A Brief Tutorial on Java File I/O

Introduction

This document is intended to be a brief, practical tutorial on how to perform input and output on text files, binary files, and random access files, using the Java language. It is not intended to be a complete and comprehensive discussion, but rather is intended to focus on practical application and solving common problems.

In a *text file*, the data is stored as Unicode characters. In applications, the fields and records of a text file are often stored as records and they may be separated by delimiters like tabs and newline characters. When information is written to a text file, the information is *encoded* from its text form to Unicode values. When information is read from a text file, the Unicode values are *decoded* back to their character equivalents.

In a *binary file*, the data is stored in a certain number of bytes, depending upon the type of variable used to reference the data. Integers would be stored in 4-bytes, doubles in 8-bytes, etc. No encoding or decoding is required to be performed.

Java uses *streams* to handle I/O operations, as you have learned in your reading and lectures. The easiest way to think about a stream is as a mechanism that deals with the flow of data from one location to another…like from a program to a disk file, or vice versa. When you work with text I/O, you use a *character stream*, and when you work with binary I/O you use a *binary stream*. Java provides numerous classes that implement streams, and for most programming applications you just need to know what streams to use, how to create them, and what methods can be called.

<u>The File Class</u>

The File class is used when we do file input and output. The File class contains methods for obtaining the properties of files and directories, as well as methods for creating, deleting, and renaming them. The File class does not contain any methods for writing or reading information to/from files and directories. The File class is in the java.io package.

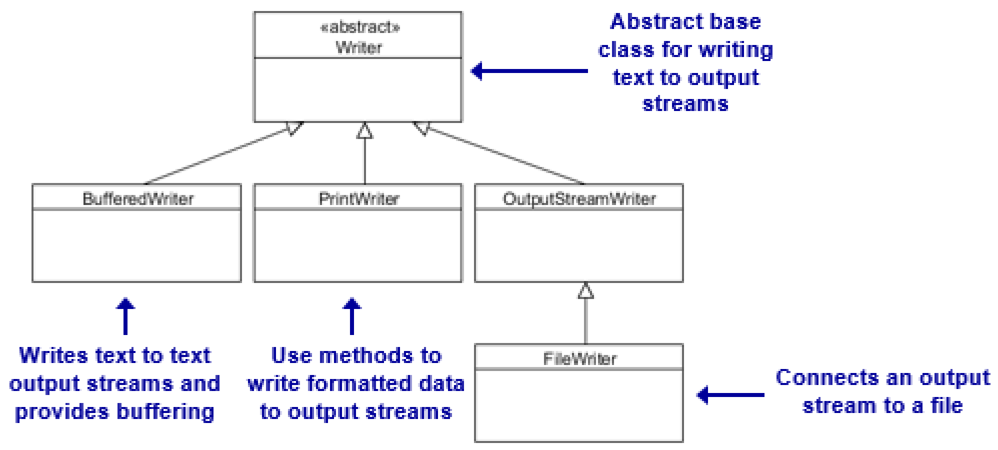
The File class contains almost two dozen methods. Some of the most commonly used methods are listed in the table below.

File(*pathname*: String)	A simple constructor method, where *pathname* is the pathname to the file to be used.
exists(): boolean	Returns *true* if the file/directory exists.
canRead(): boolean	Returns *true* if the file/directory exists and can be read.
canWrite(): boolean	Returns *true* if the file/directory exists and can be written to.
isDirectory(): boolean	Returns *true* if the file object is a directory.
isFile(): boolean	Returns *true* if the file object is an ordinary file.

An example of creating a file object follows, just to illustrate the mechanics.

```
File f = new File( "mytext.txt" );
if ( f.exists() )
        System.out.println( "The file already exists." );
else
        System.out.println( "The file does not exist." );
```

Writing Text Files


«abstract» Writer
Abstract base class for writing text to output streams
BufferedWriter
PrintWriter
OutputStreamWriter
FileWriter
Writes text to text output streams and provides buffering
Use methods to write formatted data to output streams
Connects an output stream to a file

There are a number of classes that can be used when writing to text files. All of these classes are subclasses of the abstract Writer class, as illustrated in the above inheritance hierarchy.

The PrintWriter class contains methods that can be used to write formatted data. The BufferedWriter class can be used to buffer output for more efficiency. And, the FileWriter class can be used to write basic character output. The FileWriter class also allows data to be appended to an existing file.

The PrintWriter Class

One of the ways you can write data to a text file is to use the PrintWriter class. PrintWriter makes it easy to write formatted data. It has a number of useful methods, the most common of which are listed in the table below.

PrintWriter(*pathname*: String)	A simple constructor method, where *pathname* is the pathname to the file to be written.
PrintWriter(*file*: File)	Same as the above constructor, but a File object is specified.
print(*argument*): void	Writes an *argument* to the file, where the *argument* can be any primitive type or an array of type char.
println(*argument*): void	Same as print(), but also prints a line separator.
printf(*argument*): void	Formatted print. Works just like the printf to the standard output

	display.
close(): void	Closes the PrintWriter and flushes output buffer.

Here are some simple examples of using a PrintWriter.

```
PrintWriter output1 = new PrintWriter( “mytext.txt” );

File f = new File( “mytext2.txt” );

PrintWriter output2  = new PrintWriter( “mytext2.txt” );

output1.print( “hello world “ );     // write a String to the file

output1.print( 24.45 );              // write a numeric value

output1.close();
```

The following program will demonstrate how a simple text file can be created. Let’s assume that the file will contain some information about parts in our store’s inventory. A part will have a part number, a part name, and a part price. Each line of the text file will contain the data for a single part.

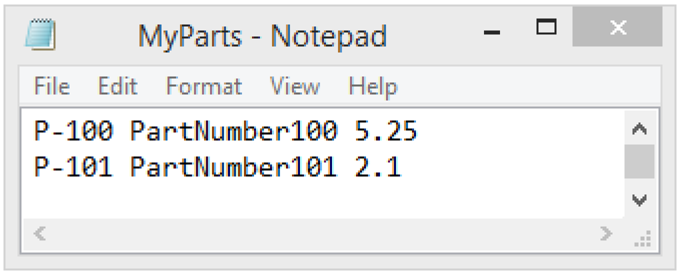
```
// This program demonstrates writing a simple text file
import java.io.*;
import java.util.Scanner;

public class TextWriteDemo1
{
  public static void main( String [] args ) throws IOException
  {
    Scanner keyboardInput = new Scanner( System.in );
    System.out.println( "Enter the name of the text file (with .txt): " );
    String fileName = keyboardInput.next();
    File f = new File( fileName );

    // test if the file already exists
    if( f.exists() )
    {
      System.out.println( "File already exists. Program ending." );
      System.exit( 1 );
    }
    else
      System.out.println( "New file being created." );

    // write records for two parts to the file & close it

    PrintWriter output = new PrintWriter( f );
    output.print( "P-100 " ); output.print( "PartNumber100 " );
    output.println( 5.25 );
    output.print( "P-101 " ); output.print( "PartNumber101 " );
    output.println( 2.10 );

    output.close();
  }
}
```

The first thing to note about this program is that the main method throws an IOException. The reason for this is that it is possible that an IOException may be thrown when a PrintWriter object is created. As a result, Java will force us to try and handle such an exception. One option for doing this is to use try and catch blocks inside the main method. A simpler option for our purposes will be to specify that the main method may throw such an exception. This will allow us to compile the program without having to build exception handling code into the main method, and will keep the code shorter for demonstration purposes so that we can focus on file writing mechanics.

The program uses a Scanner to prompt the user to enter the name of the file to be created. It then creates a File object for the file. The program then tests whether the file already exists. If it does exist, the program will terminate so that the file contents are not overwritten. If the file does not exist, the program will write two part records to the file. Note that the print() method is used to write the first two pieces of part information and the println() method is used for the last piece. This is so that a line separator is added to the file so that the part information for each part is on a separate line.

Finally, the file is closed using the close() method.

The file created by the program looks like this when loaded into Windows Notepad:



```
P-100 PartNumber100 5.25
P-101 PartNumber101 2.1
```

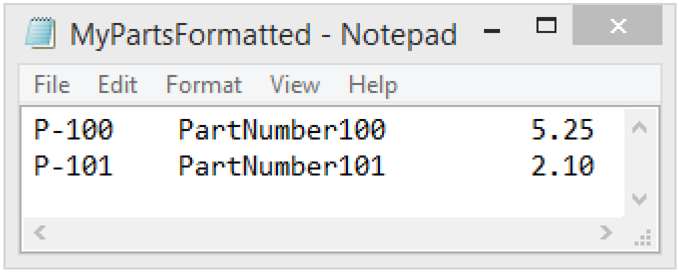
Note that the data elements for each part are separated by white space, since we want white space to be the field delimiter. You may want to look at the print statements in the program to see how the field delimiter was specified.

Sometimes it is useful to set up a text file so that fields of data are structured in a very consistent way. Using the printf() method is an easy way to do this. As an example, suppose we wanted to set up our parts inventory file so that the part number was contained in a field width of 8, the part name was contained in a field width of 15, and the part price was contained in a field width of 10 with two values after the decimal point. The following statements will produce the desired result:

```
output.printf( "%-8s %-15s %10.2f %n", "P-100", "PartNumber100", 5.25 );
output.printf( "%-8s %-15s %10.2f %n", "P-101", "PartNumber101", 2.10 );
```

The hyphen (-) in the string formats indicates that the respective objects will be left-justified in their respective fields. The %n in the format strings specifies that a line separator will be written.

If these statements were substituted for the print() statements in the TextWriteDemo1 program, a file with the necessary field format will be produced. As an exercise, you might want to try this for yourself.



```
P-100     PartNumber100          5.25
P-101     PartNumber101          2.10
```

When loaded into Windows Notepad, the file looks like the above.

Wrapping Output Streams

In practice, it is common to use multiple Write subclasses. This allows us to take advantage of special features of each class, like the ability to append to existing files and the ability to use buffered streams for increased efficiency.

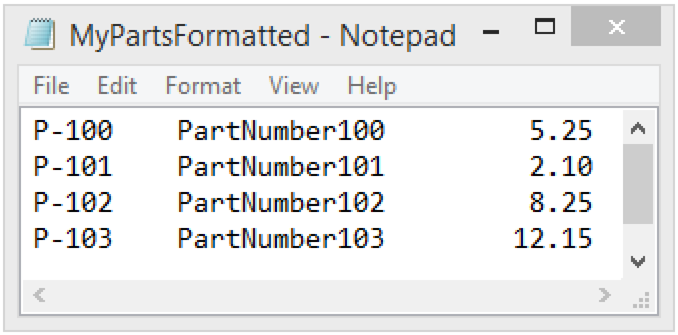
As an example, let's write a program that appends additional parts to the parts inventory file created by the TextDemo2 program. The FileWriter class has a constructor that opens a file and allows data to be appended. We'll wrap that class in a PrintWriter class so that we can append and do formatted output to the file. This is accomplished by the following statement:

```
      PrintWriter output = new PrintWriter( new FileWriter( f, true) );
```

The "true" argument in the FileWriter constructor in the above statement allows data to be appended to the file. FileWriter has numerous consructors that are not demonstrated in this tutorial for the sake of brevity. The complete program follows.

```
// This program demonstrates writing a simple text file using
// FileWriter and PrintWriter

import java.io.*;
import java.util.Scanner;

public class TextWriteDemo3
{
  public static void main( String [] args ) throws IOException
  {
    Scanner keyboardInput = new Scanner( System.in );
    System.out.println( "Enter the name of the text file (with .txt): " );
    String fileName = keyboardInput.next();
    File f = new File( fileName );

    // test if the file already exists
    if( f.exists() )
      System.out.println( "File already exists. Data will be appended." );

    else
      System.out.println( "New file being created." );
    // Create a PrintWriter by wrapping it around a FileWriter
    PrintWriter output = new PrintWriter( new FileWriter( f, true) );
    output.printf( "%-8s %-15s %10.2f %n", "P-102", "PartNumber102", 8.25 );
    output.printf( "%-8s %-15s %10.2f %n", "P-103", "PartNumber103", 12.15 );

    output.close();
  }
}
```

Note that, unlike TextWriteDemo1 and TextWriteDemo2, this program will not terminate if the user-specified file already exists, since the file will be opened for appending. If the file does not exist, it will be created.



```
P-100    PartNumber100         5.25
P-101    PartNumber101         2.10
P-102    PartNumber102         8.25
P-103    PartNumber103        12.15
```

Now, the formatted parts file looks like the above.

For efficiency, we can also wrap a PrintWriter around a BufferedWriter, as follows:

```
    PrintWriter output = new PrintWriter(
                         new BufferedWriter(
                         new FileWriter( f, true) ) );
```

Reading Text Files

To read a text file, we have a number of options. One option is to use the Scanner class. You've already been using Scanner to read from the keyboard, and you can also use it to read from other sources. Another option is to use one or more subclasses of the abstract Reader class.

Let's start with using the Scanner class. We have been using the Scanner class throughout the course to read user input from the keyboard, and have been creating Scanner objects like this:

```
Scanner input = new Scanner( System.in );
```

To use a Scanner to get input from a file, you can create a Scanner object like this:

```
Scanner input = new Scanner( new File( filename ) );
```

The same methods we've been using to get keyboard input can be used to input data from a text file.

hasNext() : boolean	Returns *true* if there is more data to be read.
next(): String	Returns the next token as a String.
nextLine(): String	Returns a line ending with the line separator.
nextByte(): byte	Returns the next token as a byte type.
…	…
nextDouble(): double	Returns the next token as a double type.
useDelimiter(pattern: String): Scanner	Sets the Scanner delimiting pattern and returns the Scanner.

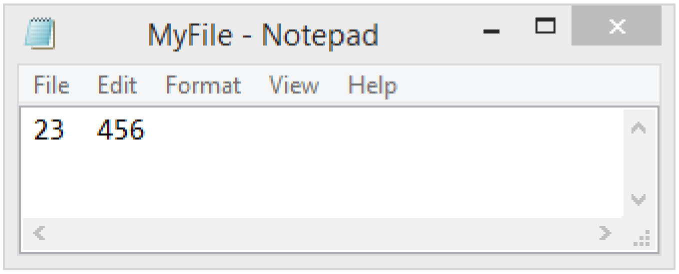
A Scanner breaks its input into a number of *tokens*, based on its delimiter pattern, which, by default, is whitespace. The tokens can then be converted into values of the various Java primitive types by using the "next" methods, which are sometimes called token-reading methods. It is important to understand how the token-reading methods work, especially when a file contains mixed data types.

A token-reading method skips over any leading delimiters, and then reads a token ending at a delimiter. The token is then automatically converted to the required primitive type (e.g., nextInt() converts a token to an integer type).

In the case of the next() method, no conversion is performed…i.e., the token is returned as a String type.

If the nextLine() method is invoked after a token-reading method, it reads characters that start from the current delimiter and end with the line separator. The separator is read, but it is not part of the String returned by nextLine().

Let's illustrate a few examples that compare reading data from a file versus from the keyboard. Suppose we had a text file that looks as illustrated below. Please note that there are two blank spaces that delimit the values.



```
23  456
```

Here's a code segment that reads the file using a Scanner:

```
Scanner input = new Scanner( new File( "MyText.txt" ) );

int x = input.nextInt();              // x is 23
String y = input.nextLine();          // y is [  456]
```

In the above code, the nextInt() method reads up to the first whitespace character…the blank space after the 3. The string 23 is then converted to the integer value 23. The nextLine() method then reads from the first whitespace character to the end of the line. It returns a string with two blank spaces followed by 456. The brackets are used in the comments to help visualize the blank spaces.

Now, here's a code segment that reads from the keyboard using a Scanner:

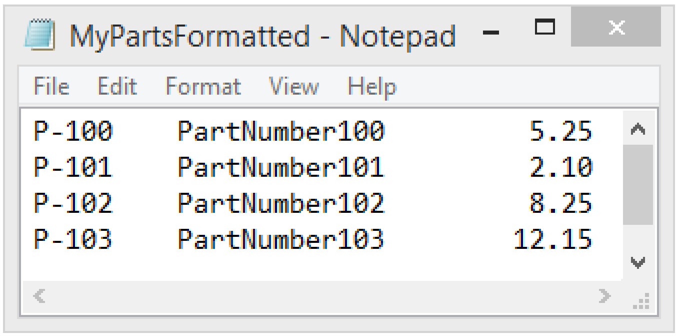
```
Scanner input = new Scanner( System.in );

int x = input.nextInt();              // x is 23
String y = input.nextLine();          // y is empty []
```

We will assume that the user inputs 23 followed by the enter key, then inputs 456. In this case, the nextInt() method reads up to the line separator (\n) and converts the string 23 to its integer value. Imagine there is a cursor in the input stream pointing to the line separator. The nextLine() method then reads up to the line separator and returns the string characters before the line separator. Because it is starting at the line separator…and there are no characters before the separator…y is empty. To further illustrate what's going on, suppose we had another call to nextLine() as illustrated in the code segment below.

```
Scanner input = new Scanner( System.in );

int x = input.nextInt();              // x is 23
String y = input.nextLine();          // y is empty []
String z = input.nextLine();          // z is [456]
```

The first call to nextLine() reads the line separator and returns an empty string. Our imaginary cursor is now pointing to the 4. The second call to nextLine() reads up to the next line separator and returns the string 456.

Let's revisit the parts inventory text file that was discussed earlier in this tutorial. To refresh our memory, we'll assume that the file looks as follows:



```
P-100    PartNumber100     5.25
P-101    PartNumber101     2.10
P-102    PartNumber102     8.25
P-103    PartNumber103    12.15
```

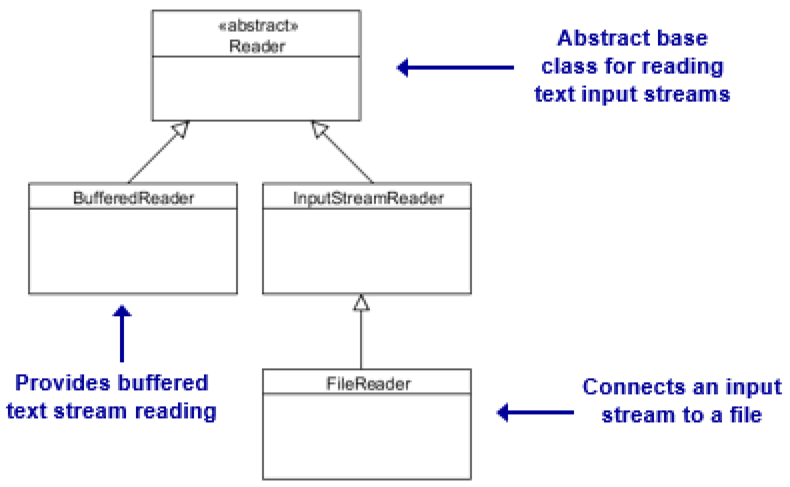
We're going to use a Scanner to read the file and display the information on the standard output. The first two data elements in each line are String types and the third data element is a float type.

Our program will prompt the user to enter the file to be read. If the file doesn't exist, the program will terminate. If the file exists, a Scanner will be created to read the file, and the data will be read and displayed.

```java
// This program demonstrates reading a simple text file
// using a Scanner

import java.io.*;
import java.util.Scanner;

public class TextReadDemo1
{
  public static void main( String [] args ) throws IOException
  {
    Scanner keyboardInput = new Scanner( System.in );
    System.out.println( "Enter the name of the text file (with .txt): " );
    String fileName = keyboardInput.next();
    File f = new File( fileName );

    // test if the file already exists
    if( !f.exists() )
    {
      System.out.println( "Does not exist. Program ending." );
      System.exit( 1 );
    }
    else
      System.out.println( "File will be read." );

    // read records for two parts from the file & close it
    Scanner input = new Scanner( f );
    while ( input.hasNext() )
    {
      String partNumber = input.next();
      String partName = input.next();
      float partPrice = input.nextFloat();

      System.out.println( partNumber + "\t" + partName + "\t" + partPrice );
    }

    input.close();
  }
}
```

In the above program, note that the next() method is used to read the String data elements, and the nextFloat() method is used to read the float data element. The hasNext() method will return *true* as long as there is more data to be read.

Another way to read a text file is to use subclasses of the abstract Reader class. The basic inheritance hierarchy is illustrated below.


«abstract» Reader
Abstract base class for reading text input streams
BufferedReader
InputStreamReader
Provides buffered text stream reading
FileReader
Connects an input stream to a file

One simple approach is to use the BufferedReader and FileReader classes. The FileReader class has several very simple methods, some of which are illustrated in the table below.

FileReader(*pathname*: String)	A simple constructor method, where *pathname* is the pathname to the file to be used.
FileReader(*filename*: File)	A constructor method, where *filename* is the name of a File object.
read(): int	Reads a single character and returns its ASCII integer value. Returns -1 on an end of file condition.
skip(*n*: long): long	Skips n characters in the input stream and returns the value for the actual number of characters skipped.
close(): void	Closes the stream and flushes the buffer.

A FileReader can read a single character at a time from a text file. This is okay if that's all you want to do. But, if a text file has specific data elements, like in the parts inventory file we've been using, then this class provides limited usefulness. Because of this, and for the sake of brevity, using only a FileReader to read and display a file won't be illustrated in this tutorial.

A FileReader can be wrapped by a BufferedReader class to achieve more efficiency and to enable easy reading of lines of text. The BufferedReader class contains a readLine() method that reads a line of text and returns it as a String type. It returns *null* if there is no more data to read.

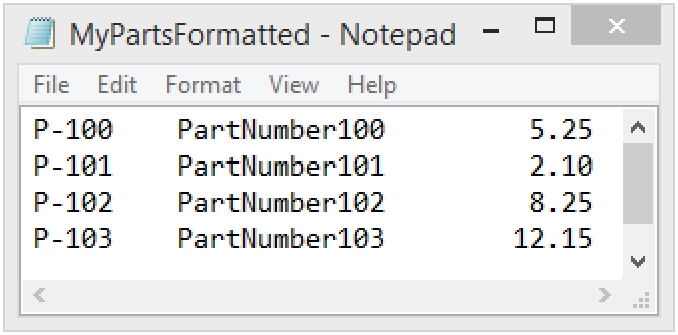
The program below demonstrates how to use a BufferedReader and FileReader class to input and display records from our parts inventory file.

```
// This program demonstrates reading a simple text file
// using a BufferedReader and FileReader

import java.io.*;
import java.util.Scanner;

public class TextReadDemo3
{
  public static void main( String [] args ) throws IOException
  {
    Scanner keyboardInput = new Scanner( System.in );
    System.out.println( "Enter the name of the text file (including .txt): "
);
    String fileName = keyboardInput.next();
    File f = new File( fileName );

    // test if the file already exists
    if( !f.exists() )
    {
      System.out.println( "Does not exist. Program ending." );
      System.exit( 1 );
    }
    else
      System.out.println( "File will be read." );

    // read records from the file & close it
    BufferedReader input = new BufferedReader( new FileReader( f ) );
    String record = null;

    while( ( record = input.readLine() ) != null )
      System.out.println( record );

    input.close();
  }
}
```

Understanding how this program works should be pretty straightforward. A BufferedReader that wraps a FileReader is created, and reads each line of text in the file until an end of file condition occurs.

In one of the course assignments, you were required to read and write information to text files in which there were multiple fields of information, and the values in some of the fields needed to be summed before being output. There are numerous that this can be accomplished, but we'll illustrate one way using the last example as a starting point, and use the parts inventory file.

In order to parse a line of text from an input file into a set of data elements, it really helps if you know the file format structure. Recall that earlier in this tutorial we had constructed a program that would create a formatted parts inventory file that looked like this:



```
P-100    PartNumber100          5.25
P-101    PartNumber101          2.10
P-102    PartNumber102          8.25
P-103    PartNumber103         12.15
```

The file was created using a format control string like this: `"%-8s %-15s %10.2f %n"`

If we treat each line in the file as a record of information about a particular part, then the above format control string sets up a file format that contains record lengths of 36 characters. You should be able to verify this just by looking at the above string. The first field will have a length of 8 characters, the second field will have a length of 15 characters, and the third field will have a length of 10 characters. Note that there is a blank space after each of the elements, so adding 3 more characters yields a record length of 36.

So, starting with the TextReadDemo3 program, we have a String variable, named *record*, that holds the data for a line of text. We can use the String class methods to parse that string and extract the information for the part number, part name, and part price. The substring() method is perfect for this, as illustrated below:

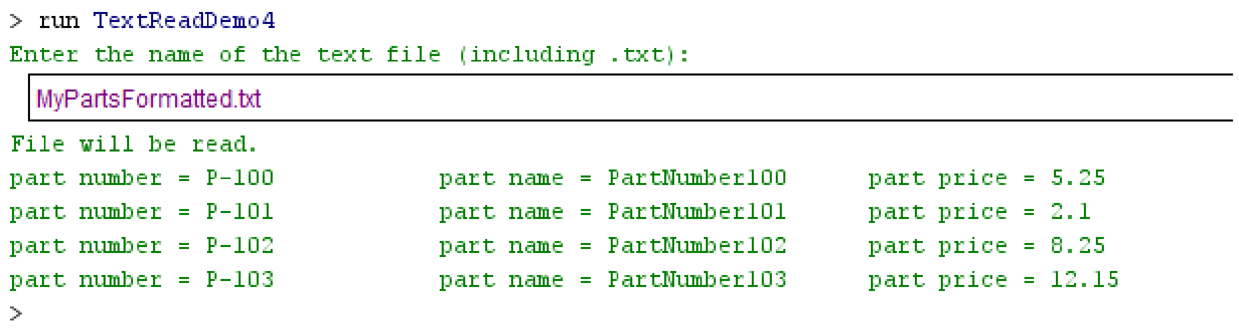
```
    String partNumber  = record.substring( 0, 8 );
    String partName = record.substring(9, 24 );
    float partPrice = Float.parseFloat(  record.substring( 25, 35 ) );
```

Note that the parseFloat() method was used in the last statement above to convert the string representation of the part price to a floating point type.

The program segment below illustrates reading the parts inventory text file and displaying each data element. Only the essential code is shown for the sake of brevity. Otherwise, the program is identical to TextReadDemo3.

```
// read records from the file & close it
    BufferedReader input = new BufferedReader( new FileReader( f ) );
    String record = null;

    while( ( record = input.readLine() ) != null )
    {
      String partNumber = record.substring( 0, 8 );
      String partName = record.substring( 9, 24 );
      float partPrice = Float.parseFloat( record.substring( 25, 35 ) );

      System.out.println( "part number = " + partNumber + "\tpart name = " +
                           partName + "\tpart price = " + partPrice );
    }
    input.close();
```

Here’s the output of the program when run from within the DrJava IDE:



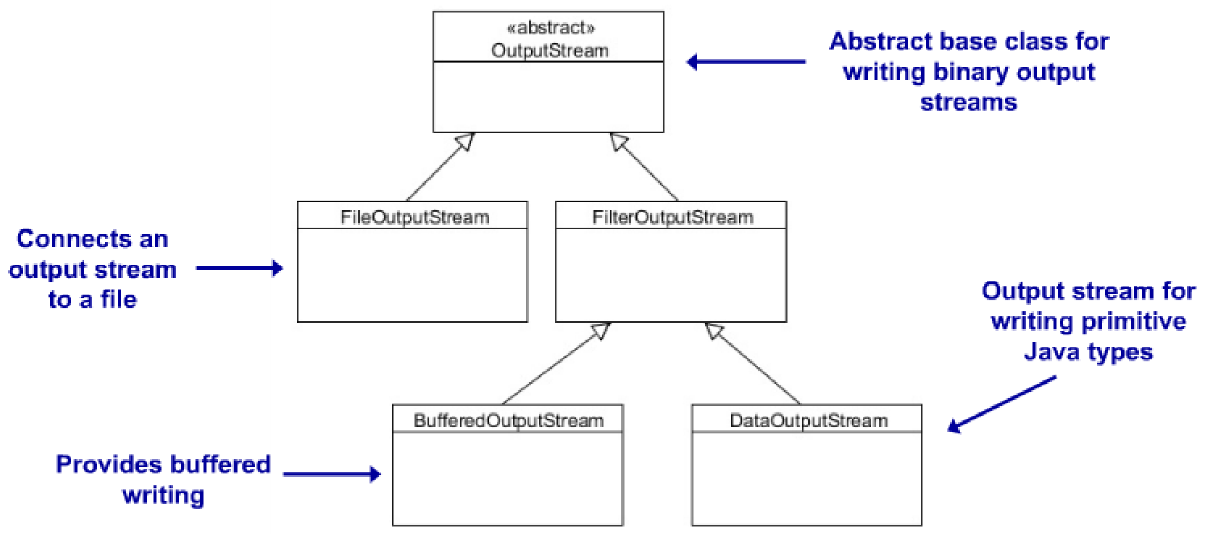
```
> run TextReadDemo4
Enter the name of the text file (including .txt):
MyPartsFormatted.txt
File will be read.
part number = P-100           part name = PartNumber100     part price = 5.25
part number = P-101           part name = PartNumber101     part price = 2.1
part number = P-102           part name = PartNumber102     part price = 8.25
part number = P-103           part name = PartNumber103     part price = 12.15
>
```

At this point, in order to gain some hands-on experience with text I/O, you may want to do the text file exercises 1 and 2 at the end of this document.

Working with Binary Files

Unlike with text files, when you read and write binary files, encoding and decoding is not needed, and as a result, binary I/O is more efficient than text I/O. Binary files are also more portable than text files because they are independent of the encoding scheme on the host computer.

Writing Binary Files


«abstract» OutputStream
Abstract base class for writing binary output streams
FileOutputStream
FilterOutputStream
Connects an output stream to a file
Output stream for writing primitive Java types
BufferedOutputStream
DataOutputStream
Provides buffered writing

There are a number of classes that can be used when writing to binary files. They are all subclasses of the abstract OutputStream class. The DataOutputStream class is used for writing primitive Java types, the BufferedOutputStream class is used to perform buffered reading, and the FileOutputStream class is used to connect an output stream to a file. The FileOutputStream class also allows data to be appended to an existing file.

The FileOutputStream Class

FileOutputStream(*pathname*: String)	A simple constructor method, where *pathname* is the pathname to the file to be written.
FileOutputStream(*file*: File)	Same as the above constructor, but a File object is specified.
flush(): void	Flushes output stream.
close(): void	Closes the FileOutputStream class and flushes output buffer.

The above table summarizes some of the common methods used with the FileOutputStream class. The constructors have a second version that takes an additional Boolean argument that, when set to 'true' allows a file to be appended to.

The DataOutputStream Class

The DataOutputStream class is used to write primitive Java types. The following table summarizes the most popular methods of the DataOutputStream class.

writeBoolean(*b*: boolean) : void	Writes a 1-byte boolean value.
writeInt(*i*: int) : void	Writes a 4-byte integer value.
….	…
writeDouble(*d*: double) : void	Writes an 8-byte double value.
writeChar(*c*: char) : void	Writes a 2-byte character value.
writeChars(*s*: String) : void	Writes every character in a string as a 2-byte value.
writeUTF(*s:* String) : void	Writes a string in UTF format.
size() : int	Returns an integer for the number of bytes written to the stream
flush() : void	Flushes the stream.
close(): void	Closes the stream and flushes output buffer.

There are methods to write all the primitive Java types…only a few are shown in the table…as well as methods to write String types in different formats.

The following program provides a simple demonstration of writing to a binary file. The program writes 14 bytes to the file: a 4-byte integer, an 8-byte double, and a 2-byte character.

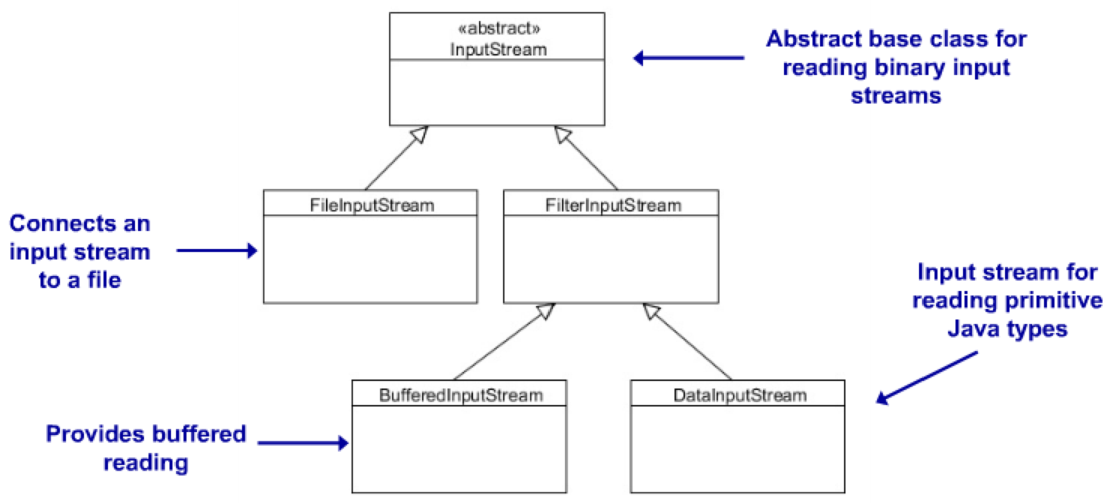
```
import java.io.*;

public class WriteBinaryDemo1
{
  public static void main( String [] args ) throws IOException
  {
    int i = 12;
    double d = 133.34;
    char c = 'A';

    File f = new File( "mybinaryfile.dat" );

    DataOutputStream out = new DataOutputStream( new FileOutputStream( f ) );

    out.writeInt( i );
    out.writeDouble( d );
    out.writeChar( c );

    System.out.println( out.size() + " bytes were written to the file" );

    out.close();
  }
}
```

Here's a second example illustrating how to write to a binary file. This program writes a first name, a last name, and a double to a binary file.

```
import java.io.*;

public class WriteBinaryDemo2
{
  public static void main( String [] args ) throws IOException
  {
    String firstName = "Tony";
    String lastName = "Baggadonuts";
    double salary = 175000.;

    File f = new File( "mybinaryfile2.dat" );

    DataOutputStream out = new DataOutputStream( new FileOutputStream( f ) );

    out.writeUTF( firstName );
    out.writeUTF( lastName );
    out.writeDouble( salary );

    System.out.println( out.size() + " bytes were written to the file" );

    out.close();
  }
}
```

In the above program, note that UTF format was used for the String types. UTF format writes a string using 2 bytes that contain the number of characters in the string, followed by the characters in the string. The actual number of bytes used to store each of the characters will be one, two, or three bytes, and will be determined according to the UTF-8 coding scheme that allows systems to operate with both ASCII and Unicode. In this example, 27 bytes will be written to the file.

Reading Binary Files


«abstract»
InputStream
Abstract base class for reading binary input streams
FileInputStream
FilterInputStream
Connects an input stream to a file
Input stream for reading primitive Java types
BufferedInputStream
DataInputStream
Provides buffered reading

There are a number of classes that can be used to read data from a binary file. They are all subclasses of the abstract InputStream class. The inheritance hierarch is illustrated above. The DataInputStream class is used to read Java primitive types, the BufferedInputStream class is used to buffer the input, and the FileInputStream class is used to connect an input stream to a file.

The FIleInputStream Class

FileInputStream(*pathname*: String)	A simple constructor method, where *pathname* is the pathname to the file to be read.
FileInputStream(*file*: File)	Same as the above constructor, but a File object is specified.
available() : int	Returns the number of bytes that can be read from the input stream.
skip(*n*: long) : long	Skips over and discards n bytes from the input stream, and returns the actual number of bytes skipped.
close(): void	Closes the FileOutputStream class and flushes output buffer.

The above table summarizes some of the common methods of the FileInputStream class. The last three methods are actually inherited from the InputStream class.

The DataInputStream Class

readBoolean() : boolean	Reads a 1-byte boolean value.
readInt() : int	Reads a 4-byte integer value.
….	…
readDouble() : double	Reads an 8-byte double value.
readChar() : char	Reads a 2-byte character value.
readUTF() : String	Reads a string in UTF format.
readLine() : String	Reads a line of characters from the input stream.

The above table illustrates some of the most common methods of the DataInputStream class. All the methods that read primitive types are not shown for the sake of brevity.

The program below reads three primitive types from the binary file that was created in the WriteBinaryDemo1 program earlier in this document, and simply displays the read values.

```
import java.io.*;

public class ReadBinaryDemo1
{
  public static void main( String [] args ) throws IOException
  {
    File f = new File( "mybinaryfile.dat" );

    DataInputStream in = new DataInputStream( new FileInputStream( f ) );

    int i = in.readInt();
    double d = in.readDouble();
    char c = in.readChar();

    System.out.println( "Values read were: " + i + "\t" + d + "\t" + c );

    in.close();
  }
}
```

The program below illustrates how to read a binary file in the situation where we know the types of data to be read, but we don't know how many data elements are in the file.

```
import java.io.*;

public class BinaryEOFDemo
{
  public static void main( String [] args ) throws IOException
  {
    File f = new File( "mybinaryfile.dat" );

    DataInputStream in = new DataInputStream( new FileInputStream( f ) );

    while( in.available() > 0 )
    {
      int i = in.readInt();
      double d = in.readDouble();
      char c = in.readChar();

      System.out.println( "Values read were: " + i + "\t" + d + "\t" + c );
    }

    in.close();
  }
}
```

In the above program, the *available()* method is used to test for an end of file condition. The loop will continue to read data until there are no more bytes remaining in the stream. Note that with each iteration of the loop a different value will be returned by a call to this method.

Here's one more example, of reading binary files. This program reads the binary file created by the WriteBinaryDemo2 program that appeared earlier in this document.

```
import java.io.*;

public class BinaryEOFDemo2
{
  public static void main( String [] args ) throws IOException
  {
    File f = new File( "mybinaryfile2.dat" );

    DataInputStream in = new DataInputStream( new FileInputStream( f ) );

    while( in.available() > 0 )
    {
      String firstName = in.readUTF();
      String lastName = in.readUTF();
      double salary = in.readDouble();

      System.out.println( "Values read were: " + firstName + "\t" + lastName
+ "\t" + salary );
    }

    in.close();
  }
}
```

Buffering Input and Output Streams

When we deal with binary files, we can buffer input and output streams for greater efficiency, just like we did when working with text files, by wrapping the BufferedInputStream and BufferedOutputStream classes. Doing this will be left as an exercise for the reader of this document.

At this point, in order to gain some hands-on experience with binary I/O, you may want to do exercises 3 and 4 at the end of this document.

Random-Access Files

All of the streams we have used in this document so far are often referred to as *read-only* or *write-only* streams. These streams are also called *sequential streams*. A file that is opened using a sequential stream is called a *sequential-access file*. For example, if we wanted to read the tenth data element stored in either a text or binary file, we'd need to design a program that first read the preceding nine data elements. And, if we wanted to update the existing data in a file, we'd have to use a program that reads the entire file and saves all the data elements, then have it change the appropriate data elements, and then write a completely new file. Sequential access files cannot be directly updated, unless we simply append data to an existing file. In practice, however, it is often necessary to modify files, and here is where the random-access file can help out.

Random-access files work a little differently than sequential-access files. They allow us to directly access any location in a file directly, and then read from or write to that location. This enables us to access or modify a file without affecting the rest of the file.

The RandomAccessFile Class

The RandomAccessFile class is used to create and manipulate random access files. It contains a number of methods, some of which are illustrated in the table below.

RandomAccessFile(*pathname*: String , *mode*: String)	A simple constructor method, where *pathname* is the pathname to the file to be read, and *mode* is the access mode.
RandomAccessFile(*file*: File , *mode*: String)	Same as the above constructor, but a File object is specified.
length() : long	Returns the number of bytes in this file.
getFilePointer() : long	Returns the offset, in bytes, from the beginning of the stream to where the next read or write occurs.
seek(*pos*: long) : void	Sets the offset to *pos* bytes from the beginning of the stream to where the next read or write occurs. If the offset is longer than the file, it will be set to the end of the file.
setLength(*len*: long) : void	Sets the new length of the file to *len* bytes. If *len* is less than the current length, the file is truncated.
skipBytes(*n*: int) : int	Skips over *n* bytes of input. Returns the actual number of bytes skipped.
close(): void	Closes the stream and releases any resources.

Take a look at the constructors in the above table. The second constructor argument is the file access mode. The two most popular file access modes are *r* and *rw*. The access mode *r* means that the file access mode is read-only. The *rw* mode means that the file allows both read and write operations.

To read primitive types, the *readInt()*, *readChar()*, … , *readDouble*() methods can be used. To read strings, the *readUTF()* method can be used. To write primitive types, the *writeInt()*, *writeChar()*, …, *writeDouble()* methods can be used. And, to write strings, the *writeUTF()* method can be used.

Several examples follow below that illustrate basic usage of random access files.

```
public class RandomAccessWriteDemo1
{
  public static void main( String [] args ) throws IOException
  {
    int [] productCodes = { 100, 102, 103 };
    double [] productPrices = { 1.10, 2.95, 1.00 };

    RandomAccessFile f = new RandomAccessFile( "productdata.dat", "rw" );

    for( int i = 0; i < 3; i++ )
    {
      f.writeInt( productCodes[i] );
      f.writeDouble( productPrices[i] );
    }

    System.out.println( f.length() + " bytes written to file" );

    f.close();
  }
}
```

The above program writes a set of data that describe codes and prices with a set of three products. A random access file is created, and the *writeInt()* and *writeDouble()* methods are used to write this data to the file. A total of 36 bytes will be written to the file. The program below calculates how many product records are in the file, reads the random access file, and displays the product data.

```
public class RandomAccessReadDemo1
{
  public static void main( String [] args ) throws IOException
  {
    int productCode;
    double productPrice;

    RandomAccessFile f = new RandomAccessFile( "productdata.dat", "rw" );

    int numRecords = (int) f.length() / 12;  // Number of records in file

    for( int i = 1; i <= numRecords; i++ )
    {
      productCode = f.readInt();
      productPrice = f.readDouble();

      System.out.println( productCode + "\t" + productPrice );
    }

    f.close();
  }
}
```

The next program will update the product data file by changing the product price for the second product.

```
// This program adds a product to the product file and changes the
// product information for the second product.

import java.io.*;

public class RandomAccessWriteDemo2
{
  public static void main( String [] args ) throws IOException
  {
    int productCode = 199;
    double productPrice = 3.50;

    // Open the file in read/write mode
    RandomAccessFile f = new RandomAccessFile( "productdata.dat", "rw" );

    long lastByte = f.length();       // Get the length of the file
    f.seek( lastByte );               // Set offset to end of file

    // Add new product info
    f.writeInt( productCode );
    f.writeDouble( productPrice );

    // Display info for second product
    f.seek( 28 );                    // Set pointer to second product price
    f.writeDouble( 2.99 );           // Change price of second product

    f.close();
  }
}
```

You can verify that the price was successfully updated by running the above program and then running the RandomAccessReadDemo1 program to display the product information.

At this point, in order to gain some hands-on experience with random access file I/O, you may want to do exercise 5 at the end of this document.

Recommended Programming Exercises

Program 1

Write a program called *WriteTaskFileExercise.java* that will create a text file for storing information about tasks: a task number, a task description, and a task due date. The task due date is represented by a Date object. You have already used a simple class file for Date objects earlier in the semester, so please reuse it. Each set of task data shall be written on a single line, and the task number shall be right-justified in a field with of 5, the task description shall be left-justified in a field width of 30, and the due date shall be written as a string right-justified in a field width of 15. Create a file called *tasks.txt* that contains the following task information:

Task #	Task Description	Due Date
1	Do Java homework	3/10/2017
2	Get concert tickets	2/20/2017
3	Study for math test	3/30/2017

Run your program to create the file, and then examine the file using a text editor or operating system command to verify that its contents were written correctly. When you have your program working, you may compare your solution to the sample solution provided as a download on the course website.

Program 2

Write a program called *TaskFileReaderExercise.java* that reads the task information in the *tasks.txt* file that was created in the last exercise and displays the information on the standard output as follows:

Task #: *num*
Task: *task description*
Date: *mm/dd/yyyy*

Write the program assuming you do not know how many task records are in the file. When you have your program running, you may compare your solution to the sample solution provided as a download on the course website.

Program 3

Write a program called *WriteBinaryFileExercise.java* that writes a binary file containing information for a hardware store's products. Write the following data for four products to a file called *products.dat*.

Product Number and Quantity should be integer types, Name should be a String type, and Cost should be a double.

Product Number	Name	Quantity	Cost
110	Hammer	20	$ 12.99
520	Lawn Mower	8	79.52
178	Monkey Wrench	52	6.95
172	Screwdriver	150	5.99

When you have your program running, you may compare your solution to the sample solution provided as a download on the course website.

Program 4

Write a program called *ReadBinaryFileExercise.java* that the binary file *products.dat* that was created in the last programming exercise and displays the product information on the standard output. Do not make any assumptions about how many data records are stored in the file.

When you have your program running, you may compare your solution to the sample solution provided as a download on the course website.

<u>Program 5</u>

Write a program called *UpdateRandomAccessFileExercise.java* that uses random access file functionality to update the file called *products.dat* that was created in an earlier exercise.

The program should update the quantity of Monkey Wrenches to 48, and add a new product to the file with the following data:

Product Number	Name	Quantity	Cost
100	Roofing Tacks	1500	$ 7.95

When you have your program running, you may compare your solution to the sample solution provided as a download on the course website.