.append(c);
    result.append("(");
    for(Data dat : d)
      result.append(dat);
    result.append(")");
    if(next != null)
```

```
      result.append(next);
    return result.toString();
  }
  public static void main(String[] args)
  throws ClassNotFoundException, IOException {
    Worm w = new Worm(6, 'a');
    print("w = " + w);
    ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("worm.out"));
    out.writeObject("Worm storage\n");
    out.writeObject(w);
    out.close(); // Also flushes output
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("worm.out"));
    String s = (String)in.readObject();
    Worm w2 = (Worm)in.readObject();
    print(s + "w2 = " + w2);
    ByteArrayOutputStream bout =
      new ByteArrayOutputStream();
    ObjectOutputStream out2 = new ObjectOutputStream(bout);
    out2.writeObject("Worm storage\n");
    out2.writeObject(w);
    out2.flush();
    ObjectInputStream in2 = new ObjectInputStream(
      new ByteArrayInputStream(bout.toByteArray()));
    s = (String)in2.readObject();
    Worm w3 = (Worm)in2.readObject();
    print(s + "w3 = " + w3);
  }
} /* Output:
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
*///:~
```

To make things interesting, the array of **Data** objects inside **Worm** are initialized with random numbers. (This way, you don't suspect the compiler

of keeping some kind of meta-information.) Each **Worm** segment is labeled with a **char** that's automatically generated in the process of recursively generating the linked list of **Worm**s. When you create a **Worm**, you tell the constructor how long you want it to be. To make the **next** reference, it calls the **Worm** constructor with a length of one less, etc. The final **next** reference is left as **null**, indicating the end of the **Worm**.

The point of all this was to make something reasonably complex that couldn't easily be serialized. The act of serializing, however, is quite simple. Once the **ObjectOutputStream** is created from some other stream, **writeObject( )** serializes the object. Notice the call to **writeObject( )** for a **String**, as well. You can also write all the primitive data types using the same methods as **DataOutputStream** (they share the same interface).

There are two separate code sections that look similar. The first writes and reads a file, and the second, for variety, writes and reads a **ByteArray**. You can read and write an object using serialization to any **DataInputStream** or **DataOutputStream**, including, as you can see in *Thinking in Enterprise Java*, a network.

You can see from the output that the deserialized object really does contain all of the links that were in the original object.

Note that no constructor, not even the default constructor, is called in the process of deserializing a **Serializable** object. The entire object is restored by recovering data from the **InputStream**.

Exercise 27: (1) Create a **Serializable** class containing a reference to an object of a second **Serializable** class. Create an instance of your class, serialize it to disk, then restore it and verify that the process worked correctly.

Finding the class

You might wonder what's necessary for an object to be recovered from its serialized state. For example, suppose you serialize an object and send it as a file or through a network to another machine. Could a program on the other machine reconstruct the object using only the contents of the file?

The best way to answer this question is (as usual) by performing an experiment. The following file goes in the subdirectory for this chapter:

```
//: io/Alien.java
// A serializable class.
```

```
import java.io.*;
public class Alien implements Serializable {} ///:~
```

The file that creates and serializes an **Alien** object goes in the same directory:

```
//: io/FreezeAlien.java
// Create a serialized output file.
import java.io.*;

public class FreezeAlien {
  public static void main(String[] args) throws Exception {
    ObjectOutput out = new ObjectOutputStream(
      new FileOutputStream("X.file"));
    Alien quellek = new Alien();
    out.writeObject(quellek);
  }
} ///:~
```

Rather than catching and handling exceptions, this program takes the quick-and-dirty approach of passing the exceptions out of **main( )**, so they'll be reported on the console.

Once the program is compiled and run, it produces a file called **X.file** in the **io** directory. The following code is in a subdirectory called **xfiles**:

```
//: io/xfiles/ThawAlien.java
// Try to recover a serialized file without the
// class of object that's stored in that file.
// {RunByHand}
import java.io.*;

public class ThawAlien {
  public static void main(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(new File("..", "X.file")));
    Object mystery = in.readObject();
    System.out.println(mystery.getClass());
  }
} /* Output:
class Alien
*///:~
```

Even opening the file and reading in the object **mystery** requires the **Class** object for **Alien**; the JVM cannot find **Alien.class** (unless it happens to be in the classpath, which it shouldn't be in this example). You'll get a

ClassNotFoundException. (Once again, all evidence of alien life vanishes before proof of its existence can be verified!) The JVM must be able to find the associated **.class** file.

Controlling serialization

As you can see, the default serialization mechanism is trivial to use. But what if you have special needs? Perhaps you have special security issues and you don't want to serialize portions of your object, or perhaps it just doesn't make sense for one subobject to be serialized if that part needs to be created anew when the object is recovered.

You can control the process of serialization by implementing the **Externalizable** interface instead of the **Serializable** interface. The **Externalizable** interface extends the **Serializable** interface and adds two methods, **writeExternal( )** and **readExternal( )**, that are automatically called for your object during serialization and deserialization so that you can perform your special operations.

The following example shows simple implementations of the **Externalizable** interface methods. Note that **Blip1** and **Blip2** are nearly identical except for a subtle difference (see if you can discover it by looking at the code):

```
//: io/Blips.java
// Simple use of Externalizable & a pitfall.
import java.io.*;
import static net.mindview.util.Print.*;

class Blip1 implements Externalizable {
  public Blip1() {
    print("Blip1 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    print("Blip1.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    print("Blip1.readExternal");
  }
}
```

```
class Blip2 implements Externalizable {
  Blip2() {
    print("Blip2 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    print("Blip2.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    print("Blip2.readExternal");
  }
}

public class Blips {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    print("Constructing objects:");
    Blip1 b1 = new Blip1();
    Blip2 b2 = new Blip2();
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Blips.out"));
    print("Saving objects:");
    o.writeObject(b1);
    o.writeObject(b2);
    o.close();
    // Now get them back:
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Blips.out"));
    print("Recovering b1:");
    b1 = (Blip1)in.readObject();
    // OOPS! Throws an exception:
//! print("Recovering b2:");
//! b2 = (Blip2)in.readObject();
  }
} /* Output:
Constructing objects:
Blip1 Constructor
Blip2 Constructor
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
```

```
Blip1.readExternal
*///:~
```

The reason that the **Blip2** object is not recovered is that trying to do so causes an exception. Can you see the difference between **Blip1** and **Blip2**? The constructor for **Blip1** is **public**, while the constructor for **Blip2** is not, and that causes the exception upon recovery. Try making **Blip2**'s constructor **public** and removing the **//!** comments to see the correct results.

When **b1** is recovered, the **Blip1** default constructor is called. This is different from recovering a **Serializable** object, in which the object is constructed entirely from its stored bits, with no constructor calls. With an **Externalizable** object, all the normal default construction behavior occurs (including the initializations at the point of field definition), and *then* **readExternal( )** is called. You need to be aware of this—in particular, the fact that all the default construction always takes place—to produce the correct behavior in your **Externalizable** objects.

Here's an example that shows what you must do to fully store and retrieve an **Externalizable** object:

```
//: io/Blip3.java
// Reconstructing an externalizable object.
import java.io.*;
import static net.mindview.util.Print.*;

public class Blip3 implements Externalizable {
  private int i;
  private String s; // No initialization
  public Blip3() {
    print("Blip3 Constructor");
    // s, i not initialized
  }
  public Blip3(String x, int a) {
    print("Blip3(String x, int a)");
    s = x;
    i = a;
    // s & i initialized only in non-default constructor.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
  throws IOException {
    print("Blip3.writeExternal");
    // You must do this:
```

```
    out.writeObject(s);
    out.writeInt(i);
  }
  public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    print("Blip3.readExternal");
    // You must do this:
    s = (String)in.readObject();
    i = in.readInt();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    print("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    print(b3);
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Blip3.out"));
    print("Saving object:");
    o.writeObject(b3);
    o.close();
    // Now get it back:
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Blip3.out"));
    print("Recovering b3:");
    b3 = (Blip3)in.readObject();
    print(b3);
  }
} /* Output:
Constructing objects:
Blip3(String x, int a)
A String 47
Saving object:
Blip3.writeExternal
Recovering b3:
Blip3 Constructor
Blip3.readExternal
A String 47
*///:~
```

The fields **s** and **i** are initialized only in the second constructor, but not in the default constructor. This means that if you don't initialize **s** and **i** in **readExternal( )**, **s** will be **null** and **i** will be zero (since the storage for the object gets wiped to zero in the first step of object creation). If you comment out the two lines of code following the phrases "You must do this:" and run

the program, you'll see that when the object is recovered, **s** is **null** and **i** is zero.

If you are inheriting from an **Externalizable** object, you'll typically call the base-class versions of **writeExternal( )** and **readExternal( )** to provide proper storage and retrieval of the base-class components.

So to make things work correctly, you must not only write the important data from the object during the **writeExternal( )** method (there is no default behavior that writes any of the member objects for an **Externalizable** object), but you must also recover that data in the **readExternal( )** method. This can be a bit confusing at first because the default construction behavior for an **Externalizable** object can make it seem like some kind of storage and retrieval takes place automatically. It does not.

Exercise 28: (2) In **Blips.java**, copy the file and rename it to **BlipCheck.java** and rename the class **Blip2** to **BlipCheck** (making it **public** and removing the public scope from the class **Blips** in the process). Remove the **//!** marks in the file and execute the program, including the offending lines. Next, comment out the default constructor for **BlipCheck**. Run it and explain why it works. Note that after compiling, you must execute the program with "**java Blips**" because the **main( )** method is still in the class **Blips**.

Exercise 29: (2) In **Blip3.java**, comment out the two lines after the phrases "You must do this:" and run the program. Explain the result and why it differs from when the two lines are in the program.

The **transient** keyword

When you're controlling serialization, there might be a particular subobject that you don't want Java's serialization mechanism to automatically save and restore. This is commonly the case if that subobject represents sensitive information that you don't want to serialize, such as a password. Even if that information is **private** in the object, once it has been serialized, it's possible for someone to access it by reading a file or intercepting a network transmission.

One way to prevent sensitive parts of your object from being serialized is to implement your class as **Externalizable**, as shown previously. Then nothing is automatically serialized, and you can explicitly serialize only the necessary parts inside **writeExternal( )**.

If you're working with a **Serializable** object, however, all serialization happens automatically. To control this, you can turn off serialization on a field-by-field basis using the **transient** keyword, which says, "Don't bother saving or restoring this—I'll take care of it."

For example, consider a **Logon** object that keeps information about a particular login session. Suppose that, once you verify the login, you want to store the data, but without the password. The easiest way to do this is by implementing **Serializable** and marking the **password** field as **transient**. Here's what it looks like:

```
//: io/Logon.java
// Demonstrates the "transient" keyword.
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Logon implements Serializable {
  private Date date = new Date();
  private String username;
  private transient String password;
  public Logon(String name, String pwd) {
    username = name;
    password = pwd;
  }
  public String toString() {
    return "logon info: \n   username: " + username +
      "\n   date: " + date + "\n   password: " + password;
  }
  public static void main(String[] args) throws Exception {
    Logon a = new Logon("Hulk", "myLittlePony");
    print("logon a = " + a);
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Logon.out"));
    o.writeObject(a);
    o.close();
    TimeUnit.SECONDS.sleep(1); // Delay
    // Now get them back:
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Logon.out"));
    print("Recovering object at " + new Date());
    a = (Logon)in.readObject();
    print("logon a = " + a);
```

```
  }
} /* Output: (Sample)
logon a = logon info:
   username: Hulk
   date: Sat Nov 19 15:03:26 MST 2005
   password: myLittlePony
Recovering object at Sat Nov 19 15:03:28 MST 2005
logon a = logon info:
   username: Hulk
   date: Sat Nov 19 15:03:26 MST 2005
   password: null
*///:~
```

You can see that the **date** and **username** fields are ordinary (not **transient**), and thus are automatically serialized. However, the **password** is **transient**, so it is not stored to disk; also, the serialization mechanism makes no attempt to recover it. When the object is recovered, the **password** field is **null**. Note that while **toString( )** assembles a **String** object using the overloaded '+' operator, a **null** reference is automatically converted to the string "null."

You can also see that the **date** field is stored to and recovered from disk and not generated anew.

Since **Externalizable** objects do not store any of their fields by default, the **transient** keyword is for use with **Serializable** objects only.

An alternative to **Externalizable**

If you're not keen on implementing the **Externalizable** interface, there's another approach. You can implement the **Serializable** interface and *add* (notice I say "add" and not "override" or "implement") methods called **writeObject( )** and **readObject( )** that will automatically be called when the object is serialized and deserialized, respectively. That is, if you provide these two methods, they will be used instead of the default serialization.

The methods must have these exact signatures:

```
private void writeObject(ObjectOutputStream stream)
throws IOException;

private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException
```

From a design standpoint, things get really weird here. First of all, you might think that because these methods are not part of a base class or the **Serializable** interface, they ought to be defined in their own interface(s). But notice that they are defined as **private**, which means they are to be called only by other members of this class. However, you don't actually call them from other members of this class, but instead the **writeObject( )** and **readObject( )** methods of the **ObjectOutputStream** and **ObjectInputStream** objects call your object's **writeObject( )** and **readObject( )** methods. (Notice my tremendous restraint in not launching into a long diatribe about using the same method names here. In a word: confusing.) You might wonder how the **ObjectOutputStream** and **ObjectInputStream** objects have access to **private** methods of your class. We can only assume that this is part of the serialization magic.[6]

Anything defined in an interface is automatically **public**, so if **writeObject( )** and **readObject( )** must be **private**, then they can't be part of an interface. Since you must follow the signatures exactly, the effect is the same as if you're implementing an interface.

It would appear that when you call **ObjectOutputStream.writeObject( )**, the **Serializable** object that you pass it to is interrogated (using reflection, no doubt) to see if it implements its own **writeObject( )**. If so, the normal serialization process is skipped and the custom **writeObject( )** is called. The same situation exists for **readObject( )**.

There's one other twist. Inside your **writeObject( )**, you can choose to perform the default **writeObject( )** action by calling **defaultWriteObject( )**. Likewise, inside **readObject( )** you can call **defaultReadObject( )**. Here is a simple example that demonstrates how you can control the storage and retrieval of a **Serializable** object:

```
//: io/SerialCtl.java
// Controlling serialization by adding your own
// writeObject() and readObject() methods.
import java.io.*;

public class SerialCtl implements Serializable {
  private String a;
```

[6] The section "Interfaces and type information" at the end of the *Type Information* chapter shows how it's possible to access **private** methods from outside of the class.

```
  private transient String b;
  public SerialCtl(String aa, String bb) {
    a = "Not Transient: " + aa;
    b = "Transient: " + bb;
  }
  public String toString() { return a + "\n" + b; }
  private void writeObject(ObjectOutputStream stream)
  throws IOException {
    stream.defaultWriteObject();
    stream.writeObject(b);
  }
  private void readObject(ObjectInputStream stream)
  throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    b = (String)stream.readObject();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    SerialCtl sc = new SerialCtl("Test1", "Test2");
    System.out.println("Before:\n" + sc);
    ByteArrayOutputStream buf= new ByteArrayOutputStream();
    ObjectOutputStream o = new ObjectOutputStream(buf);
    o.writeObject(sc);
    // Now get it back:
    ObjectInputStream in = new ObjectInputStream(
      new ByteArrayInputStream(buf.toByteArray()));
    SerialCtl sc2 = (SerialCtl)in.readObject();
    System.out.println("After:\n" + sc2);
  }
} /* Output:
Before:
Not Transient: Test1
Transient: Test2
After:
Not Transient: Test1
Transient: Test2
*///:~
```

In this example, one **String** field is ordinary and the other is **transient**, to prove that the non-**transient** field is saved by the **defaultWriteObject( )** method and the **transient** field is saved and restored explicitly. The fields are initialized inside the constructor rather than at the point of definition to prove that they are not being initialized by some automatic mechanism during deserialization.

If you use the default mechanism to write the non-**transient** parts of your object, you must call **defaultWriteObject( )** as the first operation in **writeObject( )**, and **defaultReadObject( )** as the first operation in **readObject( )**. These are strange method calls. It would appear, for example, that you are calling **defaultWriteObject( )** for an **ObjectOutputStream** and passing it no arguments, and yet it somehow turns around and knows the reference to your object and how to write all the non-**transient** parts. Spooky.

The storage and retrieval of the **transient** objects uses more familiar code. And yet, think about what happens here. In **main( )**, a **SerialCtl** object is created, and then it's serialized to an **ObjectOutputStream.** (Notice in this case that a buffer is used instead of a file—it's all the same to the **ObjectOutputStream**.) The serialization occurs in the line:

```
o.writeObject(sc);
```

The **writeObject( )** method must be examining **sc** to see if it has its own **writeObject( )** method. (Not by checking the interface—there isn't one—or the class type, but by actually hunting for the method using reflection.) If it does, it uses that. A similar approach holds true for **readObject( )**. Perhaps this was the only practical way that they could solve the problem, but it's certainly strange.

Versioning

It's possible that you might want to change the version of a serializable class (objects of the original class might be stored in a database, for example). This is supported, but you'll probably do it only in special cases, and it requires an extra depth of understanding that we will not attempt to achieve here. The JDK documents downloadable from *http://java.sun.com* cover this topic quite thoroughly.

You will also notice in the JDK documentation many comments that begin with:

> ***Warning:*** *Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications ...*

This is because the versioning mechanism is too simple to work reliably in all situations, especially with JavaBeans. They're working on a correction for the design, and that's what the warning is about.

Using persistence

It's quite appealing to use serialization technology to store some of the state of your program so that you can easily restore the program to the current state later. But before you can do this, some questions must be answered. What happens if you serialize two objects that both have a reference to a third object? When you restore those two objects from their serialized state, do you get only one occurrence of the third object? What if you serialize your two objects to separate files and deserialize them in different parts of your code?

Here's an example that shows the problem:

```
//: io/MyWorld.java
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class House implements Serializable {}

class Animal implements Serializable {
  private String name;
  private House preferredHouse;
  Animal(String nm, House h) {
    name = nm;
    preferredHouse = h;
  }
  public String toString() {
    return name + "[" + super.toString() +
      "], " + preferredHouse + "\n";
  }
}

public class MyWorld {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    House house = new House();
    List<Animal> animals = new ArrayList<Animal>();
    animals.add(new Animal("Bosco the dog", house));
    animals.add(new Animal("Ralph the hamster", house));
    animals.add(new Animal("Molly the cat", house));
    print("animals: " + animals);
    ByteArrayOutputStream buf1 =
      new ByteArrayOutputStream();
    ObjectOutputStream o1 = new ObjectOutputStream(buf1);
```

```
    o1.writeObject(animals);
    o1.writeObject(animals); // Write a 2nd set
    // Write to a different stream:
    ByteArrayOutputStream buf2 =
      new ByteArrayOutputStream();
    ObjectOutputStream o2 = new ObjectOutputStream(buf2);
    o2.writeObject(animals);
    // Now get them back:
    ObjectInputStream in1 = new ObjectInputStream(
      new ByteArrayInputStream(buf1.toByteArray()));
    ObjectInputStream in2 = new ObjectInputStream(
      new ByteArrayInputStream(buf2.toByteArray()));
    List
      animals1 = (List)in1.readObject(),
      animals2 = (List)in1.readObject(),
      animals3 = (List)in2.readObject();
    print("animals1: " + animals1);
    print("animals2: " + animals2);
    print("animals3: " + animals3);
  }
} /* Output: (Sample)
animals: [Bosco the dog[Animal@addbf1], House@42e816
, Ralph the hamster[Animal@9304b1], House@42e816
, Molly the cat[Animal@190d11], House@42e816
]
animals1: [Bosco the dog[Animal@de6f34], House@156ee8e
, Ralph the hamster[Animal@47b480], House@156ee8e
, Molly the cat[Animal@19b49e6], House@156ee8e
]
animals2: [Bosco the dog[Animal@de6f34], House@156ee8e
, Ralph the hamster[Animal@47b480], House@156ee8e
, Molly the cat[Animal@19b49e6], House@156ee8e
]
animals3: [Bosco the dog[Animal@10d448], House@e0e1c6
, Ralph the hamster[Animal@6ca1c], House@e0e1c6
, Molly the cat[Animal@1bf216a], House@e0e1c6
]
*///:~
```

One thing that's interesting here is that it's possible to use object serialization to and from a **byte** array as a way of doing a "deep copy" of any object that's **Serializable.** (A deep copy means that you're duplicating the entire web of objects, rather than just the basic object and its references.) Object copying is covered in depth in the online supplements for this book.

Animal objects contain fields of type **House**. In **main( )**, a **List** of these **Animal**s is created and it is serialized twice to one stream and then again to a separate stream. When these are deserialized and printed, you see the output shown for one run (the objects will be in different memory locations each run).

Of course, you expect that the deserialized objects have different addresses from their originals. But notice that in **animals1** and **animals2**, the same addresses appear, including the references to the **House** object that both share. On the other hand, when **animals3** is recovered, the system has no way of knowing that the objects in this other stream are aliases of the objects in the first stream, so it makes a completely different web of objects.

As long as you're serializing everything to a single stream, you'll recover the same web of objects that you wrote, with no accidental duplication of objects. Of course, you can change the state of your objects in between the time you write the first and the last, but that's your responsibility; the objects will be written in whatever state they are in (and with whatever connections they have to other objects) at the time you serialize them.

The safest thing to do if you want to save the state of a system is to serialize as an "atomic" operation. If you serialize some things, do some other work, and serialize some more, etc., then you will not be storing the system safely. Instead, put all the objects that comprise the state of your system in a single container and simply write that container out in one operation. Then you can restore it with a single method call as well.

The following example is an imaginary computer-aided design (CAD) system that demonstrates the approach. In addition, it throws in the issue of **static** fields; if you look at the JDK documentation, you'll see that **Class** is **Serializable**, so it should be easy to store the **static** fields by simply serializing the **Class** object. That seems like a sensible approach, anyway.

```
//: io/StoreCADState.java
// Saving the state of a pretend CAD system.
import java.io.*;
import java.util.*;

abstract class Shape implements Serializable {
  public static final int RED = 1, BLUE = 2, GREEN = 3;
  private int xPos, yPos, dimension;
  private static Random rand = new Random(47);
```

```
  private static int counter = 0;
  public abstract void setColor(int newColor);
  public abstract int getColor();
  public Shape(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimension = dim;
  }
  public String toString() {
    return getClass() +
      "color[" + getColor() + "] xPos[" + xPos +
      "] yPos[" + yPos + "] dim[" + dimension + "]\n";
  }
  public static Shape randomFactory() {
    int xVal = rand.nextInt(100);
    int yVal = rand.nextInt(100);
    int dim = rand.nextInt(100);
    switch(counter++ % 3) {
      default:
      case 0: return new Circle(xVal, yVal, dim);
      case 1: return new Square(xVal, yVal, dim);
      case 2: return new Line(xVal, yVal, dim);
    }
  }
}

class Circle extends Shape {
  private static int color = RED;
  public Circle(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) { color = newColor; }
  public int getColor() { return color; }
}

class Square extends Shape {
  private static int color;
  public Square(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
    color = RED;
  }
  public void setColor(int newColor) { color = newColor; }
  public int getColor() { return color; }
}
```

```
class Line extends Shape {
  private static int color = RED;
  public static void
  serializeStaticState(ObjectOutputStream os)
  throws IOException { os.writeInt(color); }
  public static void
  deserializeStaticState(ObjectInputStream os)
  throws IOException { color = os.readInt(); }
  public Line(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) { color = newColor; }
  public int getColor() { return color; }
}

public class StoreCADState {
  public static void main(String[] args) throws Exception {
    List<Class<? extends Shape>> shapeTypes =
      new ArrayList<Class<? extends Shape>>();
    // Add references to the class objects:
    shapeTypes.add(Circle.class);
    shapeTypes.add(Square.class);
    shapeTypes.add(Line.class);
    List<Shape> shapes = new ArrayList<Shape>();
    // Make some shapes:
    for(int i = 0; i < 10; i++)
      shapes.add(Shape.randomFactory());
    // Set all the static colors to GREEN:
    for(int i = 0; i < 10; i++)
      ((Shape)shapes.get(i)).setColor(Shape.GREEN);
    // Save the state vector:
    ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("CADState.out"));
    out.writeObject(shapeTypes);
    Line.serializeStaticState(out);
    out.writeObject(shapes);
    // Display the shapes:
    System.out.println(shapes);
  }
} /* Output:
[class Circlecolor[3] xPos[58] yPos[55] dim[93]
, class Squarecolor[3] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
```

```
, class Circlecolor[3] xPos[7] yPos[88] dim[28]
, class Squarecolor[3] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
, class Circlecolor[3] xPos[20] yPos[58] dim[16]
, class Squarecolor[3] xPos[40] yPos[11] dim[22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[3] xPos[75] yPos[10] dim[42]
]
*///:~
```

The **Shape** class **implements Serializable**, so anything that is inherited from **Shape** is automatically **Serializable** as well. Each **Shape** contains data, and each derived **Shape** class contains a **static** field that determines the color of all of those types of **Shape**s. (Placing a **static** field in the base class would result in only one field, since **static** fields are not duplicated in derived classes.) Methods in the base class can be overridden to set the color for the various types (**static** methods are not dynamically bound, so these are normal methods). The **randomFactory( )** method creates a different **Shape** each time you call it, using random values for the **Shape** data.

Circle and **Square** are straightforward extensions of **Shape**; the only difference is that **Circle** initializes **color** at the point of definition and **Square** initializes it in the constructor. We'll leave the discussion of **Line** for later.

In **main( )**, one **ArrayList** is used to hold the **Class** objects and the other to hold the shapes.

Recovering the objects is fairly straightforward:

```
//: io/RecoverCADState.java
// Restoring the state of the pretend CAD system.
// {RunFirst: StoreCADState}
import java.io.*;
import java.util.*;

public class RecoverCADState {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("CADState.out"));
    // Read in the same order they were written:
    List<Class<? extends Shape>> shapeTypes =
      (List<Class<? extends Shape>>)in.readObject();
```

```
    Line.deserializeStaticState(in);
    List<Shape> shapes = (List<Shape>)in.readObject();
    System.out.println(shapes);
  }
} /* Output:
[class Circlecolor[1] xPos[58] yPos[55] dim[93]
, class Squarecolor[0] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
, class Circlecolor[1] xPos[7] yPos[88] dim[28]
, class Squarecolor[0] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
, class Circlecolor[1] xPos[20] yPos[58] dim[16]
, class Squarecolor[0] xPos[40] yPos[11] dim[22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[1] xPos[75] yPos[10] dim[42]
]
*///:~
```

You can see that the values of **xPos**, **yPos**, and **dim** were all stored and recovered successfully, but there's something wrong with the retrieval of the **static** information. It's all "3" going in, but it doesn't come out that way. **Circle**s have a value of 1 (**RED**, which is the definition), and **Square**s have a value of 0 (remember, they are initialized in the constructor). It's as if the **static**s didn't get serialized at all! That's right—even though class **Class** is **Serializable**, it doesn't do what you expect. So if you want to serialize **static**s, you must do it yourself.

This is what the **serializeStaticState( )** and **deserializeStaticState( ) static** methods in **Line** are for. You can see that they are explicitly called as part of the storage and retrieval process. (Note that the order of writing to the serialize file and reading back from it must be maintained.) Thus to make these programs run correctly, you must:

1. Add a **serializeStaticState( )** and **deserializeStaticState( )** to the shapes.
2. Remove the **ArrayList shapeTypes** and all code related to it.
3. Add calls to the new serialize and deserialize **static** methods in the shapes.

Another issue you might have to think about is security, since serialization also saves **private** data. If you have a security issue, those fields should be

marked as **transient**. But then you have to design a secure way to store that information so that when you do a restore, you can reset those **private** variables.

Exercise 30: (1) Repair the program **CADState.java** as described in the text.

XML

An important limitation of object serialization is that it is a Java-only solution: Only Java programs can deserialize such objects. A more interoperable solution is to convert data to XML format, which allows it to be consumed by a large variety of platforms and languages.

Because of its popularity, there are a confusing number of options for programming with XML, including the **javax.xml.*** libraries distributed with the JDK. I've chosen to use Elliotte Rusty Harold's open-source XOM library (downloads and documentation at *www.xom.nu*) because it seems to be the simplest and most straightforward way to produce and modify XML using Java. In addition, XOM emphasizes XML correctness.

As an example, suppose you have **Person** objects containing first and last names that you'd like to serialize into XML. The following **Person** class has a **getXML( )** method that uses XOM to produce the **Person** data converted to an XML **Element** object, and a constructor that takes an **Element** and extracts the appropriate **Person** data (notice that the XML examples are in their own subdirectory):

```
//: xml/Person.java
// Use the XOM library to write and read XML
// {Requires: nu.xom.Node; You must install
// the XOM library from http://www.xom.nu }
import nu.xom.*;
import java.io.*;
import java.util.*;

public class Person {
  private String first, last;
  public Person(String first, String last) {
    this.first = first;
    this.last = last;
  }
  // Produce an XML Element from this Person object:
```

```
  public Element getXML() {
    Element person = new Element("person");
    Element firstName = new Element("first");
    firstName.appendChild(first);
    Element lastName = new Element("last");
    lastName.appendChild(last);
    person.appendChild(firstName);
    person.appendChild(lastName);
    return person;
  }
  // Constructor to restore a Person from an XML Element:
  public Person(Element person) {
    first= person.getFirstChildElement("first").getValue();
    last = person.getFirstChildElement("last").getValue();
  }
  public String toString() { return first + " " + last; }
  // Make it human-readable:
  public static void
  format(OutputStream os, Document doc) throws Exception {
    Serializer serializer= new Serializer(os,"ISO-8859-1");
    serializer.setIndent(4);
    serializer.setMaxLength(60);
    serializer.write(doc);
    serializer.flush();
  }
  public static void main(String[] args) throws Exception {
    List<Person> people = Arrays.asList(
      new Person("Dr. Bunsen", "Honeydew"),
      new Person("Gonzo", "The Great"),
      new Person("Phillip J.", "Fry"));
    System.out.println(people);
    Element root = new Element("people");
    for(Person p : people)
      root.appendChild(p.getXML());
    Document doc = new Document(root);
    format(System.out, doc);
    format(new BufferedOutputStream(new FileOutputStream(
      "People.xml")), doc);
  }
} /* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
<?xml version="1.0" encoding="ISO-8859-1"?>
<people>
    <person>
```

```
        <first>Dr. Bunsen</first>
        <last>Honeydew</last>
    </person>
    <person>
        <first>Gonzo</first>
        <last>The Great</last>
    </person>
    <person>
        <first>Phillip J.</first>
        <last>Fry</last>
    </person>
</people>
*///:~
```

The XOM methods are fairly self-explanatory and can be found in the XOM documentation.

XOM also contains a **Serializer** class that you can see used in the **format( )** method to turn the XML into a more readable form. If you just call **toXML( )** you'll get everything run together, so the **Serializer** is a convenient tool.

Deserializing **Person** objects from an XML file is also simple:

```
//: xml/People.java
// {Requires: nu.xom.Node; You must install
// the XOM library from http://www.xom.nu }
// {RunFirst: Person}
import nu.xom.*;
import java.util.*;

public class People extends ArrayList<Person> {
  public People(String fileName) throws Exception  {
    Document doc = new Builder().build(fileName);
    Elements elements =
      doc.getRootElement().getChildElements();
    for(int i = 0; i < elements.size(); i++)
      add(new Person(elements.get(i)));
  }
  public static void main(String[] args) throws Exception {
    People p = new People("People.xml");
    System.out.println(p);
  }
} /* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
*///:~
```

The **People** constructor opens and reads a file using XOM's **Builder.build( )** method, and the **getChildElements( )** method produces an **Elements** list (not a standard Java **List**, but an object that only has a **size( )** and **get( )** method—Harold did not want to force people to use Java SE5, but still wanted a type-safe container). Each **Element** in this list represents a **Person** object, so it is handed to the second **Person** constructor. Note that this requires that you know ahead of time the exact structure of your XML file, but this is often true with these kinds of problems. If the structure doesn't match what you expect, XOM will throw an exception. It's also possible for you to write more complex code that will explore the XML document rather than making assumptions about it, for cases when you have less concrete information about the incoming XML structure.

In order to get these examples to compile, you will have to put the JAR files from the XOM distribution into your classpath.

This has only been a brief introduction to XML programming with Java and the XOM library; for more information see *www.xom.nu*.

Exercise 31: (2) Add appropriate address information to **Person.java** and **People.java**.

Exercise 32: (4) Using a **Map<String,Integer>** and the **net.mindview.util.TextFile** utility, write a program that counts the occurrence of words in a file (use **"\\W+"** as the second argument to the **TextFile** constructor). Store the results as an XML file.

Preferences

The *Preferences* API is much closer to persistence than it is to object serialization, because it automatically stores and retrieves your information. However, its use is restricted to small and limited data sets—you can only hold primitives and **Strings**, and the length of each stored **String** can't be longer than 8K (not tiny, but you don't want to build anything serious with it, either). As the name suggests, the Preferences API is designed to store and retrieve user preferences and program-configuration settings.

Preferences are key-value sets (like **Map**s) stored in a hierarchy of nodes. Although the node hierarchy can be used to create complicated structures, it's typical to create a single node named after your class and store the information there. Here's a simple example:

```
//: io/PreferencesDemo.java
import java.util.prefs.*;
import static net.mindview.util.Print.*;

public class PreferencesDemo {
  public static void main(String[] args) throws Exception {
    Preferences prefs = Preferences
      .userNodeForPackage(PreferencesDemo.class);
    prefs.put("Location", "Oz");
    prefs.put("Footwear", "Ruby Slippers");
    prefs.putInt("Companions", 4);
    prefs.putBoolean("Are there witches?", true);
    int usageCount = prefs.getInt("UsageCount", 0);
    usageCount++;
    prefs.putInt("UsageCount", usageCount);
    for(String key : prefs.keys())
      print(key + ": "+ prefs.get(key, null));
    // You must always provide a default value:
    print("How many companions does Dorothy have? " +
      prefs.getInt("Companions", 0));
  }
} /* Output: (Sample)
Location: Oz
Footwear: Ruby Slippers
Companions: 4
Are there witches?: true
UsageCount: 53
How many companions does Dorothy have? 4
*///:~
```

Here, **userNodeForPackage( )** is used, but you could also choose **systemNodeForPackage( )**; the choice is somewhat arbitrary, but the idea is that "user" is for individual user preferences, and "system" is for general installation configuration. Since **main( )** is **static**, **PreferencesDemo.class** is used to identify the node, but inside a non-**static** method, you'll usually use **getClass( )**. You don't need to use the current class as the node identifier, but that's the usual practice.

Once you create the node, it's available for either loading or reading data. This example loads the node with various types of items and then gets the **keys( )**. These come back as a **String[]**, which you might not expect if you're used to the **keys( )** method in the collections library. Notice the second argument to **get( )**. This is the default value that is produced if there isn't any

entry for that key value. While iterating through a set of keys, you always know there's an entry, so using **null** as the default is safe, but normally you'll be fetching a named key, as in:

```
prefs.getInt("Companions", 0));
```

In the normal case, you'll want to provide a reasonable default value. In fact, a typical idiom is seen in the lines:

```
int usageCount = prefs.getInt("UsageCount", 0);
usageCount++;
prefs.putInt("UsageCount", usageCount);
```

This way, the first time you run the program, the **UsageCount** will be zero, but on subsequent invocations it will be nonzero.

When you run **PreferencesDemo.java** you'll see that the **UsageCount** does indeed increment every time you run the program, but where is the data stored? There's no local file that appears after the program is run the first time. The Preferences API uses appropriate system resources to accomplish its task, and these will vary depending on the OS. In Windows, the registry is used (since it's already a hierarchy of nodes with key-value pairs). But the whole point is that the information is magically stored for you so that you don't have to worry about how it works from one system to another.

There's more to the Preferences API than shown here. Consult the JDK documentation, which is fairly understandable, for further details.

Exercise 33: (2) Write a program that displays the current value of a directory called "base directory" and prompts you for a new value. Use the Preferences API to store the value.

Summary

The Java I/O stream library does satisfy the basic requirements: You can perform reading and writing with the console, a file, a block of memory, or even across the Internet. With inheritance, you can create new types of input and output objects. And you can even add a simple extensibility to the kinds of objects a stream will accept by redefining the **toString( )** method that's automatically called when you pass an object to a method that's expecting a **String** (Java's limited "automatic type conversion").

There are questions left unanswered by the documentation and design of the I/O stream library. For example, it would have been nice if you could say that you want an exception thrown if you try to overwrite a file when opening it for output—some programming systems allow you to specify that you want to open an output file, but only if it doesn't already exist. In Java, it appears that you are supposed to use a **File** object to determine whether a file exists, because if you open it as a **FileOutputStream** or **FileWriter**, it will always get overwritten.

The I/O stream library brings up mixed feelings; it does much of the job and it's portable. But if you don't already understand the Decorator design pattern, the design is not intuitive, so there's extra overhead in learning and teaching it. It's also incomplete; for example, I shouldn't have to write utilities like **TextFile** (the new Java SE5 **PrintWriter** is a step in the right direction here, but is only a partial solution). There has been a big improvement in Java SE5: They've finally added the kind of output formatting that virtually every other language has always supported.

Once you *do* understand the Decorator pattern and begin using the library in situations that require its flexibility, you can begin to benefit from this design, at which point its cost in extra lines of code may not bother you as much.

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for sale from *www.MindView.net*.