I/O Fundamentals

The Java language provides a simple model for input and output (I/O). All I/O is performed by writing to and reading from streams of data. The data may exist in a file or an array, be piped from another stream, or even come from a port on another computer. The flexibility of this model makes it a powerful abstraction of any required input and output.

One of the key issues regarding Java I/O is providing file support for all Java-enabled platforms. The Java file I/O classes must restrict themselves to a reasonable, "least common denominator" of file-system functionality. Provided functionality is restricted to only those general features that can be used on any modern platform. For example, you won't find ways to work with OS/2's "extended attributes", or "sync" a file in UNIX.

All classes referred to in this module are located in the java.io package (unless otherwise stated.)

Notes

A few notes on the content and examples in this module:

- This module refers to some methods and classes that are only available in the Java 2 SDK, standard edition v1.2 (formerly known as "JDK 1.2"). Methods and classes such as these are marked "(since 1.2)"
- We discuss the most commonly used methods in these classes, not necessarily all available methods. See the javadocs for a list of all methods in each class.
- All examples will require exception handling to catch `IOException`. This exception handling is omitted in many examples for clarity, and discussed at the end of the module. If an example is a complete class listing, the exception handling will be present.

The File Class

The `File` class is Java's representation of a file or directory path name. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them.

The `File` class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and

determining several common attributes of files and directories.

Creating a File Object

You create a `File` object by passing in a `String` that represents the name of a file, and possibly a `String` or another `File` object. For example,

```
File a = new File("/usr/local/bin/smurf");
```

defines an abstract file name for the `smurf` file in directory `/usr/local/bin`. This is an *absolute* abstract file name. It gives *all* path information necessary to find the file.

You could also create a file object as follows:

```
File b = new File("bin/smurf");
```

This is a *relative* abstract file name, because it leaves out some necessary path information, which will be filled in by the VM. By default, the VM will use the directory in which the application was executed as the "current path". You can override this default behavior by specifying the `user.dir` system property (System Properties are explained later in this module). For example, if the application were executed from directory `/usr/local`, the file could be found.

The above examples use UNIX-like file names. We can create similar `File` objects for Windows files as follows:

```
File c = new File("c:\\windows\\system\\smurf.gif");
File d = new File("system\\smurf.gif");
```

Note the double backslashes. Because the backslash is a Java String escape character, you must type two of them to represent a single, "real" backslash.

The above specifications are not very portable. The problem is that the *direction* of the slashes and the way the "root" of the path is specified is specific for the platform in question. Fortunately, there are several ways to deal with this issue.

First, Java allows *either* type of slash to be used on any platform, and translates it appropriately. This means that you could type

```
File e = new File("c:/windows/system/smurf.gif");
```

and it will find the same file on Windows. However, we still have the "root" of the path as a problem.

The easiest solution to deal with files on multiple platforms is to *always* use

relative path names. A file name like

```
File f = new File("images/smurf.gif");
```

will work on any system.

If full path names (including drive specifications) are required, we will see some methods of obtaining a list of available devices later.

Finally, you can create files by specifying two parameters: the name (String or File) of the *parent directory*, and the simple name of the file in that directory. For example:

```
File g = new File("/windows/system");
File h = new File(g, "smurf.gif");
File i = new File("/windows/system", "smurf.gif");
```

will create objects for `h` and `i` that refer to the same file.

File Attribute Methods

The `File` object has several methods that provide information on the current state of the file.

`boolean canRead()`	Returns `true` if the file is readable
`Boolean canWrite()`	Returns `true` if the file is writeable
`Boolean exists()`	Returns `true` if the file exists
`boolean isAbsolute()`	Returns `true` if the file name is an *absolute* path name
`boolean isDirectory()`	Returns `true` if the file name is a directory
`boolean isFile()`	Returns `true` if the file name is a "normal" file (depends on OS)
`boolean isHidden()` ***(since 1.2)***	Returns `true` if the file is marked "hidden"
`long lastModified()`	Returns a `long` indicating the last time the file was modified
`long length()`	Returns the length of the contents of the file

`boolean setReadOnly()` ***(since 1.2)***	Marks the file read-only (returns `true` if succeeded)
`void setLastModified(long)` ***(since 1.2)***	Explicitly sets the modification time of a file

File Name Methods

The following table shows the methods of the `File` class that relate to getting the file name, or part of it.

Some of the examples in this table use the following declaration:

```
File a = new File("\\windows\system\smurf.gif");
```

`int compareTo(File)` `int compareTo(Object)` ***(both since 1.2)***	Compares the file name to another file name or object, returning an `int` that represents the sorting order
`boolean equals(Object)`	Compares the file names to see if they are equivalent
`File getAbsoluteFile()` ***(since 1.2)***	Gets an abstract file name that represents resolution of the *absolute* file name for this `File`
`String getAbsolutePath()`	Resolves the *absolute* file name for this `File`

`String getCanonicalPath()`	Gets an abstract file name that is *unique* for the current `File`. Note that the actual name is dependent on the file system and is *always* an absolute file name. On Windows, for example, the canonical path is the absolute file name *including* the real case of the file and path names.
	
	`a.getCanonicalPath() ==>`
	`"c:\WINDOWS\SYSTEM\smurf.gif"`

`File getCanonicalFile()` ***(since 1.2)***	Same as `new File(getCanonicalPath())`
`String getName()`	Returns the name for the file without any preceding path information.

	`a.getName() ==> "smurf.gif"`
`String getParent()`	Returns the path to the file name, without the actual file name. `a.getParent() ==> "\windows\system"`
`File getParentFile()` ***(since 1.2)***	Same as `new File(getParent())`

`String getPath()`	returns the path used to construct this object.
	
	`a.getPath() ==>`
	`"\windows\system\smurf.gif"`

`URL toURL()` ***(since 1.2)***	Returns a "`file:`" `URL` object that represents the file.

File System Modification Methods

The following table shows the `File` methods you can use to alter the file system, for example by creating, deleting, and renaming files. The `boolean`results from most of these methods simply answer the question "Did it work?" (If something *really* bad happens, like a security problem or other file-system error, an exception will be thrown.)

`boolean createNewFile()` ***(since 1.2)***	Creates a new file with this abstract file name. Returns `true` if the file was created, `false` if the file already existed.

`File createTempFile(String, String)`
`File createTempFile(String, String, File)`
(both since 1.2)
	Creates a temporary file with the specified prefix and suffix Strings. If no directory `File` is specified, it will be created in the file systems' "temporary" directory.

`boolean delete()`	Deletes the file specified by this file name.
`void deleteOnExit()` ***(since 1.2)***	Sets up processing to delete this file when the VM exits (via `System.exit()` or when only daemon threads are left running.).
`boolean mkdir()`	Creates this directory. All parent directories

	must already exist.
`boolean mkdirs()`	Creates this directory *and any parent directories that do not exist.*
`boolean renameTo(File)`	Renames the file.

Directory List Methods

The following methods allow you to determine information about which roots are available on the file system and which other files reside in a directory.

`String[] list()` `String[] list(FileNameFilter)`	Returns an array of `String`s that represent the names of the files contain within this directory. Returns null if th file is not a directory. If a `FileNameFilter` object is supplie the list will be limited to those that mat the filter.

`File[] listFiles()`
`File[] listFiles(FileFilter)`
`File[] listFiles(FileNameFilter)`
(all three since 1.2)
	Similar to `list()`, but returns an array `File` objects. If a `FileFilter` or `FileNameFilter` is specified, the list v be limited to file names that match the filter.

`File[] listRoots()` ***(since 1.2)***	Returns an array of `File`s that represe the root directories for the current platform. The result is determined by t VM, and will *always* include any physically-present drives on the machi but only include remote roots if they ar somehow mapped to the physical file system. (Using `mount` on UNIX or mapping to a drive letter on Windows, example.)

UNICODE

All text in Java is represented as two-byte UNICODE characters. UNICODE is a

standard that allows characters from character sets throughout the world to be represented in two bytes. (for details see http://www.unicode.org

Characters 0-127 of the UNICODE standard map directly to the ASCII standard. The rest of the character set is composed of "pages" that represent other character sets. There are pages that map to characters from many different languages, "Dingbats" (symbols that can be used as characters), currency symbols, mathematical symbols and many others.

The trick of course, is that each platform has its own *native* character set, which usually has some mapping to the UNICODE standard. Java needs some way to map the *native* character set to UNICODE.

Java Translation

Java's text input and output classes *translate* the native characters to and from UNICODE. For each delivered JDK, there is a "default mapping" that is used for most translations. You also can specify the encoding.

For example, if you are on a machine that uses an ASCII encoding, Java will map the ASCII to UNICODE by padding the ASCII characters with extra 0 bits to create two-byte characters. Other languages have a more complex mapping.

When reading files, Java translates from native format to UNICODE. When writing files, Java translates from UNICODE to the native format.

Java Internal Formats

Many people raise concerns about Java efficiency due to the two-byte character representation of UNICODE.

Java uses the UTF-8 encoding format to store Strings in class files. UTF-8 is a simple encoding of UNICODE characters and strings that is optimized for the ASCII characters. In each byte of the encoding, the high bit determines if *more bytes follow*. A high bit of zero means that the byte has enough information to fully represent a character; *ASCII characters require only a single byte*.

Many non-ASCII UNICODE characters still need only two bytes, but some may require three to represent in this format.

The Two Types of I/O

Consider the UNICODE translation for a moment. Anytime we need to read text,

we need to perform translation from a native format (one or more bytes per character) to two-byte characters in UNICODE.

But what about binary files?

Obviously we don't want to take a binary file such as a database and perform the UNICODE translation on its contents. We need two basic ways to handle data.

Text I/O Versus Binary I/O

Java's I/O classes are divided into two main groups, based on whether you want text or binary I/O.

`Reader` and `Writer` classes handle text I/O. `InputStream` and `OutputStream` classes handle binary I/O.

Any time you see "Reader" or "Writer" as part of a Java I/O class name, you should immediately think "text I/O". Anytime you see "Stream" as part of the class name, think "binary I/O".

Reader and InputStream

Java supplies `Reader`s and `InputStream`s to read data; their use is similar. The following table shows the most commonly used methods in these classes. See the javadocs for the other methods available.

Note that these two classes are *abstract*; you won't ever create an instance of either, but they provide the base implementation details for all other input classes.

`void close()`	Closes the input. Always call this method when you are finished reading what you need, as it allows the VM to release locks on the file.
`int read()`	Reads a single *item* from the file. In `Reader`, an *item* is a `char`, while in `InputStream` it's a `byte`. The return value will either be the *item* or -1 if there is no more data (end-of-file has been reached.)
`int read(`***type***`[])`	Attempts to fill the array with as much data as possible. If enough data is available, the ***type***[] (`char[]` for `Reader`, `byte[]` for `InputStream`) will be filled with the data and the length of the

	array will be returned. If there's not enough data available, it will wait until the data is available or end-of-file is reached. In the case of end-of-file, as much data as is available will be copied into the array and the *amount* of that data will be returned. ***Note that the remaining elements in the array will* not *be modified and should not be used!*** Generally you will call this method in a loop until it does not return the length of the array as its result.
`int read(`***`type`***`[], int offset, int length)`	Similar to `read(`*`datum[]`*`)` but allows you to start at a specified offset in the input and read a limited number of bytes.
`int skip(int n)`	Skips past the next *n* bytes or characters in the file, returning the actual number that were skipped. (If for example, end-of-file was reached, it might skip fewer than requested).

Writer and OutputStream

Java supplies `Writer` and `OutputStream` to write data; their use is similar. The following table shows the methods provided in these classes.

Note that these two classes are *abstract*; you won't ever create an instance of either, but they provide the base implementation details for all other output classes.

`void close()`	Closes the file and releases and held resources or locks. *Alwa* call this method when you are finished writing a file. Note th `close()` will also `flush()` the file contents.
`void flush()`	Flushes the current contents of any internal buffers to the re file.
`void write(type[])`	Write the data from ***type***[] (`char[]` for `Writer`, `byte[]` for `OutputStream`) to the file. All data in the array will be writt

`void write(type[], int offset, int length)`	Similar to `write(type[])`, but only *length* units of data will written from ***type***[], starting at the *offset*.
`void write(int)`	Writes a single item (`char` for `Writer`, `byte` for `OutputStream`) to the file.
`void write(String)` (`Writer` only!)	Writes the contents of a `java.lang.String` to the file.
`void write(String, int offset, int length)` (`Writer` only!)	Writes the substring starting at *offset* and *length* characters lo to the file.

Reading and Writing Files

To read and write from files on a disk, use the following classes:

- `FileInputStream`
- `FileOutputStream`
- `FileReader`
- `FileWriter`

They provide concrete implementations of the abstract input and output and output classes above.

Each of these has a few constructors, where *class* is the name of one of the above classes:

- *`class`*`(File)` - create an input or output file based on the abstract path name passed in
- *`class`*`(String)` - create an input or output file based on the `String` path name
- *`class`*`(FileDescriptor)` - create an input or output file based on a `FileDescriptor` (you generally won't use this and this class will not discuss it)
- *`class`*`(String, boolean)` - [**for output classes only**] create an input or output file based on the path name passed in, and if the boolean parameter is true, *append* to the file rather than *overwrite* it

For example, we could copy one file to another by using:

```
import java.io.*;

public class FileCopy {
  public static void main(String args[]) {
    try {
      // Create an input file
      FileInputStream inFile =
        new FileInputStream(args[0]);

      // Create an output file
      FileOutputStream outFile =
        new FileOutputStream(args[1]);

      // Copy each byte from the input to output
      int byteRead;
      while((byteRead = inFile.read()) != -1)
        outFile.write(byteRead);

      // Close the files!!!
      inFile.close();
      outFile.close();
    }

    // If something went wrong, report it!
    catch(IOException e) {
      System.err.println("Could not copy "+
                         args[0] + " to " + args[1]);
      System.err.println("Reason:");
      System.err.println(e);
    }
  }
}
```

This is a *horribly* slow implementation, however. The problem is that we are reading and writing a single character every time.

Consider another case: suppose the input and output were at opposite ends of a socket. If we write a single character at a time, an entire *packet* might be sent across the network *just* to hold that single byte.

To improve performance, we could add our own simple buffering scheme using two other `read()` and `write()` methods (bold text shows the differences):

```
import java.io.*;

public class FileCopy {
  public static void main(String args[]) {
    try {
      FileInputStream inFile =
        new FileInputStream(args[0]);
```

```
      FileOutputStream outFile =
        new FileOutputStream(args[1]);

      byte[] buffer = new byte[1024];
      int readCount;

      while( (readCount = inFile.read(buffer)) > 0)
        outFile.write(buffer, 0, readCount);

      inFile.close();
      outFile.close();
    }
    catch(IOException e) {
      System.err.println("Could not copy "+
                         args[0] + " to " + args[1]);
      System.err.println("Reason:");
      System.err.println(e);
    }
  }
}
```

This creates a significantly faster program, but requires too much thinking on our part. We'll see a way to avoid any thought at all in a bit...

Other I/O Sources

Java's I/O model is very flexible. The basic input classes `InputStream` and `Reader` have a very simple interface: `read()` from somewhere. The basic output classes `OutputStream` and `Writer` have a very simple interface: `write()` to somewhere.

So far we've seen useful subclasses of these that read from and write to a file on the file system. However, we can use several other types of inputs and outputs as well.

Reading/Writing Arrays of Bytes

Classes `ByteArrayInputStream` and `ByteArrayOutputStream` provide I/O with an array of `byte`s.

The constructor for `ByteArrayInputStream` takes a parameter of type `byte[]` (an array of `byte`s), and possibly information about which contiguous section of that array to read. Any `read()` requests will grab `byte`s from the `byte` array.

The constructor for `ByteArrayOutputStream` takes either no parameter or an `int` parameter that tells it how big you want the initial output array to be. After writing to a `ByteArrayOutputStream`, you can access the filled array by calling

`toByteArray()` on it.

For example

```
import java.io.*;

public class ByteArray1 {
  public static void go(byte[] bytes) {
    try {
      // wrap an input stream around the byte array
      ByteArrayInputStream in = new ByteArrayInputStream(bytes);

      // read each byte and echo to the console
      int b;
      while((b = in.read()) != -1)
        System.out.println(b);

      // close the byte array
      in.close();
    }

    // in case there was an I/O exception...
    catch(IOException e) {
      e.printStackTrace();
    }
  }

  // run our sample test
  public static void main(String[] args) {
    // create some test data
    byte[] bytes = {1,2,3,4,5,6,7,8,9};

    // pass it to the method
    go(bytes);
  }
}
```

This isn't a terribly interesting example, but the real power of the Java I/O model can be demonstrated by changing our `go()` method to take an `InputStream`. Because `ByteArrayInputStream` is a subclass of `InputStream`, as is `FileInputStream`, we could then pass *either* subclass -- the `go()` method wouldn't know or care if its data was really coming from a file or an array of `byte`s:

```
import java.io.*;

public class ByteArray2 {
  public static void go(InputStream in) throws IOException {
    // read each byte and echo to the console
    int b;
    while((b = in.read()) != -1)
```

```
      System.out.println(b);
  }

  // run our sample test
  public static void main(String[] args) {
    try {
      // create some test data
      byte[] bytes = {1,2,3,4,5,6,7,8,9};
      InputStream in = new ByteArrayInputStream(bytes);

      // pass it to the method
      go(in);

      // close the stream
      in.close();

      // now read from a file
      in = new FileInputStream("somebytes.bin");

      // pass it to the method
      go(in);

      // close the stream
      in.close();
    }

    // in case there was an I/O exception...
    catch(IOException e) {
      e.printStackTrace();
    }
  }
}
```

In this example, we can pass in a `ByteArrayInputStream` or a `FileInputStream`; `go()` doesn't care! (Note that we rearranged where the exception handling occurs to reduce the code required.)

`ByteArrayOutputStream` works in a similar manner. You can `write()` `byte`s to a `byte` array as well.

```
import java.io.*;

public class Test1 {
  public static void go(OutputStream out) throws IOException {
    // read each byte and echo to the console
    for(int i = 0; i < 11; i++)
      out.write((byte)i);
  }

  // run our sample test
  public static void main(String[] args) {
    try {
      // create some test data
```

```
      ByteArrayOutputStream out = new ByteArrayOutputStream(10);

      // pass it to the method
      go(out);
      out.close();

      byte[] b = out.toByteArray();

      for(int i = 0; i< 11; i++)
        System.out.println(b[i]);
    }

    // in case there was an I/O exception...
    catch(IOException e) {
      e.printStackTrace();
    }
  }
}
```

In this example, the `go()` method just knows it's writing binary data to an `OutputStream`; it doesn't realize it's actually writing to an array. We create a `ByteArrayOutputStream` with space for 10 elements initially. If we need more space (as we do in this example), the `ByteArrayOutputStream` creates it for us automatically.

Once we're done writing to the `ByteArrayOutputStream`, we access the underlying array of `byte`s by calling `toByteArray()` on it.

Reading/Writing Arrays of Characters

`CharArrayReader` and `CharArrayWriter` act just like `ByteArrayInputStream` and `ByteArrayOutputStream`, except they use arrays of `char`s instead of `byte`s.

Note the differences in the names: these names have "Reader" and "Writer" in them; we're dealing with UNICODE characters rather than binary data.

You use these classes in exactly the same manner as `ByteArrayInputStream` and `ByteArrayOutputStream`.

Reading/Writing Strings

`StringReader` and `StringWriter` act the same as well. They use a `String` as their source or target instead of an array of `byte`s or `char`s.

Again, these are used in the same manner as `ByteArrayInputStream` and `ByteArrayOutputStream`.

It is worth noting that there is a class named `StringBufferInputStream`. *This*

class has been deprecated and should not be used. Note the name - it has "StringBuffer" which implies UNICODE character input, and "InputStream" which implies binary data input. The problem with this class is that native-format text is *not* translated into UNICODE during the `read()`. If you want to read from a `String`, use `StringReader`.

Remote I/O

Remote I/O is accomplished by sending data across a network connection. Java provides several networking classes in the package java.net. You can specify a `URL` for a remote file and open an `InputStream` to read it, send HTTP requests via various HTTP classes, and directly access sockets via `Socket` and `ServerSocket`.

Details on these classes are not covered in this module other than to say that communication over these connections is accomplished simply by using `InputStream`s and `OutputStream`s. This makes networking in Java incredibly easy. Just set up a connection/socket, ask for its input or output streams, then read and write.

The Java I/O model doesn't really care where the data is coming from or where you're sending it. You just `read()` and `write()`.

Filters

Taking the idea of the source or target of an I/O operation being "anything" a bit further, we *could* use *another* input or output stream (or reader/writer) as our source or target. Java defines some classes that act as *filters* in an I/O operation. They wrap another input/output stream, modifying the data after it is read or before it was written.

Think of a filter in a coffee machine. Water drips into the filter (which could contain some coffee grounds) and is *changed*, taking on the flavor of the coffee. It then drips down into the coffee pot. Think of the coffee pot as the "ultimate" destination for the water. As the coffee-machine operator, you place a *filter* between yourself and the coffee pot. You pour water into the *filter* (more or less), and the *filter* pours the water into the final destination.

We could place as many different filters in between as we would like. Suppose we had one that changes water to coffee, and another that adds a mint flavor to it. The final result is "mint coffee". We could view this as the following diagram (real mint coffee might appear a different color...)

This picture represents a substance being passed into Filter 1, being changed somehow, then being passed through Filter 2, being changed again, and finally dropping into the output.

Now suppose for some reason we wanted to change the mint coffee *back* to water. In an alternate universe, we might be able to attach a giant vacuum at the top of our coffee maker and suck the water through the mint filter (removing the mint flavor), then up through the coffee filter (removing the coffee flavor), leaving us with plain old water.

This is exactly how I/O filters work in Java, except the water represents the *data* being written, and the filters translate parts of that data as it's being written or read, or collect the data until we're really ready to release it to the ouput.

Basic Java Implementation of Filters

Java provides classes `FilterInputStream`, `FilterOutputStream`, `FilterReader`, and `FilterWriter` to capture the essence of input and output filtering. Each of these classes provides the same functionality as the input and

output classes we're already seen, *but* each takes *another* input/output class as an argument to its constructor. We'll refer to these other classes as *delegates*, classes that we pass the input or output request to. In particular,

- *outputDelegate* - this is a class whose `write()` method a filter calls when the filter is asked to `write()`
- *inputDelegate* - this is a class whose `read()` method a filter calls when the filter is asked to `read()`

So What's a Delegate???

Delegation is the formal term for "passing the buck". Rather than completely performing a task yourself, you ask someone (or something) else to perform part of all of that task for you. Let's take a simple, and very common, example.

Nancy asks her husband Scott to clean the house while she goes to the market. Scott, being a brilliant (read "lazy") programmer, decides it would be a good to delegate the task, so he asks each of his kids to clean part of the house. (In return, he promises to take them to Toys R Us to get a set of Legos.) The kids do a decent job cleaning up, but not perfect. Before Nancy returns home, Scott "fixes" things a bit, such as placing the sheets on the bed instead of the sofa.

Let's analyze what happened in this scenario:

- **Nancy asks Scott to clean the house.** Think of this as "the Nancy object sends a cleanHouse message to the Scott object."
- **Scott asks the kids to clean the house.** Here's the delegation in action. The Scott object has references to his kid objects (he'd better or the Nancy object will get upset), and when sent a cleanHouse message, *he* sends cleanHouse messages to those kid objects. These kid objects are known as *delegates,* because Scott is asking *them* to do a job he is responsible for.
- **The kids each clean part of the house.** A common use of delegation is to divide a task into smaller, more manageable pieces. Each piece performs its task *in its own way*, and the end result is that the overall task is performed.
- **Scott "fixes" things.** Often, delegates don't do the exact job that you need, and you need to "tweak" the results, or perhaps combine the results somehow (perhaps sweeping all the individual dust piles the kid objects created under the throw rug in the bathroom).

We might represent this in the Java language as follows:

```
public class Nancy {
  // keep track of a Scott object so we can ask him to clean
  private Scott scott = new Scott();

  public void getReadyToGoToMarket() {
    scott.cleanHouse(); // ask Scott to clean the house
  }
}


public class Scott {
  // keep track of the kids...
  private Kid nicole    = new Kid();
  private Kid alexander = new Kid();
  private Kid trevor    = new Kid();
  private Kid claire    = new Kid();

  public void cleanHouse() {
    // keep claire busy watching TV
    claire.watchTV();

    // ask nicole, alexander and trevor to clean some rooms
    // here we are delegating the task to other objects
    nicole.clean("kitchen");
    alexander.clean("living room");
    trevor.clean("bedroom");

    // "fix" a few things...
    find("sheet").putOn(find("bed"));
    find("phone").putOn(find("kitchen counter"));
    find("remote control").putOn("couch");

    // ... reward the kids ...
  }

  // ... other methods ...
}


public class Kid {
  public void clean(String room) {
    // ... clean the room ...
  }
}
```

You may have noticed in the above example that the delegates perform their tasks *synchronously*. Alexander waits for Nicole to finish cleaning before he starts, and Trevor waits for Alexander to finish before he starts. It is possible to have delegates run in separate threads as well, but that is not how Java uses delegates for I/O and is out of the scope of this module.

Delegation is useful in cases where

- we don't know how to perform a task, or
- it would be more efficient to have a different object perform the task, or
- code for the exact task has already been written in another class that is not our superclass, or
- we want the basic implementation of another class's method(s) but need to modify them slightly (but cannot subclass for some reason)

The last case is the one that we will concentrate on now. Java uses delegation to change the way a delegate reads and writes a stream.

Java's Use of I/O Delegates

Java uses delegates to do the "real I/O" in a filter. Each filter's constructor has an argument that is a `Reader`, `Writer`, `InputStream`, or `OutputStream`. When the filter is asked to perform I/O, it *delegates* to this argument.

```
FilterInputStream(InputStream inputDelegate)
FilterOutputStream(OutputStream outputDelegate)
FilterReader(Reader inputDelegate)
FilterWriter(Writer outputDelegate)
```

Each `read()` request first calls the *inputDelegate*'s `read()` to get the data. The data is then modified (translating or removing characters, perhaps) and returned.

Each `write()` request *first* modifies the output data, then passes it to the *outputDelegate*'s `write()` method.

Note that `FilterInputStream` is a subclass of `InputStream`! This means that you *could* pass a `FilterInputStream` to a `FilterInputStream`, chaining filters to perform a series of translations. Think of the benefit. Suppose we wanted to apply filters that uppercase all characters and translate tabs to spaces. Suppose further that sometimes we only want to perform one of these filtering operations.

Without Java's ability to "nest" filters, we would require a class for each type of output:

- `UppercaseTabToSpaceWriter`
- `UppercaseWriter`
- `TabToSpaceWriter`

- `Writer`

We would in many cases need every possible permutation of these filters, as sometimes the order of the translations might matter.

Compare this to what is required with the nesting filters:

- `UppercaseWriter`
- `TabToSpaceWriter`
- `Writer`

Note that we no longer need the `UppercaseTabToSpaceWriter` as we can simply nest the `UppercaseWriter` and `TabToSpaceWriter` when needed!

The general format for filter use is to "wrap" one around an existing binary or text stream. For example, using the above classes:

```
Writer realFile = new Writer("a.txt");
Writer filter2  = new TabToSpaceWriter(realFile);
Writer filter1  = new UppercaseWriter(filter2);
// use "filter1" as the target for output methods
```

or more simply

```
Writer out = new UppercaseWriter(
               new TabToSpaceWriter(
                 new Writer("a.txt")));
// use "out" as the target for output methods
```

Note that when combining the entire declaration, some people prefer to break the source line as above even if they have room to write it all on one line. This makes it easier to read how many levels of filters are in place.

Caution: this format is nice for examples and simple programs, but lacks robustness in the face of dealing with exceptions while opening filters! Be sure to examine the exceptions section at the end of this module for details on how to open and close filters safely!

Using the above filter is very simple: just write to it the same way you'd write to a `Writer`.

```
out.write("Hello!");
```

Closing Files with Attached Filters

It is very important that you close *only* the ***outermost*** filter. Suppose we had the

following scenario:

```
Writer realFile = new Writer("a.txt");
Writer filter2  = new TabToSpaceWriter(realFile);
Writer filter1  = new UppercaseWriter(filter2);

filter1.write("Hello!");

realFile.close();
```

Think about what might happen if either `TabToSpaceWriter` or `UppercaseWriter` decides to hold all its output until you close them (perhaps for buffering purposes). If you close the *real* file *before* closing the filters, there is no way to pump the output to the real file.

When you close a filter, it flushes any output it might have been storing to its *outputDelegate*, then calls `close()` on the *outputDelegate*. The proper code for the above would be

```
Writer realFile = new Writer("a.txt");
Writer filter2  = new TabToSpaceWriter(realFile);
Writer filter1  = new UppercaseWriter(filter2);

filter1.write("Hello!");

filter1.close();
  // flushes filter1
  // calls filter2.close()
       // flushes filter2
       // calls realFile.close()
       // close filter2 (mark closed)
  // close filter1 (mark closed)
```

Note: You do not need to close the inner filters, as the outer filter's `close()` method automatically calls the inner filter's `close()`.

We will see later, when we discuss exceptions, that closing a file is a bit more difficult than it may seem. The important concept to note at this point is the order in which you must close the filters -- close the outermost filter first!

Common Filters Provided with Java

Java comes with several useful filters that can be used "right out of the box". These filters perform various tasks such as buffering to make I/O more efficient, encoding primitive types, and writing the state of entire Java objects.

Buffering

As mentioned earlier, writing a single `byte` or a single `char` at a time is horribly inefficient. It is much better to write a block of `byte`s or `char`s in a single `write()` call. However, writing proper code to perform buffering can be tedious and error-prone. The java.io package includes four filters that can make this task painless.

- `BufferedInputStream`
- `BufferedOutputStream`
- `BufferedReader`
- `BufferedWriter`

By simply wrapping these around your I/O declarations, all input and output will be buffered and communicated with the underlying file in larger chunks.

Note that you should almost always wrap your input and output declarations in a buffer! The only time you wouldn't want to do so would be if the input or output class in question was already buffered (like `GZIPOutputStream`, which you will see later).

For example

```
BufferedInputStream bis = new BufferedInputStream(
                                new FileInputStream("a.bin"));

int byte = bis.read(); // reads a block of bytes from a.bin
                       // returns one
int byte = bis.read(); // returns next byte
int byte = bis.read(); // returns next byte
// when buffer is empty, the next read will read a block
//   from the file to refill it

bis.close(); // always close the outermost filter!
```

When using output buffers you can call the `flush()` method, which can forcefully empty the buffer into its *outputDelegate*, then call the *outputDelegate*'s `flush()` method. Closing an output buffer also performs a flush prior to closing its *outputDelegate*.

Finally, `BufferedReader` provides a very useful method, `readLine()`. This method will read all characters up to a "newline" sequence (see New-Line Characters below) and return a `String` containing those characters. For example:

```
import java.io.*;
```

```
public class ReadLineTest {
  public static void main(String[] args) {
    try {
      // create a BufferedReader so we can read a line at a time
      BufferedReader br = new BufferedReader(
                              new FileReader(args[0]));

      // print the line number and the line
      String line = null;
      int i = 0;
      while((line = br.readLine()) != null) {
        i++;
        System.out.println(i + ": " + line);
      }
      br.close(); // always close outermost filter!
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Later you will see that `DataInputStream` *also* provides a `readLine()` method. The `readLine()` in `DataInputStream` should *never* be used! Remember, "InputStream" means *binary* input!

Using Print Streams

Classes `PrintStream` and `PrintWriter` provide convenient ways to write a String representation of primitive and object values. You've certainly used a `PrintStream` already; it's the class that provides the `println()` method, as in `System.out.println(...)`.

These classes provide `print()` and `println()` methods for each primitive type and for `Object`. The `print()` method converts its argument to a `String`, then writes that `String` to its *outputDelegate*. The `println()` method does the same thing `print()` does, but follows it with a newline character that is appropriate for the current platform (`'\n'`, `'\r'`, or `'\r\n'`).

For primitive arguments, the `String` class' `valueOf()` method is invoked, passing the argument to `print()` or `println()`. For `Object` arguments, the object's `toString()` method is invoked to perform the translation.

There are two well-known `PrintStream`s in the Java core API: `System.out` and `System.err`. These streams provide access to the "standard output" and "standard error output" for the current platform.

Flushing and PrintStreams

Because `PrintStream` is commonly used to communicate status with a user (through console output), flushing the output becomes a significant issue. When you print output, you would need to flush periodically to ensure that output is available:

```
PrintStream ps = new PrintStream(new FileOuputStream("foo.txt"));
ps.println("Hello!");
ps.flush();
ps.println("How are you?");
ps.flush();
```

This would be a significant burden to ensure you are flushing at appropriate intervals, so Sun added the capability to "autoflush" `PrintStream` and `PrintWriter`. Autoflush is disabled by default, and you can enable it by passing true to the `PrintStream` or `PrintWriter` constructor:

```
PrintStream ps = new PrintStream(new FileOutputStream("foo.txt"),
                                 true);
ps.println("Hello!");
ps.println("How are you?");
```

When autoflush is enabled, whenever a `'\n'` character is written to a `PrintStream` or whenever `println()` is called, the output is flushed. Note that `System.out` and `System.err` enable autoflush.

You should be noticing by now that `Stream.html"Print` is being used for text output. `PrintStream` is setup to convert characters to the appropriate binary representation on the current platform. It does so by passing output through a special filter (which we'll discuss in a moment) that converts the characters to the appropriate binary representation. (This conversion was necessary to allow backward compatibility with earlier versions of the JDK.) *However, you should always use* `PrintWriter` *for text output*. It's more efficient and guarantees the appropriate translation.

There is a subtle difference in the use of `PrintWriter` and `PrintStream`; `PrintWriter` only performs autoflushing after `println()` calls -- it *does not* watch for `'\n'`.

IOExceptions and PrintStreams

Although we have not yet covered `IOException`s, it is important to note that `PrintStream` and `PrintWriter` *do not throw* `IOException`s! `IOException` is a *non-Runtime* exception, which means that your code must catch them or declare it can throw them.

The creators of Java realized that `System.out` and `System.err` would be very heavily used, and did not want to force inclusion of exception handling every time you wanted to write `System.out.println(4)`.

Therefore, `PrintStream` and `PrintWriter` *catch their own exceptions* and set an error flag. If you are using one of these classes for *real* output in your program (not merely using `System.out.println()`) you should call `checkError()` to see if an error has occurred.

Because of this behavior,* `PrintStream` *and* `PrintWriter` *are not well suited for use other than* `System.out` *and* `System.err`*!

Tracking Line Numbers

A common task is to keep track of which line number is being read. Working this into the Java filter model is simple!

The core Java API defines two classes to perform this task: `LineNumberInputStream` and `LineNumberReader`.

`LineNumberInputStream` *has been deprecated because it assumes that a* `byte` *is sufficient to represent a character. Do not use* `LineNumberInputStream`! That said, we have less to discuss.

`LineNumberReader` is a filter that tracks how many "new line" sequences ("\n", "\r", or "\r\n") have been encountered while reading from its *inputDelegate*. This class implements its own buffer; you do not need to wrap it in a `BufferedReader`.

After reading any text from a `LineNumberReader` (for example, calling its `readLine()` method), you can call its `getLineNumber()` method to the current line number. You can interpret this line number in two ways:

- the line number of the *next* character to read, based at "line 0"
- the line number of the *last newline* read, based at "line 1"

If you need to know the line number *before* you have read characters on that line, you'll probably want to add "1" to the line number returned. If you don't need the line number until after reading the line, the line number is fine "as is". An example of using the line number *after* reading the line:

```
import java.io.*;
```

```
public class LineNumberReaderTest {
  public static void main(String[] args) {
    try {
      LineNumberReader lnr = new LineNumberReader(
                                new FileReader(args[0]));

      String line = null;
      while((line = lnr.readLine()) != null)
        System.out.println(lnr.getLineNumber() + ": " + line);

      lnr.close(); // always close outermost!
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Primitive Data Input and Output

You can store primitive values by writing to a `PrintStream`; reading those values in requires that they be parsed from `String`s, however, which is inefficient. A better way to deal with primitive values is to write binary data to the file.

The Java core APIs define interfaces `DataInput()` and `DataOutput()` to provide methods that will write a binary representation of a primitive value (as well as UTF-8 representations of `String` values.) These interfaces define methods such as `readInt()`/`writeInt()`, `readFloat()`/`writeFloat()`, and `readUTF()`/`writeUTF()`.

Classes `DataInputStream` and `DataOutputStream` implement these interfaces as filters. You can use `DataOutputStream` to save information (application configuration, perhaps) and `DataOutputStream` to retrieve it.

For example, write some data:

```
import java.io.*;

public class DataOutputTest {
  public static void main(String[] args) {
    try {
      DataOutputStream dos = new DataOutputStream(
                                new
FileOutputStream("config.bin"));
      dos.writeInt(4);
      dos.writeFloat(3.45678f);
      dos.writeChar('a');
      dos.writeChar('b');
```

```
      dos.close(); // always close outermost!
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

then read the data:

```
import java.io.*;

public class DataInputTest {
  public static void main(String[] args) {
    try {
      DataInputStream dis = new DataInputStream(
                              new FileInputStream("config.bin"));
      int   someInt   = dis.readInt();
      float someFloat = dis.readFloat();
      char firstChar  = dis.readChar();
      char secondChar = dis.readChar();

      System.out.println(someInt);
      System.out.println(someFloat);
      System.out.println(firstChar);
      System.out.println(secondChar);

      dis.close(); // always close outermost!
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Note that you must ensure that the order of read values is the same as the order in which they were written!

We'll discuss the external format of the output a bit later.

Compressing Input and Output

If you need to send large blocks of data over the network or infrequently use large blocks of data, you can reduce network bandwidth requirements or the file size of these chucks by taking advantage of the `GZIPInputStream` and `GZIPOutputStream` classes, introduced with Java 1.1. Hiding in the `java.util.zip` package, these two classes can reduce your size requirements considerably. The only cost is a few CPU cycles to perform the compression. If you have the cycles to spare, it's well worth the effort to use the new streams, especially to reduce network bandwidth requirements.

The following program demonstrates the usage of `GZIPInputStream`, as well as shows how much the source file has been compressed (in memory only).

```
import java.io.*;
import java.net.*;
import java.util.zip.*;

public class CompressIt {
  public static void main (String args[]) {
    if (args.length !=1) {
      displayUsage();
      System.exit(-1);
    } else {
      String filename = args[0];
      compressIt(filename);
    }
  }
  private static void displayUsage() {
    System.err.println(
      "Usage: java CompressIt filename");
  }
  private static void compressIt(String filename) {
    try {
      File file = new File(filename);
      // small files only
      int length = (int)file.length();
      FileInputStream fis = new FileInputStream(file);
      BufferedInputStream bis =
        new BufferedInputStream(fis);
      ByteArrayOutputStream baos =
        new ByteArrayOutputStream(length);
      GZIPOutputStream gos =
        new GZIPOutputStream(baos);
      byte buffer[] = new byte[1024];
      int bytesRead;
      while ((bytesRead = bis.read(buffer)) != -1) {
        gos.write(buffer, 0, bytesRead);
      }
      bis.close();
      gos.close();
      System.out.println("Input Length: " + length);
      System.out.println("Output Length: " +
        baos.size());
    } catch (FileNotFoundException e) {
      System.err.println ("Invalid Filename");
    } catch (IOException e) {
      System.err.println ("I/O Exception in transit");
    }
  }
}
```

The following table demonstrates the compression results. Depending upon the type of input, the results can vary considerably.

File	Size	Compressed
CompressIt.java	1,330	540
(AWT) Component.java	133,742	26,042
Symantec JIT - symcjit.dll	419,840	193,501
Java 2 SDK, version 1.2.2 - src.jar	17,288,462	3,252,177

Using Object Serialization

Similar to primitive value input and output, object values can be written to binary files as well. Writing an object value to an output stream is known as "serializing" that object.

The Java core API defines interfaces `ObjectInput` and `ObjectOutput`. These interfaces define the `readObject()` and `writeObject()` methods, the core of Java object serialization. Two classes, `ObjectInputStream` and `ObjectOutputStream` implement these interfaces.

Another interface, `Serializable`, is used to "mark" a class as one that can be handled via `writeObject()` and `readObject()`. Note that `Serializable` defines no methods; it is simply a "tag" to indicate that a class may be serialized.

Object serialization is covered in detail in MageLang's Object Serialization course module, and will not be covered further here.

Pushing Back Input

Sometimes you want to read all input before, but not including, a certain character. For example, you might want to read the next variable name from a stream, and the name could be terminated by a space or an operator like '+' or '-'. This becomes a tricky task, as you need to *read* that next character (the space or '+' for example) to determine where the variable name ends. You then need to keep track of it so you can use it as part of the *next* chunk being read.

The Java core I/O API provides two classes, `PushbackInputStream` and

`PushbackReader` to assist in these cases. Each of these classes contains a "pushback buffer". If the buffer isn't empty, characters are read *from the buffer*. If the buffer is empty, characters are read *from its input delegate.* You can "push" any characters you want into that buffer by calling the `unread()` method. The `unread()` method can take a `char` or `char[]` for `PushbackReader`, or a `byte` or `byte[]` for `PushbackInputStream`. Keep in mind that subsequent calls to `unread()` push `char`s or `byte`s *in front of* the stream to be read *as well as any* `char`s *or* `byte`s *that are* already in the buffer.

As an example, suppose we wanted to separate variables and operators as we read simple expressions. For this example, assume an operator is any combination of "+', '-', '*', '/' and '='. Further, expressions and operators can be separated by spaces, tabs or new-line sequences. Finally, variables are any sequence of any other character.

A naive implementation might read characters until it sees a space, tab or new-line sequence, collecting every character read as an operator or variable. Anytime it sees one of these separator characters, it can simply throw it away.

The problem with that approach it that it neglects cases where variables and operators are adjacent in the input. It would then need to keep track of the "different" character and prefix it to the next sequence. We can do this easily by taking advantage of `PushbackReader`.

```
import java.io.*;

/** Demonstrate use of PushbackReader for reading a
 *    file with simple expressions.
 *  The basic idea is that whenever we read a char that
 *    terminates the current "chunk", _if_ the read char
 *    is something we care about, we push it back on the
 *    input stream
 */
public class ExpressionTest {

  /** Report the variables and operators in the expressions
   *    in the file passed in
   */
  public static void main(String[] args) {
    try {
      PushbackReader pbr = new PushbackReader(
                              new BufferedReader(
                                new FileReader(args[0])));

      int c = 0; // the character we're reading

      // are we currently reading a variable or operator?
```

```
        boolean readingVariable = false;

        while(c != -1) { // loop until end-of-file

          // the current variable or operator
          StringBuffer chunk = new StringBuffer();

          // are we done with the current chunk?
          boolean chunkEnded = false;
          while(!chunkEnded) {
            switch(c = pbr.read()) {
              // operator characters
              case '+':
              case '-':
              case '*':
              case '/':
              case '=':
                // if we're working on an operator, append to it
                if (!readingVariable)
                  chunk.append((char)c);

                // if we're working on a variable:
                //  -- STOP building the variable
                //  -- switch mode to "operator"
                //  -- put back the operator char for later
                else {
                  // Switch reading mode to operator for next time
                  readingVariable = false;
                  // PUT IT BACK so we can read it next time!
                  pbr.unread(c);
                  chunkEnded = true;
                }
                break;

              // separator characters
              // if any of these chars are encountered, we
              //   stop building the current variable or operator
              case '\n': // unix newline
              case '\r': // mac newline
                         // dos newline is "\r\n"
              case '\t': // tab
              case ' ':  // space
              case -1:   // end-of-file
                chunkEnded = true;
                // note that the loop expects readingVariable
                //   to represent the _opposite_ of what the current
                //   chunk is when printing it
                readingVariable = !readingVariable;
                break;

              // any other character is considered part of a
variable
              default:
                // if we're working on a variable, append to it
```

```
                if (readingVariable)
                  chunk.append((char)c);

                // if we're working on an operator:
                //  -- STOP building the operator
                //  -- switch mode to "variable"
                //  -- put back the variable char for later
                else {
                  // Switch reading mode to variable for next time
                  readingVariable = true;
                 // PUT IT BACK so we can read it next time!
                  pbr.unread(c);
                  chunkEnded = true;
                }
                break;

            } // end switch
          } // end while(!chunkEnded)

          // print the chunk
          if (chunk.length() > 0) {
            System.out.println(
                (readingVariable?"OPERATOR":"VARIABLE") +
                ": " + chunk);
            chunk.setLength(0); // clear the string buffer
          }
        }
        pbr.close();
      }

      catch(Exception e) {
        e.printStackTrace();
      }
    }
}
```

With the following in file `test.txt`,

```
x + y*z
x++ + ++y
x/=y+z+q
```

running command

```
java ExpressionTest test.txt
```

we get the following output

```
VARIABLE: x
OPERATOR: +
VARIABLE: y
OPERATOR: *
VARIABLE: z
VARIABLE: x
```

```
OPERATOR: ++
OPERATOR: +
OPERATOR: ++
VARIABLE: y
VARIABLE: x
OPERATOR: /=
VARIABLE: y
OPERATOR: +
VARIABLE: z
OPERATOR: +
VARIABLE: q
```

We will discuss some other approaches to separating this input at the end of this module.

Converting from Binary to Text I/O

There are some cases where you are presented a binary input or output stream and would like to process text over those streams. The Java core APIs define classes `InputStreamReader` and `OutputStreamWriter` to perform the necessary character conversion.

For example, the writeText() method in the following example is passed an `OutputStream`. To ensure proper character translation, we wrap the `OutputStream` in a `OutputStreamWriter`.

```
import java.io.*;

public class OutputStreamWriterTest {
  public static void main(String[] args) {
    try {
      FileOutputStream fos = new FileOutputStream("somefile.txt");
      writeIt(fos);
      fos.close();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  public static void writeIt(OutputStream out) {
    PrintWriter pw = new PrintWriter(new OutputStreamWriter(out));
    pw.println("Hello!");
    pw.println("How are you?");
    pw.flush(); // JUST FLUSH!!! Leave the file open
  }
}
```

Note that in this particular example, we *do not* close the file in our `writeIt()` method. The file passed in is *assumed* to be open, and we want to leave it that way when we exit. However, it is important to `flush()` the Writers so our

changes are not lost when the Writers are garbage collected.

`InputStreamReader` is handled in a similar manner.

Concatenating Input Streams

Occasionally you may want to treat several files as a single input stream. Perhaps you have a method that takes an input stream and collects summary information about it. If you had several separate streams and you wanted to summarize all as one unit, you could not simply call the method once for each stream, as you'd get separate summaries for each stream.

`SequenceInputStream` takes two `InputStream`s or an `Enumeration` of `InputStream`s to combine any number of inputs effectively. Its `read()` methods will explicitly check for end-of-file, and if *another* stream exists it will continue reading from that other stream.

As a simple example, we can concatenate two `InputStream`s into a single `SequenceInputStream`:

```
BufferedInputStream in = new BufferedInputStream(
                           new SequenceInputStream(
                             new FileInputStream("input1.bin"),
                             new FileInputStream("input2.bin")));
```

We can then use stream `in` anywhere we would normally need an `InputStream`.

Perhaps you haven't noticed by now, but we haven't mentioned a `SequenceReader` class; that's because it does not exist in the Java core API! To deal with text input, you must treat a text file as *binary* `InputStream`, then convert the binary stream to a text stream:

```
BufferedReader br = new BufferedReader(
                      new InputStreamReader(
                        new SequenceInputStream(
                          new FileInputStream("somefile.txt"),
                          new
FileInputStream("anotherfile.txt"))));
```

If you have more than two streams you want to concatenate, you must either nest `SequenceInputStream`s or provide an `Enumeration` of `InputStream`s.

For example, concatenating three binary files by nesting might look like

```
BufferedInputStream in = new BufferedInputStream(
                           new SequenceInputStream(
                             new FileInputStream("input1.bin"),
```

```
                                new SequenceInputStream(
                                  new FileInputStream("input2.bin"),
                                  new
FileInputStream("input3.bin"))));
```

You can also create an `Enumeration` of the files. First, create a reusable `ArrayEnumerator` class:

```
public class ArrayEnumerator implements Enumeration {
  private int n = 0;
  private Object[] objects;
  public ArrayEnumerator(Object[] objects) {
    this.objects = objects;
  }

  public boolean hasMoreElements() {
    return n < objects.length;
  }

  public Object nextElement() {
    return objects[n++];
  }
}
```

Then you can use it in the `SequenceInputStream` as follows:

```
BufferedInputStream in = new BufferedInputStream(
                           new SequenceInputStream(
                             new ArrayEnumerator(
                               new Object[] {
                                 new
FileInputStream("input1.bin"),
                                 new
FileInputStream("input2.bin"),
                                 new
FileInputStream("input3.bin"),
                               })));
```

Creating Your Own Filters

If you want to create your own filter, you can subclass one of the following classes:

- `FilterInputStream`
- `FilterOutputStream`
- `FilterReader`
- `FilterWriter`

These classes provide some nice default support, such as handling the `close()` cascading.

Other uses for Filters

Filters can be used for other tasks as well, and might not modify the input or output at all! They could be used to help debug or log I/O operations by announcing that writing is actually taking place. In a case like this, the `read()` or `write()` methods would call the *input/outputDelegate*'s real `read()` or `write()` method and perform some logging, such as printing to System.err "write() was called!"

Swing is part of Sun's Java Foundation Classes (see http://java.sun.com/products/jfc

```
Reader r = new BufferedReader(
                  new InputStreamReader(
                        new ProgressMonitorInputStream(someJFrame,
                              "Reading a.txt...",
                              new FileInputStream("a.txt"))));
```

As you read from `Reader r`, the progress is automatically updated.

Piped Input and Output

There are two interesting classes in the Java core API that provide stream communication between threads. `PipedInputStream` and `PipedOutputStream` can be connected together as a nice meeting point for data between two threads.

`PipedOutputStream` is essentially used as the target of an I/O operation, similar to `FileOutputStream`. Rather than writing data to a file, it is passed to a `PipedInputStream` to read it. An analogy seriously helps understanding this one...

> Flint Fredstone (no similarity to another name which requires licensing fees) works in a granite quarry. He spends his day digging rocks with his high-tech digger/loader. His friend, Bernie Rumble drives a dump truck at the same quarry. Bernie is responsible for hauling rocks to the processing plant.
>
> Flint and Bernie cooperate quite nicely. Bernie drives his truck to the pickup point, and Flint dumps the rocks he's collected into Bernie's truck. If Bernie gets there before Flint, he waits for Flint to arrive. If Flint gets there before Bernie, he waits for Bernie to arrive.
>
> Bernie's truck holds only a fixed amount of rocks. When it's full, he drives to the processing plant. (Of course he could decide to take a partial load after Flint dumps it. It depends on how fast Flint can bring the rest of the load.) Flint may

be able to only dump a partial load in Bernie's truck, *then he must wait* for Bernie to return so he can dump the rest of the load.

`PipedOutputStream` acts exactly like Flint Fredstone in this analogy. You can use it as the final target of any stream output. `PipedInputStream` acts exactly like Bernie Rumble. A few key points about this analogy:

- All data storage is done in the `PipedInputStream`, just like Bernie's dump truck.
- `PipedInputStream` can hold only a fixed amount of data, then it forces the `PipedOutputStream` to *wait* until it processes some of that data. This blocking occurs until there is enough room for the data the `PipedOutputStream` wants to dump.
- When a `PipedInputStream` wants to `read()` data, it is blocked until there is at least *one* byte available to read. If less than the desired amount of data is available, *but at least one byte* is available, it will `read()` the available amount *and not wait* for more data.
- Before any I/O can be performed, a `PipedOutputStream` *must be connected* to a `PipedInputStream`. Otherwise, Flint is simply dumping his rocks in a big pile...
- Either end can close its stream first
- Think of it as Flint or Bernie leaving work.
- If the `PipedOutputStream` closes first, the `PipedInputStream` can keep reading the data it's stored, or close at any time. If Flint goes home first, Bernie can still take the rocks in his truck to the processing plant, or call it a day himself.
- If the `PipedInputStream` closes first, the `PipedOutputStream` *cannot write any more data!* If Bernie goes home first, Flint *has nowhere to dump his rocks*!

A final note before an example: Flint and Bernie *must* be independent entities. They each have their own keys for their vehicles and are extremely territorial! `PipedInputStream` and `PipedOutputStream` use Java's thread synchronization support, and because of the manner in which they are implemented, *if both the* `PipedInputStream` *and* `PipedOutputStream` *are used in the* ***same thread*** *they may cause a deadlock!*

Pipe Example

Now, a little example, implementing our analogy. We'll create a quarry that holds 1000 rocks. Flint's loader can carry 50 rocks at a time. Bernie's dump truck can carry 75.

First, we'll define classes to help us watch the progress of this scenario. We start with two filters that report `read()`s and `close()`s. Note that here we're using filters to simply report what's happening; we're *not* modifying the data! (Also note that these classes need to be defined in separate files)

```
import java.io.*;

/** An OutputStream filter that helps us watch what
 *  Flint Fredstone is doing
 */
public class FlintLogOutputStream extends FilterOutputStream {
  public FlintLogOutputStream(OutputStream out) {
    super(out);
  }

  public void write(byte b[]) throws IOException {
    System.out.println("Fred is dumping rocks into Bernie's
truck");
    super.write(b, 0, b.length);
    System.out.println("Fred is going to dig more rocks");
  }

  public void close() throws IOException {
    super.close();
    System.out.println("Flint goes home for the day");
  }
}

// ---------------------------------------------------------------
-

import java.io.*;

/** An input filter that helps us watch what Bernie Rumble
 *  is doing */
public class BernieLogInputStream extends FilterInputStream {
  public BernieLogInputStream(InputStream in) {
    super(in);
  }

  public int read(byte b[]) throws IOException {
    System.out.println("Bernie is ready to get loaded");
    int x = super. read(b, 0, b.length);
    if (x == -1)
      System.out.println("Nothing more to put in Bernie's
```

```
truck!");
    else
      System.out.println("Bernie has dropped off " + x +
                         " at the processing plant");
    return x;
  }

  public void close() throws IOException {
    super.close();
    System.out.println("Bernie goes home for the day");
  }
}
```

Next we'll subclass `PipedOutputStream` to modify the size of its buffer. For this example, we want to treat its buffer as the same size as the chunks that Bernie is reading. Luckily, the buffer is represented by a `protected` instance variable so we can access it.

```
import java.io.*;

/** Subclass of PipedInputStream so we can control
 *  the buffer size */
public class BerniePipedInputStream extends PipedInputStream {
  /** Shrink the buffer to 75 rocks... */
  public BerniePipedInputStream(PipedOutputStream src)
         throws IOException {
    super(src);
    buffer = new byte[75];
  }
}
```

Now we define two threads to represent the jobs that Flint and Bernie perform. Flint digs a certain number of rocks and dumps them somewhere (note that all he knows is that he's dumping them to an `OutputStream`!) Bernie takes any rocks he has been given to the processing plant, in groups of 75 if possible. Note that Bernie also has no idea where the data is coming from; all he knows is that it is an `InputStream`.

```
import java.io.*;

/** A thread that represents Flint Fredstone's job.
 *  He digs up a certain number of rocks and drops them off
 *    in a stream
 */
public class FlintThread extends Thread {
  private byte[] rocks = new byte[50];
  private OutputStream out;

  public FlintThread(OutputStream out) {
    this.out = out;
  }
```

```
  public void run() {
    try {
      // init out "array of rocks"
      for(int i = 0; i < 50; i++)
        rocks[i] = (byte)1;

      // have Fred deliver 350 rocks, 50 at a time
      for(int i = 0; i < 7; i++)
        out.write(rocks);

      out.close();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}

// ---------------------------------------------------------------

import java.io.*;

/** A thread that represents Bernie Rumble -
 *  He keeps hauling rocks to the processing plant as long
 *    as there are rocks for him to carry
 */
public class BernieThread extends Thread {
  private byte[] rocks = new byte[75];
  private InputStream in;

  public BernieThread(InputStream in) {
    this.in = in;
  }

  public void run() {
    try {
      // Take rocks as we get them to the processing plant
      // Note that we read blocks of 75 at a time
      int n;
      while((n = in.read(rocks)) != -1);
      in.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Finally, a test class to create the streams and send Flint and Bernie to work:

```
import java.io.*;

/** A Simple test scenario for Piped Input and Output,
```

```
 *    using the Flint Fredstone and Bernie Rumble
 *    analogy
 */
public class PipeTest {
  public static void main(java.lang.String[] args) {
    try {
      // Create the pipe streams for Flint and Bernie
      PipedOutputStream flintFredstone = new PipedOutputStream();
      BerniePipedInputStream bernieRumble =
         new BerniePipedInputStream(flintFredstone);

      // wrap the pipes in filters that allow us to watch the,
      InputStream in   = new BernieLogInputStream(bernieRumble);
      OutputStream out = new FlintLogOutputStream(flintFredstone);

      // start Flint and Bernie's work day
      // Note that they have no idea they're working together --
      //   they just get sent a stream to work with
      new BernieThread(in).start();
      new FlintThread(out).start();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

When we run the above program, we see the following results. Note the initial waiting that Bernie does to get a full load (because he is reading 75 "rocks" at a time.)

```
Bernie is ready to get loaded
Fred is dumping rocks into Bernie's truck
Fred is going to dig more rocks
Fred is dumping rocks into Bernie's truck
Bernie has dropped off 75 at the processing plant
Bernie is ready to get loaded
Fred is going to dig more rocks
Fred is dumping rocks into Bernie's truck
Fred is going to dig more rocks
Fred is dumping rocks into Bernie's truck
Bernie has dropped off 75 at the processing plant
Bernie is ready to get loaded
Fred is going to dig more rocks
Fred is dumping rocks into Bernie's truck
Bernie has dropped off 75 at the processing plant
Bernie is ready to get loaded
Fred is going to dig more rocks
Fred is dumping rocks into Bernie's truck
Fred is going to dig more rocks
Fred is dumping rocks into Bernie's truck
Bernie has dropped off 75 at the processing plant
Bernie is ready to get loaded
```

```
Fred is going to dig more rocks
Bernie has dropped off 50 at the processing plant
Flint goes home for the day
Bernie is ready to get loaded
Nothing more to put in Bernie's truck!
Bernie goes home for the day
```

Using Pipes to Pass Objects

Because pipes handle all the thread coordination for you, they can be an ideal way to pass `Object`s or primitive data between threads. If the only time you need threads to cooperate is to pass some objects back and forth, you can set up a `PipedOutputStream` wrapped in an `ObjectOutputStream` or `DataOutputStream` to *send* data from one thread, and the correspondingly wrapped `PipedInputStream` to receive the data in the other thread..

This can make inter-thread communication *much* simpler and more reliable if all you need is data passing!

Random Access Files

`RandomAccessFile` provides two-way communication with a file-system file to and from specific locations in that file.

Before describing this any further, a few things are important to note:

- You can use `RandomAccessFile` to work only with physical devices that provide random-access support. For instance, you cannot open a tape-drive stream as a random-access file.
- `RandomAccessFile` *does not* extend `InputStream`, `OutputStream`, `Reader`, *or* `Writer`. This means that you *cannot* wrap it in a filter!
- *You* are responsible for correct read positioning! If you write an `int` at position 42 in the file, you must make sure you are at position 42 when attempting to read that `int`!

`RandomAccessFile` implements `DataInput` and `DataOutput`; it *acts* like a combination of `DataInputStream` and `DataOutputStream`. You can read and write primitive and `String` data at any position in the stream. Of course you can also read and write bytes as you do when using `InputStream` and `OutputStream`.

To work with a `RandomAccessFile`, you create a new instance passing the name of the file (a `String` or `File` object) and the *mode* in which you want to work with the file. The mode is a `String` that can be either:

- `"r"` - the file will be opened in "read" mode. If you try to use any output methods an exception will be thrown. If the file does not exist, a `FileNotFoundException` will be thrown. (More on I/O exceptions later!)
- `"rw"` - the file will be opened in "read/write" mode. If the file does not exist, it will try to create it.

In either mode, you can use the `read()` methods as well as methods like `readInt()`, `readLong()` and `readUTF()`. In read/write mode you can use the corresponding `write()` methods.

The big differences with `RandomAccessFile` are the positioning methods. You can call `getFilePointer()` (which returns a `long`) at any time to determine the current position within the file. This method is useful if you want to track positions as you write data to a file, possibly to write a corresponding index.

You can jump to any location in the file as well, by calling `seek()` (passing a `long` value) to position the file pointer.

As an example, we present a simple address book lookup program. There are two parts to this example:

- **Address data creation** - address-listing objects are created and their data stored in a `RandomAccessFile`, *and* a sequential index file tracks the position of each record.
- **Address data lookup** - user inputs a name to seek, the index is determined for that name, and the name record is read from the `RandomAccessFile`.

Note that this is not an example of efficient indexing; a very simple sequential index is used. For higher efficiency and easy relationship tracking, we recommend using a database management system (DBMS).

First, we create a class to represent the data. It has variables for all pieces of information about a person. ***Note: to keep this example short, we have made the instance variables package-accessible; in general, all data* should *be private, and only accessed through*** `set()`/`get()` ***methods!***

```
import java.io.*;

/** A simple class that tracks address book
 *    information for a person.
 *  Note -- all data is package-accessible to keep the
 *          example short. In general, all data _should_
 *          be private and accessed through "get" and "set"
```

```
methods.
 */
public class AddressData {
  String name;
  String street;
  String city;
  String state;
  int    zip;
  int    age;
  String phone;

  /** Fill in our data from a random access file.
   *  Note that we must read in the same order we wrote
   *    the data...
   */
  public void readFrom(RandomAccessFile f) throws IOException {
    name   = f.readUTF();
    street = f.readUTF();
    city   = f.readUTF();
    state  = f.readUTF();
    zip    = f.readInt();
    age    = f.readInt();
    phone  = f.readUTF();
  }

  /** Create a string describing the person */
  public String report() {
    String nl = System.getProperty("line.separator");
    return name + nl +
           "  Address: " + street + nl +
           "           " + city + ", " + state + nl +
           "           " + zip + nl +
           "  Age:     " + age + nl +
           "  Phone:   " + phone;
  }

  /** Write our data to a random access file */
  public long writeTo(RandomAccessFile f) throws IOException {
    long location = f.getFilePointer();
    f.writeUTF(name);
    f.writeUTF(street);
    f.writeUTF(city);
    f.writeUTF(state);
    f.writeInt(zip);
    f.writeInt(age);
    f.writeUTF(phone);
    return location;
  }
}
```

Next, we create a program that writes our phone data and index.

```
import java.io.*;
```

```
/** A sample of creating an indexed random access file */
public class CreateAddressBook {
  public static void main(String[] args) {
    try {
      // create the index file
      PrintWriter index = new PrintWriter(
                            new FileWriter("phone.idx"));

      // create the phone data file
      RandomAccessFile data =
          new RandomAccessFile("phone.dat","rw");

      // write a few sample records
      writeIt(data, index, "Ernie",
              "123 Sesame Street, Apt E",
              "MuppetVille", "PBS", 12345,
              5, "123-4567");

      writeIt(data, index, "Cookie Monster",
              "123 Sesame Street, Apt C",
              "MuppetVille", "PBS", 12345,
              3, "COO-KIES");

      writeIt(data, index, "Fred Rogers",
              "111 Trolley Lane",
              "Neighborhood", "MakeBelive", 22222,
              78, "BMY-NABR");

      // close the index and the data file
      index.close();
      data.close();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  /** Write our information to a data file and create
   *    an index entry
   */
  public static void writeIt(RandomAccessFile data,
                 PrintWriter index,
                 String name,
                           String street,
                           String city,
                           String state,
                           int zip,
                           int age,
                           String phone)
                throws IOException {

    AddressData d = new AddressData();
    d.setName(name);
    d.setStreet(street);
```

```
    d.setCity(city);
    d.setState(state);
    d.setZip(zip);
    d.setAge(age);
    d.setPhone(phone);

    index.println(d.getName());
    index.println(d.writeTo(data));
  }
}
```

Finally, we write a program that allows a user to query the address book:

```
import java.io.*;
import java.util.*;

/** A sample of reading from an indexed random access file */
public class ReadAddressBook {

  public static void main(String[] args) {
    try {
      // Read the index file into a Hashtable for
      //   fast lookup - note we assume the file is ok...
      //   (generally this assumption is _not_ a good idea...)
      Hashtable indexHash = new Hashtable();
      BufferedReader index = new BufferedReader(
                             new FileReader("phone.idx"));
      String line;
      while((line = index.readLine()) != null)
        indexHash.put(line, new Long(index.readLine()));

      index.close();

      // wrap System.in so we can read a line at a time
      BufferedReader in = new BufferedReader(
                            new InputStreamReader(System.in));

      // open the phone data file
      RandomAccessFile data =
          new RandomAccessFile("phone.dat","r");

      // create a record to read data into
      AddressData d = new AddressData();

      // loop until user presses enter by itself
      boolean done = false;
      while(!done) {
        // ask for the name
        System.out.print("Name: ");
        String name = in.readLine();

        // if empty, quit!
        if (name.equals(""))
          done = true;
```

```
        else {
          // otherwise lookup the name in the index
          Long position = (Long)indexHash.get(name);

          // if not found, complain
          if (position == null)
            System.out.println(name + " not found!");

          else {
            // otherwise, read the phone data & report it
            data.seek(position.longValue());
            d.readFrom(data);
            System.out.println(d.report());
          }
        }
      }

      data.close();
    }

    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Running the above programs produces the following output. User input is in ***bold-italic*** type.

```
Name: Scott <enter>
Scott not found!
Name: Fred Rogers <enter>
Fred Rogers
  Address: 111 Trolley Lane
           Neighborhood, MakeBelive
           22222
  Age:     78
  Phone:   BMY-NABR
Name: Ernie <enter>
Ernie
  Address: 123 Sesame Street, Apt E
           MuppetVille, PBS
           12345
  Age:     5
  Phone:   123-4567
Name: Cookie Monster <enter>
Cookie Monster
  Address: 123 Sesame Street, Apt C
           MuppetVille, PBS
           12345
  Age:     3
  Phone:   COO-KIES
Name: <enter>
```

Platform Specifics

File systems are one of the most varying features of different operating systems. Every operating system seems to have a different way of handling files.

There are usually many "common concepts", but the implementations vary significantly. Most file systems have some sort of hierarchical structure, and each "file" in that structure usually has some information about whether it can be read and/or written, and what size it is.

However, many systems have "unique" features, such as OS/2's extended attributes or UNIX's levels of file permissions. Some systems offer case-insensitivity in file names, while others make case significant. Even the characters allowed in a file name can vary greatly from one platform to the next.

When writing file input and output code in a Java program, there are several issues you must be cautious about. We'll discuss some of the major issues here.

Usually it's simply a matter of making sure you're *aware* that there may be problems on other platforms. It's very easy to create a program that works on *your* platform, and send it to a friend where it fails.

Characters in File Names

Each platform defines which characters can be used in file names, whether or not case is significant, and how long a file name can be.

To write a successful, platform-independent program, you have to be careful about what assumptions you make when choosing file names.

Character Choices

Simply put, choose a bad character for a file name, and your program will not work on all platforms.

Take the asterisk ('*'), for example. Some platforms allow it as a character in a file name, while others do not. If you call your "saved options" file `foo*2.options`, your program may work on your development machine, but will fail on Windows, for example.

If you restrict your choice of characters in a file name to simply letters (A-Z and a-z) and numbers (0-9), *and* start the file name with a letter, your file name should

work just about anywhere.

File Name Length

This can be a very tricky thing to get right. Suppose you were writing a program on UNIX, which allows very long file names. If you take advantage of this capability, you could encounter file I/O errors on other platforms.

The absolutely safest approach is to restrict file names to the `8.3` file name scheme. Have no more than 8 characters before a ".", only a single ".", and no more than three characters *after* the ".". It can be ugly, but it *will* work.

Many platforms have limits on the length of the *path* specification of a file. Be careful of *very deeply* nested files in the file system. You might not be able to drill down far enough to find them.

This limitation is of particular concern if you are walking through a hierarchy and *building* the path names as you go. Depending on how you build them, you might end up with a name like

```
some/directory/../where/../over/../here/../../images/foo.gif
```

rather than the more direct

```
images/foo.gif
```

This can be a *very* easy trap to fall into.

File Name Case Sensitivity

Some platforms, like UNIX, provide case sensitivity in file names. The names `littleMissMuffet.txt`, `LittleMissMuffet.txt`, and `littleMissMuffet.TXT` represent unique files. On other systems, such as Windows, these names all refer to the same file.

This can become a significant problem in a Java program. Take the following code as an example:

```
FileWriter writer = new FileWriter("littleMissMuffet.txt");
writer.write("Some text");
writer.close();

// ... lots of code in between to hide the difference ...

BufferedReader reader = new BufferedReader(
                            new FileReader("LittleMissMuffet.txt"));
```

```
String text = reader.readLine();
reader.close();
```

Note the difference in the name of the file. If we run this example on Windows, *text* is set to "Some text". However, if we run this program on UNIX, a `FileNotFoundException` is reported, stating that file `LittleMissMuffet.txt` is not found.

As another example, consider the following code, written on UNIX, to generate a C and C++ program:

```
FileWriter writer = new FileWriter("program.c");
writer.write("/* ... c code ... */");
writer.close();

FileWriter writer = new FileWriter("program.C");
writer.write("/* ... C++ code ... */");
writer.close();
```

On UNIX, this program performs as expected, producing both C and C++ outputs (lowercase ".c" is used for C source, while uppercase ".C" is used for C++ code). When run on Windows, only one file exists after execution, containing the C++ source code.

To ensure your program works properly on all platforms, follow these guidelines for file name case:

- Always specify the same case of a file name everywhere that name appears!
- Never intentionally use the same name with different cases!

New-Line Characters

One of the most significant differences between file systems is how they determine where one "line" of text in a file ends and the next begins.

There are three general patterns used:

- `"\n"` - the "UNIX way"
- `"\r"` - the "Macintosh way"
- `"\r\n"` - the "DOS/Windows way"

Many people who are used to the C and C++ programming languages will immediately fall into the following trap:

```
FileWriter writer = new FileWriter("smurf.out");
writer.write("This is a test\n" +
             "followed by another line\n" +
             "and another");
```

From years of using `printf`, many programmers are used to using "\n" whenever they need a new line in output. This is *not* the case in Java.

In Java, *you are simply writing characters to a file*. Character "\n" is simply another character.

If you run the above code on a UNIX platform, *it will behave as expected*. If you open `smurf.out` in a text editor, you will see

```
This is a test
followed by another line
and another
```

However, if you run the above code under Windows and open it in notepad, you will see something like

```
This is a test[]followed by another line[]and another
```

where we use `[]` is a representation of a "bad character".

To make it work on Windows, your code would need to look like

```
writer.write("This is a test\r\n" +
             "followed by another line\r\n" +
             "and another");
```

but it would then be wrong on UNIX. Of course *both* solutions are incorrect on the Macintosh.

Fortunately, Java provides a few ways to represent the new-line symbol successfully.

- asking the VM to identify the platform, and writing a new-line sequence based on that information
- a system property named `line.separator`
- a set of classes called `PrintWriter` and `PrintStream`

The first option *may* sound tempting, but *at best* you'll anticipate the current platforms, and more platforms may be added. At worst, you may miss several important platforms or write incorrect code for a certain platform.

You've already seen how the `PrintWriter` and `PrintStream` classes work. What you may not have realized is that the way their `println` methods work is to write the *appropriate* new-line character based on the current platform. They get that information from the system property `line.separator`.

To use this information directly, you could write code as follows:

```
String newLine = System.getProperty("line.separator");
writer.write("This is a test"           + newLine +
             "followed by another line" + newLine +
             "and another");
```

The `line.separator` property is given a value *by the VM*. Because the VM knows the platform on which it is running, it can choose the proper new-line character sequence.

Slash and Path Separation Characters

Of similar concern are two path-related issues:

- which character is used to separate directories when specifying a file's path
- which character is used to separate files/directories in "path sequences"

The first issue may not seem too difficult, as you've already see that when Java creates a `File` object it "does the right thing" with regard to slashes.

However, the caution here has more to do with what information you show your users, or write to a generated batch/script file.

When presenting a file name to the user, if you create a `File` object and display one of its path names, it will be presented appropriately. If instead you need simply to put together a file name and show it to the user or write it to a file, you may be in for some confusion.

Suppose your program is running on Windows and contains the following code:

```
System.out.println("I cannot file images/smurf.gif!");
```

While many Windows users will interpret the message properly, some who think more literally may see the message and think "what's the 'smurf.gif' option to the 'images' program???"

If the situation is reversed, running on UNIX with the following code:

```
System.out.println("I cannot file images\\smurf.gif!");
```

the result is likely to be several choice words from the UNIX user regarding "stupid DOS programmers."

In both scenarios, you lose, because you're not presenting data properly for the user on a given platform.

To resolve this issue, use the system property `file.separator`. You can use it two ways:

```
String slash = System.getProperty("file.separator");
System.out.println("I cannot find images" + slash + "smurf.gif");
```

or, because the `File` class provides a static variable that's initialized to `file.separator`'s value:

```
System.out.println("I cannot find images" +
                   File.separator + "smurf.gif");
```

The other path-related issue is how to represent "path sequences". For example:

```
c:\windows;c:\bin
/usr/local/bin:/bin
```

The first is a Windows path sequence, while the second is from UNIX. Note the use of semicolon (';') in the first and colon (':') in the second. These are called "path separation characters", and can be different from one platform to the next.

Java provides a system property named `path.separator`, which is also loaded into `File.pathSeparator`. If you need to construct a path sequence for a user message (for example "You need to set your CLASSPATH to ...") it will help the user a great deal to use the proper path separation character.

```
System.out.println("CLASSPATH should be set to");
System.out.println("  " + dir1 + File.pathSeparator + dir2);
```

Endianness

The last major platform concern is its Endianness.

Many primitive data types, such as `int`, require multiple bytes to hold their value. There are two ways a machine can order these multi-byte quantities when writing them to a file: most-significant byte (MSB) first, or least-significant byte (LSB) first. The MSB-first scheme is known as the "Big-Endian" approach, because the bytes are written "from the big end first". The LSB-first scheme is known as "Little-Endian"; bytes are written "from the little end first."

Factoid: Where did the terms "Big-Endian" and "Little-Endian" originate?

Jonathan Swift used these terms when writing about a "holy war" in "Gulliver's Travels" (1726). The Lilliputian ruler declared that all Lilliputians must eat their eggs starting at the "Little End" (because he clumsily cut himself while attempting to eat the egg from the big end). Many Lilliputians rebelled, and took refuge in nearby Blefuscu, where all were required to eat their eggs starting at the "Big End". The "Little-Endians" and "Big-Endians" went to war over the entire silly issue.

Danny Cohen wrote an essay on byte-ordering in 1980, "On Holy Wars and a Plea for Peace". In that essay, he likens the entire issue to the egg war in "Gulliver's Travels", giving us the terms "Big-Endian" and "Little-Endian" for byte ordering. You can read his essay at http://www.op.net/docs/RFCs/ien-137

The problem Endianness causes is that a translation must be performed whenever someone writes a binary file on a Big-Endian platform, and tries to read it on a Little-Endian platform. Currently, most types of platforms, such as the various UNIX machines, are Big-Endian. Some, like the PC, are Little-Endian.

Java is Big-Endian

The Java Virtual Machine *is* a platform, and as such, it needed to choose how it would interpret data.

The choice seemed rather obvious, as Sun, a UNIX machine producer, created the platform.

Java is a Big-Endian platform.

This means if you run a Java program under Solaris, and write several `int`s to the file (using `DataOutputStream`, for example), native Solaris programs can read the `int`s from the file. If you run the same Java program on a PC, *native PC programs cannot **directly** use the data*!

Dealing With Java I/O on Little-Endian Platforms

There are several ways to deal with this issue:

- *Only* have Java programs process the data in question. If the binary data is not needed by any external *native* programs, this is the simplest solution.
  For example, if you are writing a binary configuration file that will be read

only by your application, you have no concerns.

- Write *native* methods to perform *all* input and output for the files in question. Because the native programs "do the right thing on the current platform", you again have no worry. *However*, it is important to note that you will need to deliver a *different* executable for *each* target platform. Also, this solution will not work for unsigned applets.
- If you know the Endianness of the platform *or* the file being read, you could write a translation filter for input or output.
- Use combinations of the above approaches.

For further information and some utilities to assist, look up "Endian" in Roedy Green's Java Glossary (http://mindprod.com/gloss.html

Input/Output Exceptions

Until this point, we have avoided the topic of exception handling in I/O operations. So far, all of our example code has been of the form:

```
try {
  // do some I/O operation(s)
}
catch(Exception e) {
  e.printStackTrace();
}
```

While this simplifies the examples significantly, it is hardly appropriate for real programming.

The Java core libraries define several I/O exceptions, all of which extend a class named `IOException`. `IOException` directly extends `Exception`; that means that it is *not* a "runtime exception". As a result, you must *always* either provide `try-catch` blocks around *any* I/O code, or add a `throws` clause to the method that performs the I/O. Our example above shows a simple try-catch situation, while the following code demonstrates a `throws` clause:

```
public void doSomeIO() throws IOException {
  // do some I/O operation(s)
  // note that we're not in a try-catch block!
}
```

Of course we're now requiring any code that calls `doSomeIO()` to either surround it with a `try-catch` block *or* add a `throws` clause to *that* method.

In some cases it may be perfectly appropriate to "pass the buck" to the caller by adding a `throws` clause. In other cases it's more appropriate to handle the exception and shield the caller from ever knowing something went wrong. This is strictly up to your application design.

The java.io Exceptions

The following is a list of all exceptions defined in package java.io, with a brief description of each. A few of these exceptions will be covered in detail after this section.

`CharConversionException`	A problem occurred while converting a `char` to one or more `byte`s, or vice-versa. This can occur in: `OutputStreamWriter` (note that `FileWriter` subclasses `OutputStreamWriter`!) Converting between `byte[]` to a `String` (in either direction)
`EOFException`	The end of a stream was unexpectedly reached. Normally hitting "end-of-file" is signaled by a special value, such as a -1 return from a `read()` call. However, some streams, like `DataInputStream` might hit end-of-file in the middle of reading a multi-byte quantity, such as a call to `readInt()`. In cases like that, `EOFException` is thrown.
`FileNotFoundException`	A file specified in a constructor to `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`, or `RandomAccessFile` does not exist. Note that this could also mean that the file was being opened in the wrong mode, for example, trying to open a read-only file for write access.
`InterruptedIOException`	An I/O operation was halted because the thread performing the operation was terminated.

`InvalidClassException`	Thrown during a Serialization operation because a serialized object does not match the class that the runtime sees, or necessary types or constructors were not present.
`InvalidObjectException`	Thrown during de-serialization if a de-serialized class fails validation tests. This could happen if the serialized file was corrupted or modified.
`IOException`	The general I/O exception class -- all other I/O exceptions extend this. This exception indicates that something undesirable happened while performing an I/O operartion.
`NotActiveException`	Thrown if serialization support methods are called outside the scope of the valid serialization methods.
`NotSerializableException`	Thrown if an object that is being serialized does not implement `Serializable`. Note that this can also be thrown for non-serializable objects *referenced* by a serializable object.
`ObjectStreamException`	A common class for all exceptions related to serialization.
`OptionalDataException`	Thrown when unexpected data is encountered while trying to de-serialize an object.
`StreamCorruptedException`	Thrown when a serialized object stream is corrupted and no other sense can be made out of it.
`SyncFailedException`	Thrown if a file's output content could not be actually written to the physical file system.
`UnsupportedEncodingException`	A specific character encoding is not supported (if you tried to open a `Reader` or `Writer` with a specific encoding.

`UTFDataFormatException`	An attempt to read a UTF-8 stream failed due to malformed UTF-8 content. This can occur when calling the `readUTF()` method of `DataInputStream` or `RandomAccessFile`.
`WriteAbortedException`	Thrown while *reading* a serialized object stream *if* the stream was only partially written because an `ObjectStreamException` was thrown while *writing* it.

Closing Streams

The most difficult thing to do with regard to I/O is to properly close a stream. There are several subtle problems. The main concerns are:

- where should I put the `close()` call?
- how do I deal with exceptions thrown when opening filters?
- what happens if a filter's `close()` throws an exception?

Let's start with the basic `close()` handling case first. The basic issue is, "where do I put the close() to ensure my file will be really be closed?" A first attempt might look like:

```
// ... CAUTION ...WRONG WAY TO DO THINGS...

try {
  FileReader r = new FileReader("somefile.txt");
  // read some text from the file
  r.close();
}
catch(IOException e) {
  // display a message about the error
}
```

This approach is too simple. The file will be closed *only* if there are *no* exceptions thrown between the file opening and the attempt to close it. A second attempt might be:

```
// ... CAUTION ...WRONG WAY TO DO THINGS...

FileReader r = null;
try {
  r = new FileReader("somefile.txt");
  // read some text from the file
```

```
  r.close();
}
catch(IOException e) {
  // display a message about the error
  if (r != null) {
    try {
      r.close();
    }
    // tried to close but couldn't anyway!
    // should inform the user if the data was important...
    catch(Exception ignoreMe) {}
  }
}
```

Note how the declaration of the `FileReader` has moved outside the `try` block so it can be accessed in the `catch` block. But this is still problematic, for two reasons:

- If there were several catch blocks, the close code would need to be repeated in each, making it easy to make an error or even miss a spot.
- If a non-`IOException` were thrown, the file would not be closed! For example, a `NullPointerException` would cause a problem.

A better approach would be to use a `finally` clause to do the closing:

```
FileReader r = null;
try {
  r = new FileReader("somefile.txt");
  // read some text from the file
  // NOTE: No close() here!
}
catch(IOException e) {
  // display a message about the error
}
finally {
  if (r != null) {
    try {
      r.close();
    }
    // tried to close but couldn't anyway!
    // should inform the user if the data was important...
    catch(Exception ignoreMe) {}
  }
}
```

This ensures that no matter what happens, we at least *try* to close the file.

But what happens if we start using filters, and for some reason a filter does not open completely?

Think about the following code:

```
// ... CAUTION ... WRONG WAY TO DO THINGS ...
Reader r = new BufferedReader(
             new FileReader("somefile.txt"));
```

Suppose `BufferedReader` cannot obtain enough memory to allocate its buffer. It will throw an `OutOfMemoryException` which exits the constructor. But what happens to the `FileReader`? It has been opened!

A safer way to deal with filter nesting is to track *all* streams. For example

```
// A SAFER OPEN/CLOSE SCENARIO

FileReader       fr   = null;
BufferedReader   br   = null;
TabToSpaceReader ttsr = null; // assuming you defined such as
filter...

try {
  ttr = new TabToSpaceReader(
          br = new BufferedReader(
            fr = new FileReader("somefile.txt")));

  // read some text from ttr
}
finally {
  if (ttr != null) {
    try {
      ttr.close();
      br  = null;
      fr  = null;
    }
    catch(Exception ignoreMe) {}
    ttr = null;
  }

  if (br != null) {
    try {
      br.close();
      fr = null;
    }
    catch(Exception ignoreMe) {}
    br = null;
  }

  if (fr != null) {
    try {
      fr.close();
    }
    catch(Exception ignoreMe) {}
    fr = null;
  }
```

```
}
```

This code requires some explanation. The idea is that we keep track of *all* streams created. If they cannot be created successfully, their variables will still be `null` so we won't try to close them. At the end of the I/O operations (in the finally block) we check each stream and try to `close()` it. If the `close()` succeeds, it has also closed any *delegate* steams, so we set *their* variables to null as well. If the `close()` fails, the *delegate* streams will still be non-null and we will attempt to close them in the next block. Of course the order of the closing is extremely important.

One final note: the action to take when a close() fails is completely up to your application design. If the data is important, you should definitely inform the user that there was a problem closing the file. If the data was not important, it may be ok to simply ignore the error as we do in the above example.

[*MML: 1.03*]
[*Version: $ /JavaIO/JavaIO.mml#7 $*]