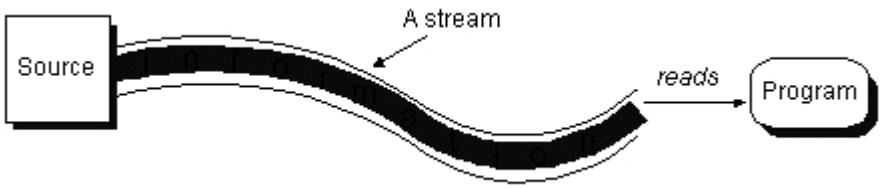
Java Input/Output – Text and Binary Streams

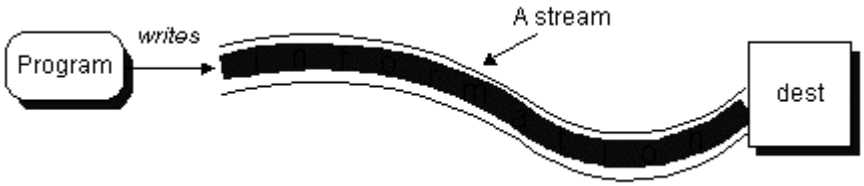
Introduction to Data Streams

Often programs need to bring in information from an external source or send out information to an external destination. The information can be anywhere: in a file, on disk, somewhere on the network, in memory, or in another program. Also, it can be of any type: objects, characters, images, or sounds.

To bring in information, a program opens a stream on an information source (a file, memory, a socket) and reads the information serially, like this:



Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out serially, like this:

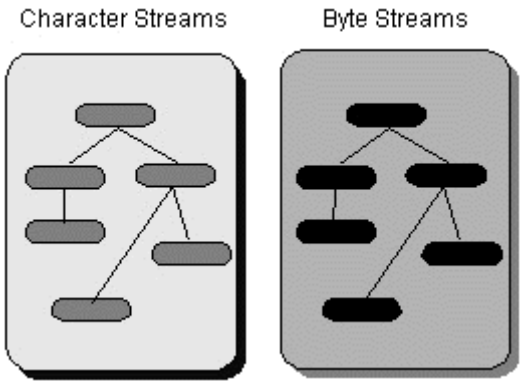


No matter where the information is coming from or going to and no matter what type of data is being read or written, the algorithms for reading and writing data is pretty much always the same.

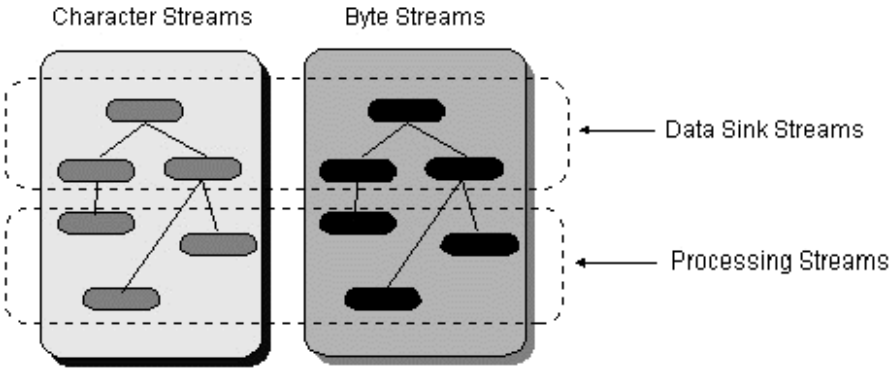
Reading	Writing

`open a stream`
`while more information`
`  read information`
`close the stream`
	`open a stream`
	`while more information`
	`  write information`
	`close the stream`

The `java.io` package contains a collection of stream classes that support these algorithms for reading and writing. These classes are divided into two class hierarchies based on the data type (either characters or bytes) on which they operate.



However, it's often more convenient to group the classes based on their purpose rather than on the data type they read and write. Thus, we can cross-group the streams by whether they read from and write to data "sinks" or process the information as its being read or written.

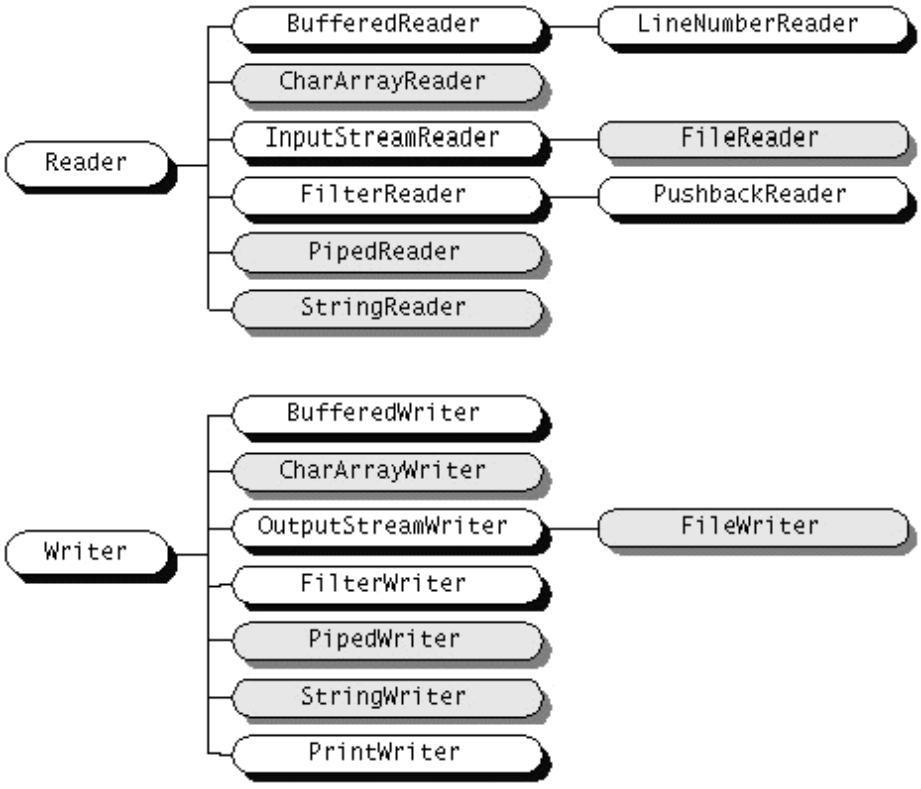


Overview of I/O Streams

Character Streams

`Reader` and `Writer` are the abstract superclasses for character streams in `java.io`. `Reader` provides the API and partial implementation for readers--streams that read 16-bit characters--and `Writer` provides the API and partial implementation for writers--streams that write 16-bit characters.

Subclasses of `Reader` and `Writer` implement specialized streams and are divided into two categories: those that read from or write to data sinks (shown in gray in the following figures) and those that perform some sort of processing (shown in white). The figure shows the class hierarchies for the `Reader` and `Writer` classes.

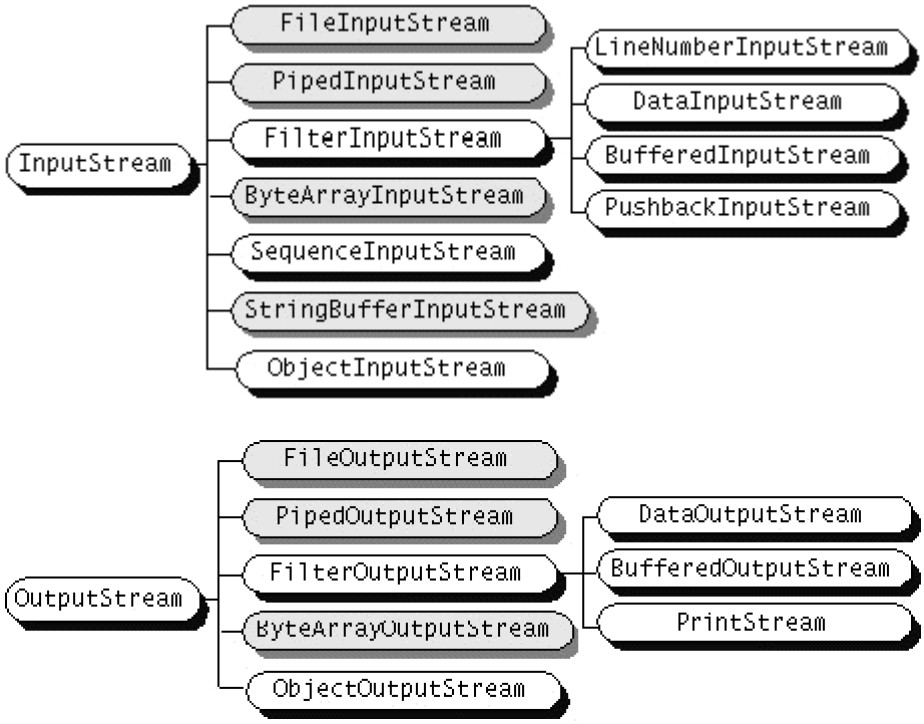


Most programs should use readers and writers to read and write information. This is because they both can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).

Byte Streams (Binary Streams)

Programs should use the byte streams, descendants of `InputStream` and `OutputStream`, to read and write 8-bit bytes. `InputStream` and `OutputStream` provide the API and some implementation for input streams (streams that read 8-bit bytes) and output streams (streams that write 8-bit bytes). These streams are typically used to read and write binary data such as images and sounds.

As with `Reader` and `Writer`, subclasses of `InputStream` and `OutputStream` provide specialized I/O that falls into two categories: data sink streams and processing streams. Figure 56 shows the class hierarchies for the byte streams.



As mentioned, two of the byte stream classes, `ObjectInputStream` and `ObjectOutputStream`, are used for object serialization.

Understanding the I/O Superclasses

`Reader` and `InputStream` define similar APIs but for different data types. For example, `Reader` contains these methods for reading characters and arrays of characters:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

`InputStream` defines the same methods but for reading bytes and arrays of bytes:

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

Also, both `Reader` and `InputStream` provide methods for marking a location in the stream, skipping input, and resetting the current position.

`Writer` and `OutputStream` are similarly parallel. `Writer` defines these methods for writing characters and arrays of characters:

```
void write(int c)
void write(char cbuf[])
void write(char cbuf[], int offset, int length)
void write(String s)
```

And `OutputStream` defines the same methods but for bytes:

```
void write(int c)
void write(byte cbuf[])
void write(byte cbuf[], int offset, int length)
```

All of the streams--readers, writers, input streams, and output streams--are automatically opened when created. You can close any stream explicitly by calling its `close` method. Or the garbage collector can implicitly close it, which occurs when the object is no longer referenced.

How to Use File Streams

File streams are perhaps the easiest streams to understand. Simply put, the file streams - `FileReader`, `FileWriter`, `FileInputStream`, and `FileOutputStream` - each read or write from a file on the native file system. You can create a file stream from a filename, a `File` object, or a `FileDescriptor` object.

The following program uses `FileReader` and `FileWriter` to copy the contents of a file named input.txt into a file called `output.txt` :

```
import java.io.*;

public class Copy {
    public static void main(String[] args) throws IOException {
      File inputFile = new File("input.txt");
      File outputFile = new File("output.txt");

        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;

        while ((c = in.read()) != -1)
           out.write(c);

        in.close();
        out.close();
    }
}
```

This program is very simple. It opens a `FileReader` on `input.txt` and opens a `FileWriter` on `output.txt`. The program reads characters from the reader as long as there's more input in the input file. When the input runs out, the program closes both the reader and the writer.

Notice the code that the `Copy` program uses to create a `FileReader`:

```
File inputFile = new File("input.txt");
FileReader in = new FileReader(inputFile);
```

This code creates a `File` object that represents the named file on the native file system. `File` is a utility class provided by `java.io`. This program uses this object only to construct a `FileReader` on `input.txt`. However, it could use `inputFile` to get information about `input.txt`, such as its full pathname.

`After you've run the program, you should find an exact copy of input.txt in a file named output.txt in the same directory.`

Remember that `FileReader` and `FileWriter` read and write 16-bit characters. However, most native file systems are based on 8-bit bytes. These streams encode the characters as they operate according to the default character-encoding scheme. You can find out the default character-encoding by using `System.getProperty ("file.encoding")`. To specify an encoding other than the default, you should construct an `OutputStreamWriter` on a `FileOutputStream` and specify it.

For the curious, here is another version of this program, CopyBytes, which uses `FileInputStream` and `FileOutputStream` in place of `FileReader` and `FileWriter` :

```
import java.io.*;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("input.txt");
        File outputFile = new File("output.txt");

        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new FileOutputStream(outputFile);
        int c;

        while ((c = in.read()) != -1)
           out.write(c);

        in.close();
        out.close();
    }
}
```

How to Use LineNumberReader

The `LineNumberReader` class is a subclass of `BufferedReader`. Its `read()` methods contain additional logic to count end-of-line characters and thereby maintain a line number. Since different platforms use different characters to represent the end of a line, `LineNumberReader` takes a flexible approach and recognizes `"\n"`, `"\r"`, or `"\r\n"` as the end of a line. Regardless of the end-of-line character it reads, `LineNumberReader` returns only `"\n"` from its `read()` methods.

You can create a `LineNumberReader` by passing its constructor a `Reader`. The following example prints out all the lines of a file, with each line prefixed by its number. If you try this example, you'll see that the line numbers begin at `0` by default:

```
try {
      FileReader fileIn = new FileReader("text.txt");
      LineNumberReader in = new LineNumberReader(fileIn);
      while ((line=in.readLine()) != null) {
            System.out.println(in.getLineNumber() + ". " + line);
      }
} catch (IOException ioe) {
      ioe.printStackTrace();
}
```

The `LineNumberReader` class has two methods pertaining to line numbers. The `getLineNumber()` method returns the current line number. If you want to change the current line number of a `LineNumberReader`, use `setLineNumber()`. This method does not affect the stream position; it merely sets the value of the line number.

How to Use PrintWriter

The `PrintWriter` class is a subclass of `Writer` that provides a set of methods for printing string representations of every Java data type. A `PrintWriter` can be wrapped around an underlying `Writer` object or an underlying `OutputStream` object. In the case of wrapping an `OutputStream`, any characters written to the `PrintWriter` are converted to bytes using the default encoding scheme.[2] Additional constructors allow you to specify if the underlying stream should be flushed after every line-separator character is written.

The `PrintWriter` class provides a `print()` and a `println()` method for every primitive Java data type. As their names imply, the `println()` methods do the same thing as their `print()` counterparts, but also append a line separator character.

The following example demonstrates how to wrap a `PrintWriter` around an `OutputStream`:

```
boolean b = true;
char c = '%'
double d = 8.31451
int i = 42;
String s = "R = ";
PrintWriter out = new PrintWriter(System.out, true);
out.print(s);
out.print(d);
out.println();
out.println(b);
out.println(c);
out.println(i);
```

This example produces the following output:

```
R = 8.31451
true
%
42
```

`PrintWriter` objects are often used to report errors. For this reason, the methods of this class do not throw exceptions. Instead, the methods catch any exceptions thrown by any downstream `OutputStream` or `Writer` objects and set an internal flag, so that the object can remember that a problem occurred. You can query the internal flag by calling the `checkError()` method.

Although you can create a `PrintWriter` that flushes the underlying stream every time a line-separator character is written, this may not always be exactly what you want. Suppose that you are writing a program that has a character-based user interface, and that you want the program to output a prompt and then allow the user to input a response on the same line. In order to make this work with a `PrintWriter`, you need to get the `PrintWriter` to write the characters in its buffer without writing a line separator. You can do this by calling the `flush()` method.

How to Use DataInputStream and DataOutputStream

This page shows you how to use the `java.io` DataInputStream and DataOutputStream classes. It features an example, DataIOTest, that reads and writes tabular data (invoices for Java merchandise) :

```
import java.io.*;

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

        // write the data out
        DataOutputStream out = new DataOutputStream(new
            FileOutputStream("invoice1.txt"));

        double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
        int[] units = { 12, 8, 13, 29, 50 };
        String[] descs = { "Java T-shirt",
                           "Java Mug", "Duke Juggling Dolls",
                           "Java Pin", "Java Key Chain" };

        for (int i = 0; i < prices.length; i ++) {
            out.writeDouble(prices[i]);
            out.writeChar('\t');
            out.writeInt(units[i]);
            out.writeChar('\t');
            out.writeChars(descs[i]);
            out.writeChar('\n');
        }
        out.close();
```

```
        // read it in again
        DataInputStream in = new DataInputStream(new
             FileInputStream("invoice1.txt"));

        double price;
        int unit;
        String desc;
        double total = 0.0;

        try {
            while (true) {
                price = in.readDouble();
                in.readChar();       // throws out the tab
                unit = in.readInt();
                in.readChar();       // throws out the tab
                desc = in.readLine();
                System.out.println("You've ordered " + unit + " units of " +
                      desc + " at $" + price);
                total = total + unit * price;
            }
        } catch (EOFException e) {
        }
        System.out.println("For a TOTAL of: $" + total);
        in.close();
    }
}
```

The tabular data is formatted in columns, where each column is separated from the next by tabs. The columns contain the sales price, the number of units ordered, and a description of the item, like this:

```
19.99   12   Java T-shirt
9.99    8    Java Mug
```

`DataOutputStream`, like other filtered output streams, must be attached to some other `OutputStream`. In this case, it's attached to a `FileOutputStream` that's set up to write to a file named `invoice1.txt`.

```
DataOutputStream dos = new DataOutputStream(new FileOutputStream("invoice1.txt"));
```

Next, `DataIOTest` uses `DataOutputStream`'s specialized `write`*XXX* methods to write the invoice data (contained within arrays in the program) according to the type of data being written:

```
for (int i = 0; i < prices.length; i ++) {
    dos.writeDouble(prices[i]);
    dos.writeChar('\t');
    dos.writeInt(units[i]);
    dos.writeChar('\t');
    dos.writeChars(descs[i]);
    dos.writeChar('\n');
}
dos.close();
```

Note that this code snippet closes the output stream when it's finished.
Next, `DataIOTest` opens a `DataInputStream` on the file just written:

```
DataInputStream dis = new DataInputStream(new FileInputStream("invoice1.txt"));
```

`DataInputStream` also must be attached to some other `InputStream`; in this case, a `FileInputStream` set up to read the file just written - `invoice1.txt`. `DataIOTest` then just reads the data back in using `DataInputStream`'s specialized `read`*XXX* methods.

```
try {
      while (true) {
            price = dis.readDouble();
```

```
            dis.readChar();       // throws out the tab
            unit = dis.readInt();
            dis.readChar();       // throws out the tab
            desc = dis.readLine();
            System.out.println("You've ordered " + unit + " units of " + desc
                  + " at $" + price);
            total = total + unit * price;
      }
}
catch (EOFException e) {
}
System.out.println("For a TOTAL of: $" + total);
dis.close();
```

When all of the data has been read, `DataIOTest` displays a statement summarizing the order and the total amount owed, and closes the stream.

Note the loop that `DataIOTest` uses to read the data from the `DataInputStream`. Normally, when reading you see loops like this:

```
while ((input = dis.readLine()) != null)
{
      . . .
}
```

The `readLine` method returns a value, null, that indicates that the end of the file has been reached. Many of the `DataInputStream` `read`*XXX* methods can't do this because any value that could be returned to indicate end-of-file may also be a legitimate value read from the stream. For example, suppose that you wanted to use -1 to indicate end-of-file? Well, you can't because -1 is a legitimate value that can be read from the input stream using `readDouble`, `readInt`, or one of the other read methods that reads numbers. So `DataInputStream`'s `read`*XXX* methods throw an `EOFException` instead. When the `EOFException` occurs the `while (true)` terminates.

When you run the `DataIOTest` program you should see the following output:

```
You've ordered 12 units of Java T-shirt at $19.99
You've ordered 8 units of Java Mug at $9.99
You've ordered 13 units of Duke Juggling Dolls at $15.99
You've ordered 29 units of Java Pin at $3.99
You've ordered 50 units of Java Key Chain at $4.99
For a TOTAL of: $892.88
```

File Manipulation

While streams are used to handle most types of I/O in Java, there are some nonstream-oriented classes in `java.io` that are provided for file manipulation. Namely, the `File` class represents a file on the local filesystem, while the `RandomAccessFile` class provides nonsequential access to data in a file. In addition, the `FilenameFilter` interface can be used to filter a list of filenames.

File

The `File` class represents a file on the local filesystem. You can use an instance of the `File` class to identify a file, obtain information about the file, and even change information about the file. The easiest way to create a `File` is to pass a filename to the `File` constructor, like this:

```
new File("readme.txt")
```

Although the methods that the `File` class provides for manipulating file information are relatively platform independent, filenames must follow the rules of the local filesystem. The `File` class does provide some information that can be helpful in interpreting filenames and path specifications. The variable `separatorChar`

specifies the system-specific character used to separate the name of a directory from what follows. In a Windows environment, this is a backslash (\), while in a UNIX or Macintosh environment it is a forward slash (/). File separator can be obtained as System.getProperty('file.separator'), which is how the File class gets it. You can create a `File` object that refers to a file called `readme.txt` in a directory called `myDir` as follows:

```
new File("myDir" + File.separatorChar + "readme.txt")
```

The `File` class also provides some constructors that make this task easier. For example, there is a `File` constructor that takes two strings as arguments: the first string is the name of a directory and the second string is the name of a file. The following example does the exact same thing as the previous example:

```
new File("myDir", "readme.txt")
```

The `File` class has another constructor that allows you to specify the directory of a file using a `File` object instead of a `String`:

```
File dir = new File("myDir");
File f = new File(dir, "readme.txt");
```

Sometimes a program needs to process a list of files that have been passed to it in a string. For example, such a list of files is passed to the Java environment by the `CLASSPATH` environment variable and can be accessed by the expression:

```
System.getProperty("java.class.path")
```

This list contains one or more filenames separated by separator characters. In a Windows or Macintosh environment, the separator character is a semicolon (`;`), while in a UNIX environment, the separator character is a colon (`:`). The system-specific separator character is specified by the `pathSeparatorChar` variable. Thus, to turn the value of `CLASSPATH` into a collection of `File` objects, we can write:

```
StringTokenizer s;
Vector v = new Vector();
s = new StringTokenizer(System.getProperty("java.class.path"), File.pathSeparator);
while (s.hasMoreTokens())
      v.addElement(new File(s.nextToken()));
```

You can retrieve the pathname of the file represented by a `File` object with `getPath()`, the filename without any path information with `getName()`, and the directory name with `getParent()`.

The `File` class also defines methods that return information about the actual file represented by a `File` object. Use `exists()` to check whether or not the file exists. `isDirectory()` and `isFile()` tell whether the file is a file or a directory. If the file is a directory, you can use `list()` to get an array of filenames for the files in that directory. The `canRead()` and `canWrite()` methods indicate whether or not a program is allowed to read from or write to a file. You can also retrieve the length of a file with `length()` and its last modified date with `lastModified()`.

A few `File` methods allow you to change the information about a file. For example, you can rename a file with `rename()` and delete it with `delete()`. The `mkdir()` and `mkdirs()` methods provide a way to create directories within the filesystem.

Many of these methods can throw a `SecurityException` if a program does not have permission to access the filesystem, or particular files within it. If a `SecurityManager` has been installed, the `checkRead()` and `checkWrite()` methods of the `SecurityManager` verify whether or not the program has permission to access the filesystem.

FilenameFilter

The purpose of the `FilenameFilter` interface is to provide a way for an object to decide which filenames should be included in a list of filenames. A class that implements the `FilenameFilter` interface must define a method called `accept()`. This method is passed a `File` object that identifies a directory and a `String` that names a file. The `accept()` method is expected to return `true` if the specified file should be included in the list, or `false` if the file should not be included. Here is an example of a simple `FilenameFilter` class that only allows files with a specified suffix to be in a list:

```
import java.io.File;
import java.io.FilenameFilter;
public class SuffixFilter implements FilenameFilter {
    private String suffix;
    public SuffixFilter(String suffix) {
        this.suffix = "." + suffix;
    }
    public boolean accept(File dir, String name) {
        return name.endsWith(suffix);
    }
}
```

A `FilenameFilter` object can be passed as a parameter to the `list()` method of `File` to filter the list that it creates. You can also use a `FilenameFilter` to limit the choices shown in a `FileDialog`.

RandomAccessFile

The `RandomAccessFile` class provides a way to read from and write to a file in a nonsequential manner. The `RandomAccessFile` class has two constructors that both take two arguments. The first argument specifies the file to open, either as a `String` or a `File` object. The second argument is a `String` that must be either `"r"` or `"rw"`. If the second argument is `"r"`, the file is opened for reading only. If the argument is `"rw"`, however, the file is opened for both reading and writing. The `close()` method closes the file. Both constructors and all the methods of the `RandomAccessFile` class can throw an `IOException` if they encounter an error.

The `RandomAccessFile` class defines three different `read()` methods for reading bytes from a file. The `RandomAccessFile` class also implements the `DataInput` interface, so it provides additional methods for reading from a file. Most of these additional methods are related to reading Java primitive types in a machine-independent way. Multibyte quantities are read assuming the most significant byte is first and the least significant byte is last. All of these methods handle an attempt to read past the end of file by throwing an `EOFException`.

The `RandomAccessFile` class also defines three different `write()` methods for writing bytes of output. The `RandomAccessFile` class also implements the `DataOutput` interface, so it provides additional methods for writing to a file. Most of these additional methods are related to writing Java primitive types in a machine-independent way. Again, multibyte quantities are written with the most significant byte first and the least significant byte last.

The `RandomAccessFile` class would not live up to its name if it did not provide a way to access a file in a nonsequential manner. The `getFilePointer()` method returns the current position in the file, while the `seek()` method provides a way to set the position. Finally, the `length()` method returns the length of the file in bytes.