Unit 4
Topic: Exploring Java I/O
Unit 4/Lecture 1

File:

Files are a primary source and destination for data within many programs. Although there are severe restrictions on their use within applets for security reasons, files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list ( )** method.

The following constructors can be used to create **File** objects:

File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)

File defines many methods that obtain the standard properties of a **File** object. For example, **getName( )** returns the name of the file, **getParent( )** returns the name of the parent directory,
and **exists( )** returns **true** if the file exists, **false** if it does not. The **File** class, however, is not symmetrical. By this, we mean that there are a few methods that allow you to *examine* the properties of a simple file object, but no corresponding function exists to change those attributes.

The following example demonstrates several of the **File** methods:

```
// Demonstrate File.
import java.io.File;
class FileDemo {
static void p(String s) {
System.out.println(s);
}
public static void main(String args[]) {
File f1 = new File("/java/COPYRIGHT");
p("File Name: " + f1.getName());
p("Path: " + f1.getPath());
p("Abs Path: " + f1.getAbsolutePath());
p("Parent: " + f1.getParent());
p(f1.exists() ? "exists" : "does not exist");
p(f1.canWrite() ? "is writeable" : "is not writeable");
p(f1.canRead() ? "is readable" : "is not readable");
p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
p(f1.isFile() ? "is normal file" : "might be a named pipe");
p(f1.isAbsolute() ? "is absolute" : "is not absolute");
p("File last modified: " + f1.lastModified());
p("File size: " + f1.length() + " Bytes");
}
```

```
}
```

When you run this program, you will see something similar to the following:

```
File Name: COPYRIGHT
Path: /java/COPYRIGHT
Abs Path: /java/COPYRIGHT
Parent: /java
exists
is writeable
is readable
is not a directory
is normal file
is absolute
File last modified: 812465204000
File size: 695 Bytes
```

Directories

A directory is a **File** that contains a list of other files and directories. When you create a **File** object and it is a directory, the **isDirectory( )** method will return **true**. In this case, you can call **list( )** on that object to extract the list of other files and directories inside. It has two forms. The first is shown here:

```
String[ ] list( )
```

The list of files is returned in an array of **String** objects. The program shown here illustrates how to use **list( )** to examine the contents of a directory:

```
// Using directories.
import java.io.File;
class DirList {
public static void main(String args[]) {
String dirname = "/java";
File f1 = new File(dirname);
if (f1.isDirectory()) {
System.out.println("Directory of " + dirname);
String s[] = f1.list();
for (int i=0; i < s.length; i++) {
File f = new File(dirname + "/" + s[i]);
if (f.isDirectory()) {
System.out.println(s[i] + " is a directory");
} else {
System.out.println(s[i] + " is a file");
}
}
} else {
System.out.println(dirname + " is not a directory");
}
}
```

```
}
```

Here is sample output from the program. (Of course, the output you see will be different, based on what is in the directory.)

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
src is a directory
```

Unit 4

Topic: Exploring Java I/O(Byte Stream Classes)

Unit 4/Lecture 2

The Byte Stream[RGPV Dec 2014(2)]

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. Since the byte stream classes are topped by **InputStream** and **OutputStream**.

InputStream

InputStream is an abstract class that defines Java's model of streaming byte input. It implements the **Closeable** interface. Most of the methods in this class will throw an **IOException** on error conditions. (The exceptions are **mark ( )** and **markSupported ( )**. Table 4.1 shows the methods in **InputStream**.

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns **true** if **mark( )**/**reset( )** are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

Table 4.1: The Methods defined by InputStream

OutputStream

OutputStream is an abstract class that defines streaming byte output. It implements the **Closeable** and **Flushable** interfaces. Most of the methods in this class return **void** and throw an **IOException** in the case of errors. (The exceptions are **mark ( )** and **markSupported ( )**. Table 4.2 shows the methods in **OutputStream**.

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with expressions without having to cast them back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

Table 4.2: The Methods defined by OutputStream

FileInputStream [RGPV/Dec-2013(8), Dec-2014(2)]

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Its two most common constructors are shown here:

FileInputStream(String *filepath*)
FileInputStream(File *fileObj*)

Either can throw a **FileNotFoundException**. Here, *filepath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. The following example creates two **FileInputStream**s that use the same disk file and each of the two constructors:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Although the first constructor is probably more commonly used, the second allows us to closely examine the file using the **File** methods, before we attach it to an input stream. When a **FileInputStream** is created, it is also opened for reading. **FileInputStream** overrides six of the methods in the abstract class **InputStream**. The **mark( )** and **reset( )** methods are not overridden, and any attempt to use **reset( )** on a **FileInputStream** will generate an **IOException**.

The next example shows how to read a single byte, an array of bytes, and a subrange array of bytes. It also illustrates how to use **available( )** to determine the number of bytes remaining, and how to use the **skip( )** method to skip over unwanted bytes. The program reads its own source file, which must be in the current directory.

```
// Demonstrate FileInputStream.
import java.io.*;
class FileInputStreamDemo {
public static void main(String args[]) throws IOException {
int size;
InputStream f =
new FileInputStream("FileInputStreamDemo.java");
System.out.println("Total Available Bytes: " +
(size = f.available()));
int n = size/40;
System.out.println("First " + n +
" bytes of the file one read() at a time");
for (int i=0; i < n; i++) {
System.out.print((char) f.read());
}
System.out.println("\nStill Available: " + f.available());
```

```
System.out.println("Reading the next " + n +
" with one read(b[])");
byte b[] = new byte[n];
if (f.read(b) != n) {
System.err.println("couldn't read " + n + " bytes.");
}
System.out.println(new String(b, 0, n));
System.out.println("\nStill Available: " + (size = f.available()));
System.out.println("Skipping half of remaining bytes with skip()");
f.skip(size/2);
System.out.println("Still Available: " + f.available());
System.out.println("Reading " + n/2 + " into the end of array");
if (f.read(b, n/2, n/2) != n/2) {
System.err.println("couldn't read " + n/2 + " bytes.");
}
System.out.println(new String(b, 0, b.length));
System.out.println("\nStill Available: " + f.available());
f.close();
}
}
```

Here is the output produced by this program:

```
Total Available Bytes: 1433
First 35 bytes of the file one read() at a time
// Demonstrate FileInputStream.
im
Still Available: 1398
Reading the next 35 with one read(b[])
port java.io.*;
class FileInputS
Still Available: 1363
Skipping half of remaining bytes with skip()
Still Available: 682
Reading 17 into the end of array
port java.io.*;
read(b) != n) {
S
Still Available: 665
```

FileOutputStream

FileOutputStream creates an **OutputStream** that you can use to write bytes to a file. Its most commonly used constructors are shown here:

FileOutputStream(String *filePath*)
FileOutputStream(File *fileObj*)
FileOutputStream(String *filePath*, boolean *append*)
FileOutputStream(File *fileObj*, boolean *append*)

The following example creates a sample buffer of bytes by first making a **String** and then using the **getBytes( )** method to extract the byte array equivalent. It then creates three files.

The first, **file1.txt**, will contain every other byte from the sample. The second, **file2.txt**, will contain the entire set of bytes. The third and last, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileOutputStream.
import java.io.*;
class FileOutputStreamDemo {
public static void main(String args[]) throws IOException {
String source = "Now is the time for all good men\n"
+ " to come to the aid of their country\n"
+ " and pay their due taxes.";
byte buf[] = source.getBytes();
OutputStream f0 = new FileOutputStream("file1.txt");
for (int i=0; i < buf.length; i += 2) {
f0.write(buf[i]);
}
f0.close();
OutputStream f1 = new FileOutputStream("file2.txt");
f1.write(buf);
f1.close();
OutputStream f2 = new FileOutputStream("file3.txt");
f2.write(buf,buf.length-buf.length/4,buf.length/4);
f2.close();
}
}
```

Here are the contents of each file after running this program. First, **file1.txt**:

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Next, **file2.txt**:

```
Now is the time for all good men
to come to the aid of their country
and pay their due taxes.
```

Finally, **file3.txt**:

```
nd pay their due taxes.
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What is stream? Explain types of stream and classes of the stream	Dec 2014	2
Q.2	Explain the Constructors of FileInputStream class. Write a program to read a text file stored in the same directory as the program, reverse its contents and display them on the screen.	Dec 2013, Dec 2014	08,7
Q-3	Explain Java Input and Output Stream.	Dec 2009	10

Unit 4
Topic: Exploring Java I/O
Unit 4/Lecture 3

PrintStream

The **PrintStream** class provides all of the output capabilities we have been using from the **System** file handle, **System.out**, since the beginning of the book. This makes **PrintStream** one of Java's most often used classes. It implements the **Appendable**, **Closeable**, and **Flushable** interfaces. **PrintStream** defines several constructors. The ones shown next have been specified from the start:

PrintStream(OutputStream *outputStream*)
PrintStream(OutputStream *outputStream*, boolean *flushOnNewline*)
PrintStream(OutputStream *outputStream*, boolean *flushOnNewline*,
String *charSet*)

RandomAccessFile

RandomAccessFile encapsulates a random-access file. It is not derived from **InputStream** or **OutputStream**. Instead, it implements the interfaces **DataInput** and **DataOutput**, which define the basic I/O methods. It also implements the **Closeable** interface. **RandomAccessFile** is special because it supports positioning requests—that is, you can position the *file pointer* within the file. It has these two constructors:

RandomAccessFile(File *fileObj*, String *access*) throws FileNotFoundException
RandomAccessFile(String *filename*, String *access*) throws FileNotFoundException

In the first form, *fileObj* specifies the name of the file to open as a **File** object. In the second form, the name of the file is passed in *filename.* In both cases, *access* determines what type of file access is permitted. If it is "r", then the file can be read, but not written. If it is "rw", then the file is opened in read-write mode. If it is "rws", the file is opened for read-write operations and every change to the file's data or metadata will be immediately written to the physical device. If it is "rwd", the file is opened for read-write operations and every change to the file's data will be immediately written to the physical device.

The method **seek( )**, shown here, is used to set the current position of the file pointer within the file:
void seek(long *newPos*) throws IOException

Here, *newPos* specifies the new position, in bytes, of the file pointer from the beginning of the file. After a call to **seek( )**, the next read or write operation will occur at the new file position. **RandomAccessFile** implements the standard input and output methods, which you can use to read and write to random access files. It also includes some additional methods. One is **setLength( )**. It has this signature:

void setLength(long *len*) throws IOException

This method sets the length of the invoking file to that specified by *len.* This method can be used to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.

DataOutputStream and DataInputStream [RGPV/Dec-2010(5)]

DataOutputStream and **DataInputStream** enable you to write or read primitive data to or from a stream. They implement the **DataOutput** and **DataInput** interfaces, respectively. These interfaces define methods that convert primitive values to or from a sequence of bytes. These streams make it easy to store binary data, such as integers or floating-point values, in a file. Each is examined here.

DataOutputStream extends **FilterOutputStream**, which extends **OutputStream**. **DataOutputStream** defines the following constructor:

DataOutputStream(OutputStream *outputStream*)
Here, *outputStream* specifies the output stream to which data will be written.

DataOutputStream supports all of the methods defined by it superclasses. However, it is the methods defined by the **DataOutput** interface, which it implements, that make it interesting. **DataOutput** defines methods that convert values of a primitive type into a byte sequence and then writes it to the underlying stream. Here is a sampling of these methods:

final void writeDouble(double *value*) throws IOException
final void writeBoolean(boolean *value*) throws IOException
final void writeInt(int *value*) throws IOException

Here, *value* is the value written to the stream.

DataInputStream is the complement of **DataOuputStream**. It extends **FilterInputStream**, which extends **InputStream**, and it implements the **DataInput** interface. Here is its only constructor:
DataInputStream(InputStream *inputStream*)
Here, *inputStream* specifies the input stream from which data will be read.
Like **DataOutputStream**, **DataInputStream** supports all of the methods of its superclasses, but it is the methods defined by the **DataInput** interface that make it unique. These methods read a sequence of bytes and convert them into values of a primitive type. Here is a sampling
of these methods:
double readDouble( ) throws IOException
boolean readBoolean( ) throws IOException
int readInt( ) throws IOException

The following program demonstrates the use of **DataOutputStream** and **DataInputStream**:

```
import java.io.*;
class DataIODemo {
public static void main(String args[])
```

```
throws IOException {
FileOutputStream fout = new FileOutputStream("Test.dat");
DataOutputStream out = new DataOutputStream(fout);
out.writeDouble(98.6);
out.writeInt(1000);
out.writeBoolean(true);
out.close();
FileInputStream fin = new FileInputStream("Test.dat");
DataInputStream in = new DataInputStream(fin);
double d = in.readDouble();
int i = in.readInt();
boolean b = in.readBoolean();
System.out.println("Here are the values: " +
d + " " + i + " " + b);
in.close();
}
}
```

The output is shown here:

```
Here are the values: 98.6 1000 true
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What are the difference between DataOutputStream and PrintStream.	Dec 2010	05

Unit 4

Topic: Exploring Java I/O (Character Streams)

Unit 4/Lecture 4

Character Streams [RGPV/Dec 2012(10)]

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes
of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters. In this section, several of the character I/O classes are discussed

Reader

Reader is an abstract class that defines Java's model of streaming character input. It implements the **Closeable** and **Readable** interfaces. All of the methods in this class (except for **markSupported( )**) will throw an **IOException** on error conditions.

Writer

Writer is an abstract class that defines streaming character output. It implements the **Closeable**, **Flushable**, and **Appendable** interfaces. All of the methods in this class throw an **IOException** in the case of errors. Table 19-4 shows a synopsis of the methods in **Writer**.

FileReader

The **FileReader** class creates a **Reader** that you can use to read the contents of a file. Its two most commonly used constructors are shown here:

FileReader(String *filePath*)
FileReader(File *fileObj*)
Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

The following example shows how to read lines from a file and print these to the standard output stream. It reads its own source file, which must be in the current directory.

```
// Demonstrate FileReader.
import java.io.*;
class FileReaderDemo {
public static void main(String args[]) throws IOException {
FileReader fr = new FileReader("FileReaderDemo.java");
BufferedReader br = new BufferedReader(fr);
String s;
while((s = br.readLine()) != null) {
System.out.println(s);
}
fr.close();
```

```
}
}
```

FileWriter

FileWriter creates a **Writer** that you can use to write to a file. Its most commonly used constructors are shown here:

FileWriter(String *filePath*)
FileWriter(String *filePath*, boolean *append*)
FileWriter(File *fileObj*)
FileWriter(File *fileObj*, boolean *append*)

They can throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File**
object that describes the file. If *append* is **true**, then output is appended to the end of the file.

Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.

The following example is a character stream version of an example shown earlier when **FileOutputStream** was discussed. This version creates a sample buffer of characters by first making a **String** and then using the **getChars( )** method to extract the character array equivalent. It then creates three files. The first, **file1.txt**, will contain every other character from the sample. The second, **file2.txt**, will contain the entire set of characters. Finally, the third, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileWriter.
import java.io.*;
class FileWriterDemo {
public static void main(String args[]) throws IOException {
String source = "Now is the time for all good men\n"
+ " to come to the aid of their country\n"
+ " and pay their due taxes.";
char buffer[] = new char[source.length()];
source.getChars(0, source.length(), buffer, 0);
FileWriter f0 = new FileWriter("file1.txt");
for (int i=0; i < buffer.length; i += 2) {
f0.write(buffer[i]);
}
f0.close();
FileWriter f1 = new FileWriter("file2.txt");
f1.write(buffer);
f1.close();
FileWriter f2 = new FileWriter("file3.txt");
f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
f2.close();
}
```

BufferedReader

BufferedReader improves performance by buffering input. It has two constructors:
BufferedReader(Reader *inputStream*)
BufferedReader(Reader *inputStream*, int *bufSize*)
The first form creates a buffered character stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize.*
As is the case with the byte-oriented stream, buffering an input character stream also provides the foundation required to support moving backward in the stream within the available buffer. To support this, **BufferedReader** implements the **mark( )** and **reset( )** methods, and **BufferedReader.markSupported( )** returns **true**. The following example reworks the **BufferedInputStream** example, shown earlier, so that it uses a **BufferedReader** character stream rather than a buffered byte stream. As before, it uses **mark( )** and **reset( )** methods to parse a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands, to show the case where the **reset( )** happens and where it does not. Output is the same as that shown earlier.

```
// Use buffered input.
import java.io.*;
class BufferedReaderDemo {
public static void main(String args[]) throws IOException {
String s = "This is a &copy; copyright symbol " +
"but this is &copy not.\n";
char buf[] = new char[s.length()];
s.getChars(0, s.length(), buf, 0);
CharArrayReader in = new CharArrayReader(buf);
BufferedReader f = new BufferedReader(in);
int c;
boolean marked = false;
while ((c = f.read()) != -1) {
switch(c) {
case '&':
if (!marked) {
f.mark(32);
marked = true;
} else {
marked = false;
}
break;
case ';':
if (marked) {
marked = false;
System.out.print("(c)");
} else
System.out.print((char) c);
break;
case ' ':
```

```
if (marked) {
marked = false;
f.reset();
System.out.print("&");
} else
System.out.print((char) c);
break;
default:
if (!marked)
System.out.print((char) c);
break;
}
}
}
}
```

BufferedWriter

A **BufferedWriter** is a **Writer** that buffers ouput. Using a **BufferedWriter** can increase performance by reducing the number of times data is actually physically written to the output stream.

A**BufferedWriter** has these two constructors:
BufferedWriter(Writer *outputStream*)
BufferedWriter(Writer *outputStream*, int *bufSize*)

The first form creates a buffered stream using a buffer with a default size. In the second, the size of the buffer is passed in *bufSize.*

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What are the abstract classes for handling byte stream and character stream? Explain.	Dec 2012	10

Unit 4
Topic: Exploring Java I/O
Unit 4/Lecture 5

PrintWriter

PrintWriter is essentially a character-oriented version of **PrintStream**. It implements the **Appendable**, **Closeable**, and **Flushable** interfaces. **PrintWriter** has several constructors. The following have been supplied by **PrintWriter** from the start:

PrintWriter(OutputStream *outputStream*)
PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)
PrintWriter(Writer *outputStream*)
PrintWriter(Writer *outputStream*, boolean *flushOnNewline*)

Here, *outputStream* specifies an open **OutputStream** that will receive output. The *flushOnNewline* parameter controls whether the output buffer is automatically flushed every time **println( )**, **printf( )**, or **format( )** is called. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic. Constructors that do not specify the *flushOnNewline* parameter do not automatically flush.

These allow a **PrintWriter** to be created from a **File** object or by specifying the name of a file.
In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the **PrintWriter** object directs all output to the specified file. You can specify a character encoding by passing its name in *charSet.*
PrintWriter supports the **print( )** and **println( )** methods for all types, including **Object**. If an argument is not a primitive type, the **PrintWriter** methods will call the object's **toString( )** method and then output the result.

PrintWriter also supports the **printf ( )** method. It works the same way it does in the **PrintStream** class described earlier: it allows you to specify the precise format of the data. Here is how **printf( )** is declared in **PrintWriter**:

PrintWriter printf(String *fmtString*, Object ... *args*)
PrintWriter printf(Locale *loc*, String *fmtString*, Object ... *args*)
The first version writes *args* to standard output in the format specified by *fmtString,* using the
default locale. The second lets you specify a locale. Both return the invoking **PrintWriter**.
The **format ( )** method is also supported. It has these general forms:

PrintWriter format(String *fmtString*, Object ... *args*)
PrintWriter format(Locale *loc*, String *fmtString*, Object ... *args*)
It works exactly like **printf( )**.

Serialization [RGPV/Dec-2010(10),Dec-2014(2)]

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization. Serialization is also needed to implement Remote Method Invocation (RMI). RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.

An overview of the interfaces and classes that support serialization follows.

Serializable

Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities. The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable. Variables that are declared as **transient** are not saved by the serialization facilities. Also, **static** variables are not saved.

Externalizable

The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The **Externalizable** interface is designed for these situations.

The **Externalizable** interface defines these two methods:

void readExternal(ObjectInput *inStream*)
throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput *outStream*)
throws IOException

In these methods, *inStream* is the byte stream from which the object is to be read, and *outStream* is the byte stream to which the object is to be written.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What is Serialization? Write a program in Java for Serialization in an Object.	Dec 2012 Dec 2014	10,2

Unit 4
Topic: Exploring Java I/O
Unit 4/Lecture 6

ObjectOutput

The **ObjectOutput** interface extends the **DataOutput** interface and supports object serialization. It defines the methods shown in Table 19-6. Note especially the **writeObject( )** method. This is called to serialize an object.

All of these methods will throw an **IOException** on error conditions.

ObjectOutputStream

The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface. It is responsible for writing objects to a stream. A constructor of this class is ObjectOutputStream(OutputStream *outStream*) throws IOException The argument *outStream* is the output stream to which serialized objects will be written. Several commonly used methods in this class are shown in The table given below. They will throw an **IOException** on error conditions.

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*.
void writeObject(Object *obj*)	Writes object *obj* to the invoking stream.

TABLE 19-6 The Methods Defined by ObjectOutput

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a **boolean** to the invoking stream.
void writeByte(int *b*)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to the invoking stream.
void writeChar(int *c*)	Writes a **char** to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invoking stream.
void writeDouble(double *d*)	Writes a **double** to the invoking stream.
void writeFloat(float *f*)	Writes a **float** to the invoking stream.
void writeInt(int *i*)	Writes an **int** to the invoking stream.
void writeLong(long *l*)	Writes a **long** to the invoking stream.
final void writeObject(Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int *i*)	Writes a **short** to the invoking stream.

TABLE 19-7 Commonly Used Methods Defined by ObjectOutputStream

ObjectInput

The **ObjectInput** interface extends the **DataInput** interface and defines the methods shown in Table 19-8. It supports object serialization. Note especially the **readObject( )** method. This is called to deserialize an object. All of these methods will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**.

ObjectInputStream

The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface. **ObjectInputStream** is responsible for reading objects from a stream. Aconstructor of this class is

ObjectInputStream(InputStream *inStream*)
throws IOException

The argument *inStream* is the input stream from which serialized objects should be read. Several commonly used methods in this class are shown in Table given below. They will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**. There is also an inner class to **ObjectInputStream** called **GetField**. It facilitates the reading of persistent fields, and its use is beyond the scope of this book.

A Serialization Example

The following program illustrates how to use object serialization and deserialization. It begins

by instantiating an object of class **MyClass**. This object has three instance variables that are of

types **String**, **int**, and **double**. This is the information we want to save and restore.

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer*, returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
Object readObject( )	Reads an object from the invoking stream.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored.

TABLE 19-8 The Methods Defined by **ObjectInput**

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
boolean readBoolean( )	Reads and returns a **boolean** from the invoking stream.
byte readByte( )	Reads and returns a **byte** from the invoking stream.
char readChar( )	Reads and returns a **char** from the invoking stream.
double readDouble( )	Reads and returns a **double** from the invoking stream.
float readFloat( )	Reads and returns a **float** from the invoking stream.
void readFully(byte *buffer*[ ])	Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read.
void readFully(byte *buffer*[ ], int *offset*, int *numBytes*)	Reads *numBytes* bytes into *buffer* starting at *buffer*[*offset*]. Returns only when *numBytes* have been read.
int readInt( )	Reads and returns an **int** from the invoking stream.
long readLong( )	Reads and returns a **long** from the invoking stream.
final Object readObject( )	Reads and returns an object from the invoking stream.
short readShort( )	Reads and returns a **short** from the invoking stream.
int readUnsignedByte( )	Reads and returns an unsigned **byte** from the invoking stream.
int readUnsignedShort( )	Reads and returns an unsigned **short** from the invoking stream.

TABLE 19-9 Commonly Used Methods Defined by **ObjectInputStream**

```
import java.io.*; [RGPV/Dec-2010(10)]
public class SerializationDemo {
public static void main(String args[]) {
// Object serialization
try {
MyClass object1 = new MyClass("Hello", -7, 2.7e10);
System.out.println("object1: " + object1);
FileOutputStream fos = new FileOutputStream("serial");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(object1);
oos.flush();
oos.close();
}
catch(IOException e) {
System.out.println("Exception during serialization: " + e);
System.exit(0);
}
// Object deserialization
try {
MyClass object2;
FileInputStream fis = new FileInputStream("serial");
ObjectInputStream ois = new ObjectInputStream(fis);
object2 = (MyClass)ois.readObject();
ois.close();
System.out.println("object2: " + object2);
}
catch(Exception e) {
System.out.println("Exception during deserialization: " + e);
System.exit(0);
}
}
}
class MyClass implements Serializable {
String s;
int i;
double d;
public MyClass(String s, int i, double d) {
this.s = s;
this.i = i;
this.d = d;
}
public String toString() {
return "s=" + s + "; i=" + i + "; d=" + d;
}
}
```

This program demonstrates that the instance variables of **object1** and **object2** are identical.

The output is shown here:
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What is Serialization? Write a program in Java for Serialization in an Object.	Dec 2012	10

Unit 4

Topic: Java Database Connectivity (JDBC)

Unit 4/Lecture 7

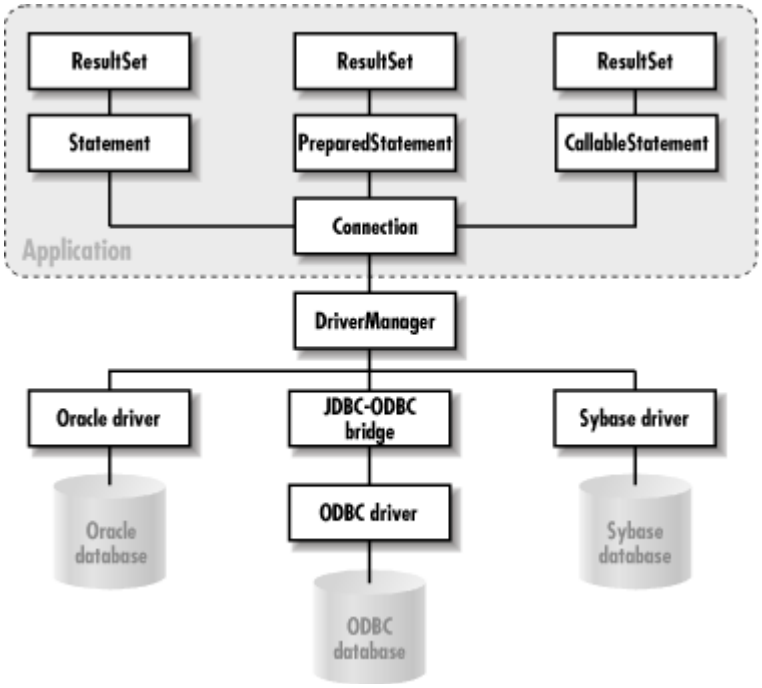
What is JDBC [RGPV/June 2011(8),June 2014(7)]

JDBC (Java Database Connectiveity) is an API (Application Programming Interface), That is, a collection of classes and interfaces.

• JDBC is used for accessing (mainly) databases from Java applications.
• Information is transferred from *relations* to *objects* and vice-versa.
  – *databases* optimized for *searching/indexing*
  – *objects* optimized for *engineering/flexibility*

JDBC Architecture

The architecture of JDBC is given below, the components of this architecture is described below.



DriverManager

- Loads JDBC driver into JVM
- Used to obtain connections to a DataSource

Connection

- Represents a connection with a DataSource
- Used to create Statement, PreparedStatement and
- CallableStatement objects.

Statement

- Represents a static SQL statement.
- Can be used to retrieve ResultSet objects.

PreparedStatement

- Higher performance alternative to Statement object,
- represents a precompiled SQL statement.

CallableStatement

- Represents a stored procedure. Can be used to execute stored procedures in a RDBMS which
  supports them.

ResultSet

- Represents a database result set generated by using a SELECT SQL statement.

SQLException

- An exception class which encapsulates database base access errors.

DataSource

- Abstracts a data source. This object can be used in place of DriverManager to efficiently obtain data source connections

There are seven steps for the Java Database connectivity.

- Load the driver
- Define the connection URL
- Establish the connection
- Create a Statement object
- Execute a query using the Statement
- Close the connection

Step-One Loading the Driver

The driver we are using will need to be registered with the JDBC DriverManager Typically done via loading the class directly.

Class.forName(“com.oracle.jdbc.OracleDriver”);
This method may throw a ClassNotFoundException, so we must wrap it in a try/catch block

```
try {
Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
System.err.println("Failed to load the JDBC driver");
}
```

Step Two Define the Connection URL

JDBC drivers use a JDBC URL to identify and connect to a given database Typically specify

- Driver name
- Machine to connect to
- Database name
- Username (typically optional)
- Password (typically optional)
- **General format like so:**
  jdbc:driver:databasename

Step-three Connecting to the Database

To connect to the database we need to get a connection off of the DriverManager Connection conn = DriverManager.getConnection("url", "user", "password");

We should also take some care to close the connection when we are done with it to free up resources

Example.

```
Connection connection = null;
try {
Class.forName("com.mysql.jdbc.Driver");
connection = DriverManager.getConnection(
"jdbc:mysql://127.0.0.1:3306/test","username","password");
} catch (SQLException e) {
e.printStackTrace();
} catch (ClassNotFoundException e) {
System.err.println("Failed to load the JDBC driver");
} finally {
if (connection != null) {
try {
connection.close();
} catch (SQLException e) {
System.err.println("Failed to close the connection.");
}
}
}
```

Statement- Four Creating a Statement

- **There are 3 different types of statements that are supported Statement**
  A basic SQL statement
- **PreparedStatement**
  A precompiled SQL statement
- **CallableStatement**
  Access to stored procedures
- **Just like a connection, we should close the statement when we are done with it.**

Example

```
Connection connection = null;
Statement statement = null;
try {
Class.forName("com.mysql.jdbc.Driver");
connection = DriverManager.getConnection(
"jdbc:mysql://127.0.0.1:3306/test","username","password");
statement = connection.createStatement();
} catch (SQLException e) {
```

```
e.printStackTrace();
} catch (ClassNotFoundException e) {
System.err.println("Failed to load the JDBC driver");
}
// continued on next slide
// continued from previous slide
finally {
if (statement != null) {
try {
statement.close();
} catch (SQLException e) {
System.err.println("Failed to close the statement.");
}
}
if (connection != null) {
try {
connection.close();
} catch (SQLException e) {
System.err.println("Failed to close the connection.");
}
}
}
```

Step Five Obtaining the Resultset. [RGPV/Dec 2011(12)]

We can get a ResultSet back which represents the results of our query.

A ResultSet is returned from executing a query

```
statement.executeQuery("select * from people");
```

You can think of a result set as an iterator over a collection of results that you can walk through

next() – returns a boolean if there is more data and advances to the next item in the collection

There are a lot of methods to get data out of the results set in the following format

getType(int colNum)

getType(String colName)

i.e. String name = rs.getString(“first”);

```
Connection connection = null;
Statement statement = null;
try {
Class.forName("com.mysql.jdbc.Driver");
connection = DriverManager.getConnection(
"jdbc:mysql://127.0.0.1:3306/test","username","password");
statement = connection.createStatement();
ResultSet results =
statement.executeQuery("select * from people");
while (results != null && results.next()) {
```

```
System.out.println(results.getString("first") + " " +
results.getString("last") );
}
} catch (SQLException e) {
e.printStackTrace();
} catch (ClassNotFoundException e) {
System.err.println("Failed to load the JDBC driver");
} // continued on next slide
// continued from previous slide
finally {
if (statement != null) {
try {
statement.close();
} catch (SQLException e) {
System.err.println("Failed to close the statement.");
}
}
if (connection != null) {
try {
connection.close();
} catch (SQLException e) {
System.err.println ("Failed to close the connection.");
}
}
}
```

Step Six Closing Connection

We have to remember to: close Connections, Statements, Prepared Statements and Result Sets. With following statements.

```
con.close();
stmt.close();
pstmt.close();
rs.close()
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Explain the Architecture of JDBC	June 2011 June 2013	08,7
Q-2	How can we move the cursor in movable resultset?	June 2011	12

Unit 4
Topic: Java Database Connectivity (JDBC) ODBC Bridge
Unit 4/Lecture 8

JDBC-ODBC Bride [RGPV/Dec 2009(10) Dec 20012(7), Dec 2014(7]

A **JDBC driver** is a software component enabling a Java application to interact with a database.[1] JDBC drivers are analogous to ODBC drivers, ADO.NET data providers, and OLE DB providers.

To connect with individual databases, JDBC (the Java Database Connectivity API) requires drivers for each database. The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.

JDBC technology drivers fit into one of four categories.

1. JDBC-ODBC bridge
2. Native-API Driver
3. Network-Protocol Driver(MiddleWare Driver)
4. Database-Protocol Driver(Pure Java Driver)

The JDBC type 1 driver, also known as the **JDBC-ODBC bridge**, is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls.

The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the underlying operating system the JVM is running upon. Also, use of this driver leads to other installation dependencies; for example, ODBC must be installed on the computer having the driver and the database must support an ODBC driver. The use of this driver is discouraged if the alternative of a pure-Java driver is available. The other implication is that any application using a type 1 driver is non-portable given the binding between the driver and platform. This technology isn't suitable for a high-transaction environment. Type 1 drivers also don't support the complete Java command set and are limited by the functionality of the ODBC driver.

Sun provides a JDBC-ODBC Bridge driver: sun.jdbc.odbc.JdbcOdbcDriver. This driver is native code and not Java, and is closed source.

If a driver has been written so that loading it causes an instance to be created and also calls DriverManager.registerDriver with that instance as the parameter (as it should do), then it is in the DriverManager's list of drivers and available for creating a connection.

It may sometimes be the case that more than one JDBC driver is capable of connecting to a given URL. For example, when connecting to a given remote database, it might be possible to use a JDBC-ODBC bridge driver, a JDBC-to-generic-network-protocol driver, or a driver supplied by the database vendor. In such cases, the order in which the drivers are tested is significant because the DriverManager will use the first driver it finds that can successfully

connect to the given URL.

First the DriverManager tries to use each driver in the order it was registered. (The drivers listed in jdbc.drivers are always registered first.) It will skip any drivers that are untrusted code unless they have been loaded from the same source as the code that is trying to open the connection.

It tests the drivers by calling the method Driver.connect on each one in turn, passing them the URL that the user originally passed to the method DriverManager.getConnection. The first driver that recognizes the URL makes the connection.

A type 2 driver, on the other hand, is a Java class that calls a native database API (for example, client APIs for Oracle or Sybase). These are not very different from type 1 drivers; they still use an external, non-Java bridge to work with the database.

Type 3 drivers are all Java and use a network transport to communicate with a database server using a database-independent protocol. Since many database vendors offer this network service, a type 3 driver is a flexible option. The driver can be pure Java and still interface with a proprietary database format.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What is JDBC? Explain JDBC-ODBC bridge?	June 2014 June 2012	7,7
Q.2	Explain JDBC-ODBC Bridge and role of the Driver Manager.	Dec 2009	10

BOOK	AUTHOR	PRIORITY
The Complete Reference Java 2	Naughton & Schildt	1
Java- How to Program	Deitel	2
Core Java 2" (Vol I & II)	Horstmann & Cornell	3
Java 2.0	lvan Bayross	4
Beginning Java 2, JDK 5 Ed.	Ivor Horton's	5
Java Programming for the absolute beginners	Russell	6