Power Programming in Java

Session: 5

Stream and File Handling in Java

Objectives

- Define data streams
- Identify purpose of the File class, its constructors and methods
- Describe `DataInput` and `DataOutput` interfaces
- Describe byte stream and character streams in `java.io.package`
- Explain `InputStream` and `OutputStream` classes
- Describe `BufferedInputStream` and `BufferedOutputStream` classes
- Describe Character stream classes
- Describe Console class
- Explain `DeflaterInputStream` and `InflaterOutputStream` classes
- Describe the file system

Stream Classes

- Java works with streams of data. A stream is a sequence of data or logical entity that produces or consumes information.
- A data stream is a channel through which data travels from a source to a destination. This source or destination can be an input or output device, storage media or network computers.
- The standard input/output stream in Java is represented by three fields of the `System` class:

in:	out:	err:
The standard input stream is used for reading characters of data.	The standard output stream is used to typically display the output on the screen or any other output medium.	This is the standard error stream.

- In Java, streams are required to perform all Input/Output (I/O) operations.

Necessity for Stream Classes

- Thus, `Stream` classes help in:

1. Reading input from a stream.
2. Writing output to a stream.
3. Managing disk files.
4. Share data with a network of computers.

Steps for Using Stream Classes

- Steps to read or write data using Input/Output streams:

1. Open a stream that points at a specific data source: a file, a socket, URL, and so on.
2. Read or write data from/to this stream.
3. Close the stream.

- Input and Output streams are abstract classes and are used for reading and writing of unstructured sequence of bytes.
- Other input and output streams are subclasses of basic Input and Output stream class and are used for reading and writing to a file.
- Different types of byte streams can be used interchangeably as they inherit the structure of Input/Output stream class.
- For reading or writing bytes, a subclass of the `InputStream` or `OutputStream` class has to be used respectively.

Reactive Streams

- Reactive programming is a programming paradigm.
- According to the Reactive Manifesto, any reactive application must adhere to four key characteristics:

Responsive	Resilient	Elastic	Message-driven
The system focuses on providing quick and consistent response times and responds in a timely manner.	The system is resilient and remains responsive even when encountered with failure.	The system is elastic and can adjust according to varying workloads.	The system depends on asynchronous message-driven operations.

- In Java, Reactive Streams provide a common API to implement reactive programming.
- `RxJava` and `Akka` Streams are the popular implementation of Reactive Streams.

Implementation of Reactive Streams using Flow API

- Java Flow API from version 9 onwards implements Reactive Streams specification.
- If an application pulls items from the source, it is called a pull model followed by Iterator.
- If the source pushes an item to the application, it is called the push model followed by Observer.
- Flow API Classes and Interfaces:
  - `java.util.concurrent.Flow`
  - `java.util.concurrent.Flow.Publisher`
  - `java.util.concurrent.Flow.Subscriber`
    - Four methods of this Subscriber interface are `onSubscribe`, `onNext`, `onError`, and `onComplete`.
  - `java.util.concurrent.Flow.Subscription`
  - `java.util.concurrent.Flow.Processor`
  - `java.util.concurrent.SubmissionPublisher`

takeWhile, dropWhile, ofNullable, and iterate with Condition [1-2]

Following methods refine streams:

- `takeWhile(Predicate` Interface)

Code Snippet

```
import java.util.stream.Stream;
public class StreamDemo {
public static void main(String[] args) {
Stream.of("H","o","w"," are"," you", ", friend? ","end", "next").
takeWhile(s->!s.equals("end")).forEach(
System.out::print);
 }
}
```

- `dropWhile(Predicate` Interface)

Code Snippet

```
import java.util.stream.Stream;
public class Tester {
public static void main(String[] args) {
Stream.of("g","h","i","","k","l").dropWhile(s-> !s.isEmpty())
.forEach(System.out::print);
System.out.println();
Stream.of("g","h","i","","k","","l").dropWhile(s-> !s.isEmpty())
.forEach(System.out::print);
 }
}
```

takeWhile, dropWhile, ofNullable, and iterate with Condition [2-2]

- iterate

Code Snippet

```
import java.util.stream.IntStream;
public class Tester {
public static void main(String[] args) {
IntStream.iterate(4, x -> x < 11, x -> x+ 4).forEach(System. out::println);
}
```

- ofNullable

Code Snippet

```
import java.util.stream.IntStream;
public class Tester {
public static void main(String[] args) {
long count = Stream.ofNullable(100).count();
System.out.println(count);
count = Stream.ofNullable(null).count();
System.out.println(count);
}
}
```

File Class

- `File` class directly works with files and the file system.
- Files are named using file-naming conventions of host operating system. These conventions are encapsulated using `File` class constants.
- Classes in `java.io` resolve relative pathnames against current user directory, which is named by the system property `user.dir`.
- The directory methods in the `File` class allow creating, deleting, renaming, and listing of directories.
- The `toPath()` method helps to obtain a Path that uses abstract path. A `File` object uses this path to locate a file.
- Constructors of `File` class are:
  - `File(String dirpath)`
  - `File(String parent, String child)`
  - `File(File fileobj, String filename)`
  - `File(URL urlobj)`

Methods in File Class [1-3]

- Methods in `File` class help to manipulate the file on the file system.
- Some of the methods in the `File` class are:

`renameTo(File newname)`	Names the existing File object with the new name specified by the variable newname.
`delete()`	Deletes the file represented by the abstract path name.
`exists()`	Tests the existence of file or directory denoted by this abstract pathname.
`getPath()`	Converts the abstract pathname into a pathname string.
`isFile()`	Checks whether the file denoted by this abstract pathname is a normal file.
`createNewFile()`	Creates a new empty file whose name is the pathname for this file. It is only created when the file of similar name does not exist.
`mkdir()`	Creates the directory named by this abstract pathname.
`toPath()`	Returns a java.nio.file.Path object constructed from the abstract path.
`toURI()`	Constructs a file, URI. This file represents this abstract pathname.

Methods in File Class [2-3]

- Following Code Snippet displays the use of methods of the File class:

Code Snippet

```
. . .
File fileObj = new File("C:/Java/Hello.txt");
System.out.println("Path is: " +fileObj.getPath());
System.out.println("Name is: " +fileObj.getName());
System.out.println("File exists is: " +fileObj.exists());
System.out.println("File is: " +fileObj.isFile());
...
```

- In the code, full path and the filename of the invoking File object is displayed.
- Checks for the existence of the file and returns true if the file exists, false if it does not.
- isFile() method returns true if called on a file and returns false if called on a directory.

Methods in File Class [3-3]

Code Snippet displays use of `FilenameFilter` class to filter files with a specific extension:

Code Snippet

```
import java.io.*;
class FileFilter implements FilenameFilter {
    String ext;
    public FileFilter(String ext) {
    this.ext = "." + ext;
    }
    public boolean accept (File dir, String fName) {
    return fName.endsWith(ext);
    }
}
public class DirList {
    public static void main (String [] args) {
         String dirName = "d:/resources";
         File fileObj = new File ("d:/resources");
         FilenameFilter filterObj = new FileFilter("java");
         String[] fileName = fileObj.list(filterObj);
         System.out.println("Number of files found: " + fileName.length);
         System.out.println("" );
         System.out.println("Names of the files are: " );
         System.out.println("---------------------- " );
         for(int ctr=0; ctr < fileName.length; ctr++) {
            System.out.println(fileName[ctr]);
         }
       }
}
```

FileDescriptor Class

`FileDescriptor` class provides access to the file descriptors that are maintained by the OS when files and directories are being accessed.

In practical use, a file descriptor is used to create a `FileInputStream` or `FileOutputStream` to contain it.

File descriptors should not be created on their own by applications as they are tied to the operating system.

DataInput Interface and DataOutput Interface

Data stream supports input/output of primitive data types and string values. The data streams implement `DataInput` or `DataOutput` interface.

`DataInput` interface has methods for:

- Reading bytes from a binary stream and convert the data to any of the Java primitive types.
- Converting data from Java modified Unicode Transmission Format (UTF)-8 format into string form.

`DataOutput` interface has methods for:

- Converting data present in Java primitive type into a series of bytes and write them onto a binary stream.
- Converting string data into Java-modified UTF-8 format and write it into a stream.

Methods of DataInput Interface

- Important methods in this interface are:
  - readBoolean()
  - readByte()
  - readInt()
  - readDouble()
  - readChar()
  - readLine()
  - readUTF()

- Code Snippet displays use of DataInput interface:

Code Snippet

```
try
{
DataInputStream dis = new DataInputStream(System.in);
double d = dis.readDouble();
int num = dis.readInt();
}
catch(IOException e) {}
. . .
```

Methods of DataOutput Interface

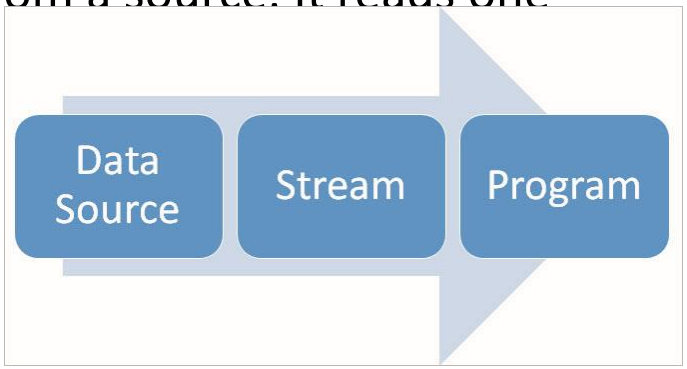
- Important methods in this interface are:
  - `writeBoolean(boolean b)`
  - `writeByte(int value)`
  - `writeInt(int value)`
  - `writeDouble(double value)`
  - `writeChar(int value)`
  - `writeChars(String value)`
  - `writeUTF(String value)`
- Code Snippet displays the use of `DataOutput` interface:

Code Snippet

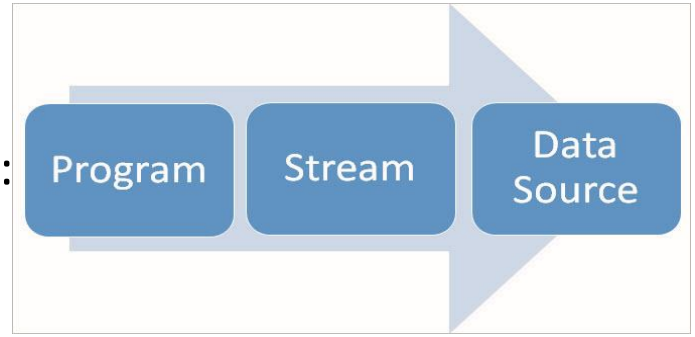
```
try
{
outStream.writeBoolean(true);
outStream.writeDouble(9.95);
. . .
}
catch (IOException e) {}
. . .
```

java.io Package [1-4]

- A stream represents many sources and destinations, such as disk files and memory arrays. It is a sequence of data.
- An I/O Stream represents an input source or an output destination.
- Streams support many forms of data, such as simple bytes, primitive date type, localized characters, and so on.
- A program uses an input stream to read data from a source. It reads one item at a time.
- This figure illustrates the input stream model:



- This figure illustrates the output stream model:



java.io Package [2-4]

Code Snippet displays the working of byte streams using `FileInputStream` class and `FileOutputStream` class:

Code Snippet

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class ByteStreamApp {
public static void main(String[] args) throws IOException {
   FileInputStream inObj = null;
   FileOutputStream outObj = null;
   try {
    inObj = new FileInputStream("c:/java/hello.txt");
    outObj = new FileOutputStream("outagain.txt");
    int ch;
    while ((ch = inObj.read()) != -1) {
     outObj.write(ch);
    }}
    finally {if (inObj != null) {
    inObj.close();}
    if (outObj != null) {
    outObj.close();
    }
    }
   }
}
```

java.io Package [3-4]

Code Snippet

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class CharStreamApp {
 public static void main(String[] args) throws IOException {
  FileReader inObjStream = null;
  FileWriter outObjStream = null;
  try {
   inObjStream = new FileReader("c:/java/hello.txt");
   outObjStream = new FileWriter("charoutputagain.txt");
   int ch;
   while ((ch = inObjStream.read()) != -1) {
    outObjStream.write(ch);
   }
   outObjStream.close();
  } finally {
    if (inObjStream != null) {
     inObjStream.close();
     }
   }
}
```

java.io Package [4-4]

- Character streams act as wrappers for byte streams.
- The character stream manages translation between characters and bytes and uses the byte stream to perform the physical I/O operations.
- Character I/O typically occurs in bigger units than single characters, such as a line that includes a string of characters with a line terminator at the end.
- A line terminator can be any one of following:
  - Carriage-return/line-feed sequence ("\r\n")
  - A single carriage-return ("\r")
  - A single line-feed ("\n").
- `BufferedReader.readLine()` and `PrintWriter.println()` methods:
  - The `readLine()` method returns a line of text with the line.
  - The `println()` method outputs each line on a new line as it appends the line terminator for the current operating system.

ByteStream and Its Different Methods

`ByteStream` classes are used to read bytes from the input stream and write bytes to the output stream.

`ByteStream` classes read/write the data.

These classes are part of the `java.io` package.

The `ByteStream` classes are divided into two types of classes, that is, `InputStream` and `OutputStream`.

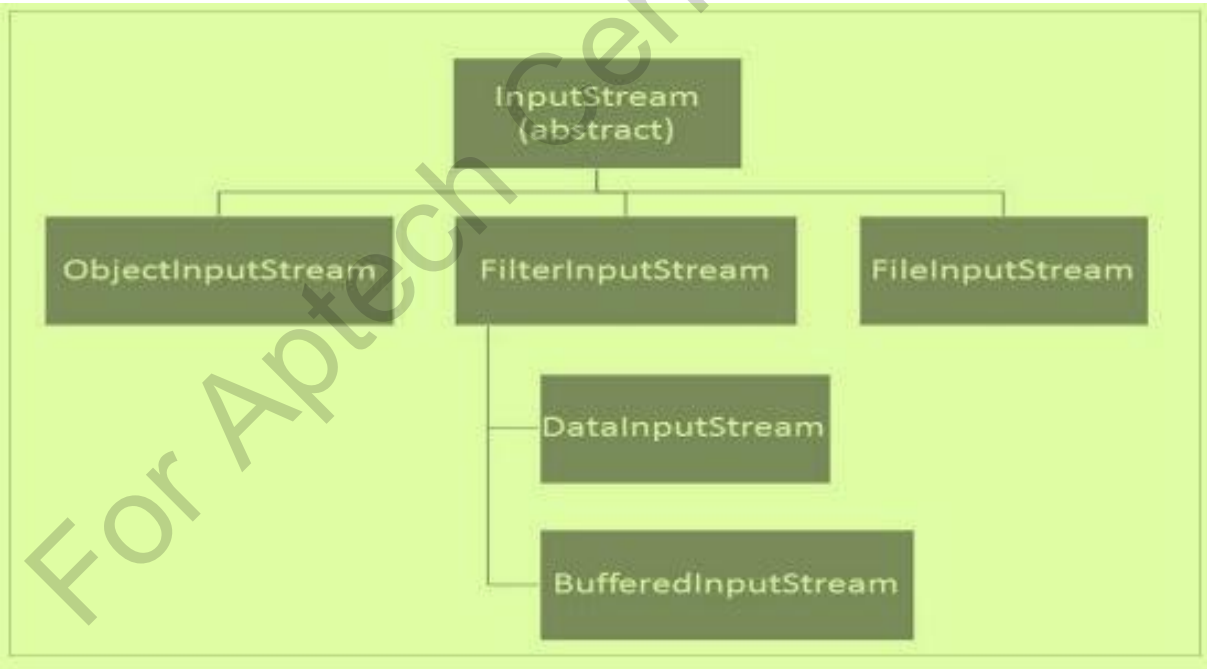
These classes are abstract and the super classes of all the Input/Output stream classes.

InputStream Class and Its Subclasses

InputStream class is an abstract class that defines how streams receive data and is the superclass of all stream classes.

This class has methods to read bytes or array of bytes, mark locations in the stream, find out number of bytes that has been read or available for reading, and so on.

This class is used to read data from an input stream.



Methods of InputStream Class

read(): The **read()** method reads the next bytes of data from the input stream and returns an int value in the range of 0 to 255. The method returns -1 when end of file is reached.

close(): The **close()** method closes the input stream. It releases the system resources associated with the stream.

skip(long n): The skip(long n) method skips n bytes of data while reading from an input stream.

available(): The **available()** method returns the number of bytes that can be read without blocking.

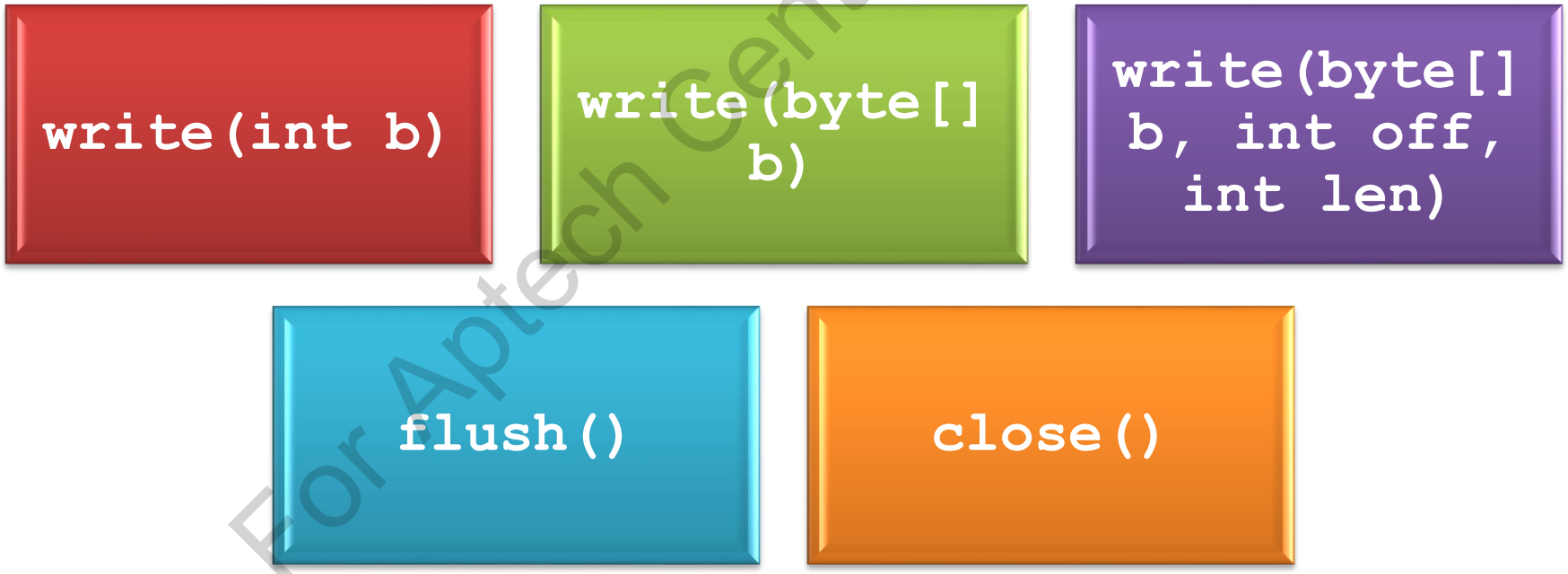
mark(int n): The **mark(int n)** method marks the current position in the stream and will remain valid until the number of bytes specified in the variable, **n**, is read.

reset(): The **reset()** method rests the reading pointer to the previously set mark in the stream.

OutputStream Class and its Subclasses

- The `OutputStream` class is an abstract class that defines the method in which bytes or arrays of bytes are written to streams.
- `ByteArrayOutputStream` and `FileOutputStream` are the subclasses of `OutputStream` class.

Methods in OutputStream class



FileOutputStream Class

- `FileOutputStream` class creates an `OutputStream` that is used to write bytes to a file.
- `FileOutputStream` may or may not create the file before opening it for output and it depends on the underlying platform.
- Some of the commonly used constructors of this class are:
  - `FileOutputStream(String filename)`
  - `FileOutputStream(File name)`
  - `FileOutputStream(String filename, boolean flag)`
  - `FileOutputStream(File name, boolean flag)`

Code Snippet

```
...
String temp = "One way to get the most out of life is to look
upon it as an adventure."
byte [] bufObj = temp.getBytes();
OutputStream fileObj = new FileOutputStream("Thought.txt");
fileObj.write(bufObj);
fileObj.close();
...
```

Filter Streams [1-4]

FilterInputStream class

- provides additional functionality by using an input stream as its basic source of data.

FilterOutputStream class

- These streams are over existing output streams.

FilterInputStream:

- `FilterInputStream` class overrides all the methods of the `InputStream` class that pass all requests to the contained input stream.

- Following are the fields and constructors for `java.io.FilterInputStream` class:
  - `protected InputStream in`
  - `protected FilterInputStream(InputStream in)`

- Following are the methods of this class:

```
mark(int readlimit),markSupported(),read(),available(),
close(),read(byte[] b),reset(),skip(long n),read(byte[] b,
int off,int len)
```

Filter Streams [2-4]

Code Snippet demonstrates the use of `FilterInputStream` class:

Code Snippet

```
package javaioapplication;
...
public class FilterInputApplication {
 public static void main(String[] args) throws Exception {
  InputStream inputObj = null;
  FilterInputStream filterInputObj = null;
   try {
       // creates input stream objects
      inputObj = new FileInputStream("C:/Java/Hello.txt");
      filterInputObj = new BufferedInputStream(inputObj);
      // reads and prints from filter input stream
      System.out.println((char) filterInputObj.read());
      System.out.println((char) filterInputObj.read());
      // invokes mark at this position
      filterInputObj.mark(0);
      System.out.println("mark() invoked");
      System.out.println((char) filterInputObj.read());
      System.out.println((char) filterInputObj.read());
   } catch (IOException e) {
    // prints if any I/O error occurs
    e.printStackTrace();
   } finally {
         //releases system resources associated with the stream
         if (inputObj != null) {
              inputObj.close();
         }
         if (filterInputObj != null) {
              filterInputObj.close();
         }
      }
   }
}
```

Filter Streams [3-4]

FilterOutputStream Class:

- The `FilterOutputStream` class overrides all methods of `OutputStream` class that pass all requests to the underlying output stream.
- Subclasses of `FilterOutputStream` can also override certain methods and give additional methods and fields.
- The `java.io.FilterOutputStream` class includes the protected `OutputStreamout` field, which is the output stream to be filtered.
- `FilterOutputStream(OutputStream` out) is the constructor of this class.
- This creates an output stream filter that exist class over the defined output stream.

Filter Streams [4-4]

Code Snippet demonstrates the use of `FilterOutputStream` class:

Code Snippet

```
public class FilterOutputApplication {
  public static void main(String[] args) throws Exception {
  OutputStream OutputStreamObj = null;
  FilterOutputStream filterOutputStreamObj = null;
  FileInputStream filterInputStreamObj = null;
  byte[] bufObj = {81, 82, 83, 84, 85};
  int i=0;
  char c;
  //encloses the creation of stream objects within try-catch block
  try{
  // creates output stream objects
  OutputStreamObj = new FileOutputStream("C:/Java/test.txt");
  filterOutputStreamObj = new FilterOutputStream(OutputStreamObj);
  // writes to the output stream from bufObj
  filterOutputStreamObj.write(bufObj);
  // forces the byte contents to be written to the stream
  filterOutputStreamObj.flush();
  // creates an input stream object
  filterInputStreamObj = new FileInputStream("C:/Java/test.txt");
  while((i=filterInputStreamObj.read())!=-1)
  { c = (char)i; // converts integer to character
  // prints the character read
  System.out.println("Character read after conversion is: "+ c);
  } }catch(IOException e){
  // checks for any I/O errors
  System.out.print("Close() is invoked prior to write()");
  }finally{
  // releases system resources associated with the stream
  if(OutputStreamObj!=null)
  OutputStreamObj.close();
  if(filterOutputStreamObj!=null)
  filterOutputStreamObj.close();
  }
 }
}
```

BufferedInputStream and BufferedOutputStream Classes

BufferedInputStream Class

BufferedInputStream class allows the programmer to wrap any InputStream class into a buffered stream. It act as a cache for inputs.

It does so by creating the array of bytes which are utilized for future reading.

The simplest way to read data from an instance of BufferedInputStream class is to invoke its read() method.

BufferedInputStream class also supports the mark() and reset() methods.

The function markSupported() will return true if it is supported.

BufferedInputStream class defines two constructors:

- BufferedInputStream(InputStream in)
- BufferedInputStream(InputStream in, int size)

BufferedOutputStream Class

BufferedOutputStream creates a buffer which is used for an output stream.

It provides the same performance gain that is provided by the BufferedInputStream class.

The main concept remains the same, that is, instead of going every time to the operating system to write a byte, it is cached in a buffer.

It is the same as OutputStream except that the flush() method ensures that the data in the buffer is written to the actual physical output device.

The constructors of this class are:

- BufferedOutputStream(OutputStream os)
- BufferedOutputStream(OutputStream os, int size)

Character Streams [1-3]

- Byte stream classes provide methods to handle any type of I/O operations except Unicode characters.
- Character streams provide functions to handle character I/O operations.
- All the methods of this class throw an `IOException`.
- All character stream class are derived from the `Reader` and `Writer` class.

`Reader` Class:

- `Reader` class is an abstract class used for reading character streams.
- The `read()` method returns -1 when end of the file is encountered.
- Following are the two constructors for the `Reader` class:
  - `Reader()` and `Reader(Object lock)`

`Writer` Class:

- `Writer` class is an abstract class and supports writing characters into streams through methods that can be overridden by its subclasses.
- Constructors for the `Writer` class are:
  - `Writer()` and `Writer(Object lock)`

Character Streams [2-3]

`PrintWriter` Class:

- The `PrintWriter` class is a character-based class that is useful for console output.
- It implements all the print methods of the `PrintStream` class.
- The `PrintWriter` class differs from the `PrintStream` class as it can handle multiple bytes and other character sets properly.
- The printed output is tested for errors using the `checkError()` method.
- The `PrintWriter` class also provides support for printing primitive data types, character arrays, strings, and objects.
- It provides formatted output through its `print()` and `println()` methods.
- The `toString()` methods will enable the printing of values of objects.

Character Streams [3-3]

- Following Code Snippet displays the use of the `PrintWriter` class:

Code Snippet

```
. . .
InputStreamReader reader = new InputStreamReader (System.in);
OutputStreamWriter writer = new OutputStreamWriter (System.out);
PrintWriter pwObj = new PrintWriter (writer, true);
. . .
try
{
while (tmp != -1)
{
tmp = reader.read();
ch = (char) tmp;
pw.println ("echo " + ch);
}
}
catch (IOException e)
{
System.out.println ("IO error:" + e );
}
. . .
```

CharArrayReader Class

- `CharArrayReader` class is a subclass of `Reader` class.
- The class uses character array as the source of text to be read.
- `CharArrayReader` class has two constructors and reads stream of characters from an array of characters.
- The constructors of this class are:
  - `CharArrayReader(char arr[])`
  - `CharArrayReader(char arr[], int start, int num)`
- Following Code Snippet displays the use of the `CharArrayReader` class:

Code Snippet

```
. . .
String temp = "Hello World";
int size = temp.length();
char [] ch = new char[size];
temp.getChars(0, size, ch, 0);
CharArrayReader readObj = new CharArrayReader(ch, 0, 5);
```

CharArrayWriter Class [1-2]

- `CharArrayWriter` class is a subclass of `Writer` class. It uses a character array into which characters are written.
- The methods to `CharArray()`, `toString()`, and `writeTo()` method can be used to retrieve the data.
- Following Code Snippet displays the use of the `CharArrayWriter` class:

Code Snippet

```
. . .
CharArrayWriter fObj = new CharArrayWriter();
. . .
String temp = "Hello World";
int size = temp.length();
char [] ch = new char[size];
temp.getChars(0, temp.length(), ch, 0);
fObj.write(ch);
char[] buffer = fObj.toCharArray();
System.out.println(buffer);
System.out.println(fObj.toString());
. . .
```

Serialization

Serialization is the process of reading and writing objects to a byte stream.

An object that implements the Serializable interface will have its state saved and restored using serialization and deserialization facilities.

When a Java object's class or superclass implements the `java.io.Serializable` interface or its subinterface, `java.io.Externalizable`, the Java object becomes serializable.

The `java.io.Serializable` interface defines no methods.

It indicates that the class should be considered for serialization.

If a superclass is serializable, then its subclasses are also serializable.

The only exception is if a variable is transient and static, its state cannot be saved by serialization facilities.

When the serialized form of an object is converted back into a copy of the object, this process is called deserialization.

ObjectOutputStream Class

- `ObjectOutputStream` class extends the `OutputStream` class and implements the `ObjectOutput` interface.
- It writes primitive data types and object to the output stream.

Methods in `ObjectOutputStream` Class:

`writeFloat(float f)`

`writeObject(Object obj)`

`defaultWriteObject()`

Code Snippet

```
. . .
Point pointObj = new Point(50,75);
FileOutputStream fObj = new FileOutputStream("point");
ObjectOutputStream oos = new ObjectOutputStream(fObj);
oos.writeObject(pointObj);
oos.writeObject(new Date());
oos.close();
. . .
```

ObjectInputStream Class [1-3]

`ObjectInputStream` class extends the `InputStream` class and implements the `ObjectInput` interface.

`ObjectInput` interface extends the `DataInput` interface and has methods that support object serialization.

`ObjectInputStream` is responsible for reading object instances and primitive types from an underlying input stream.

It has `readObject()` method to restore an object containing non-static and non-transient fields.

ObjectInputStream Class [2-3]

Following Code Snippet displays the creation of an instance of `ObjectInputStream` class:

Code Snippet

```
. . .
FileInputStream fObj = new FileInputStream("point");
ObjectInputStream ois = new ObjectInputStream(fObj);
Point obj = (Point) ois.readObject();
ois.close();
```

- In the Code Snippet, an instance of `FileInputStream` is created that refers to the file named point.
- An `ObjectInputStream` is created from that file stream.
- The `readObject()` method returns an object which deserialize the object. Finally, the object input stream is closed.
- `ObjectInputStream` class deserializes an object.
- Object to be deserialized must have already been created using the `ObjectOutputStream` class.

ObjectInputStream Class [3-3]

Code Snippet demonstrates the `Serializable` interface:

Code Snippet

```
import java.io.Serializable;
public class Employee implements Serializable{
     String lastName;
     String firstName;
     double sal;
}
public class BranchEmpProcessor {
   public static void main(String[] args) {
           FileInputStream fIn = null;
           FileOutputStream fOut = null;
           ObjectInputStream oIn = null;
           ObjectOutputStream oOut = null;
           try {
            fOut = new FileOutputStream("E:\\NewEmplyee.Ser");
            oOut = new ObjectOutputStream(fOut);
            Employee e = new Employee();
            e.lastName = "Smith";
            e.firstName = "John";
            e.sal = 5000.00;
            oOut.writeObject(e);
            oOut.close();
            fOut.close();
            fIn = new FileInputStream("E:\\NewEmplyee.Ser");
            oIn = new ObjectInputStream(fIn);
            //de-serializing employee
            Employee emp = (Employee) oIn.readObject();
            System.out.println("Deserialized - "+ emp.firstName  +  "  "  +
            emp.lastName + " from NewEmployee.ser");
            } catch (IOException e) {e.printStackTrace();
            } catch (ClassNotFoundException e) {
            e.printStackTrace();
            } finally {
            System.out.println("finally");
            }
     }
}
```

Console Class

- Java provides the Console class to enhance and simplify the development of command line applications.
- The Console class is a part of `java.io` package and has the ability to read text from the terminal without echoing it on the screen.
- The Console object provides input and output of character streams through its Reader and Writer classes.
- The Console class provides various methods to access character-based console device.

Methods in Console Class:

- `format(String fmt, Object... args)`
- `printf(String fmt, Object... args)`
- `reader()`
- `readLine()`
- `readLine(String fmt, Object... args)`

Classes in `java.util.zip`

- Java SE 6 introduced a few changes in JAR and ZIP files.
- In its `java.util.zip` package, Java provides classes that can compress and decompress files.

Classes in java.util.zip package:

- `CheckedInputStream`
- `CheckedOutputStream`
- `Deflater`
- `DeflaterInputStream`
- `DeflaterOutputStream`
- `Inflater`
- `InflaterInputStream`
- `InflaterOutputStream`
- `ZipInputStream`
- `ZipOutputStream`

Deflater and Inflater Classes

- Deflater and Inflater classes extend from `Object` class. These classes are used for compressing and decompressing data.

Deflater Class

- Deflater class compresses the data present in an input stream. It compresses the data using the ZLIB compression library.
- Constructor of the Deflater class is used to create instances of the Deflater class.

  Syntax: `public Deflater()`

- Methods

```
deflate(byte[] buffer)
deflate(byte[] buffer, int offset,
int len)
setInput(byte[] buffer)
setInput(byte[] buffer, int
offset, int len)
finish()
end()
```

Inflater Class

- The Inflater class decompresses the compressed data. This class supports decompression using the ZLIB compression library.

  Syntax: `public Inflater()`

- Methods:

```
inflate(byte[] buffer)
inflate(byte[] buffer, int offset,
int len)
setInput(byte[] buffer)
setInput(byte[] buffer, int
offset, int len)
end()
```

DeflaterInputStream and DeflaterOutputStream Classes

- DeflaterInputStream and DeflaterOutputStream classes are inherited from the FilterInputStream and FilterOutputStream classes respectively.

DeflaterInputStream

DeflaterInputStream class reads the source data from an input stream and then, compresses it in the 'deflate' compression format.

This class provides its own constructors and methods. The constructor creates a new input stream with the buffer size and the specified compressor.

Methods:

```
read()
read(byte[] buffer, int offset, int buffSize) throws IOException
close()
boolean markSupported()
int available()
long skip(long n)
```

DeflaterOutputStream

DeflaterOutputStream class reads the source data, compresses it in the 'deflate' compression format and then, writes the compressed data to a predefined output stream.

It also acts as the base for other types of compression filters, such as GZIPOutputStream.

Methods:

```
write(int buffer)
write(byte[] buffer, int offset, int buffSize
deflate()
close()
finish()
```

InflaterInputStream and InflaterOutputStream Classes

- The `InflaterInputStream` and `InflaterOutputStream` classes are inherited from the `FilterInputStream` and `FilterOutputStream` classes respectively.

InflaterInputStream Class

The `InflaterInputStream` class reads the compressed data and decompresses it in the 'deflate' compression format.

Syntax: `public InflaterInputStream(InputStream in)`

InflaterOutputStream Class

The `InflaterOutputStream` class reads the compressed data, decompresses the data stored in the deflate compression format, and then, writes the decompressed data to an output stream.

Syntax: `public InflaterOutputStream(OutputStream out)`

java.nio Package

Java New Input/Output (NIO) API package was introduced in 2002 with J2SE 1.4 to enhance the input/ output processing tasks in the Java application development.

The primary goal for both NIO and NIO.2 remains same, that is, to enhance the I/O processing tasks in the Java application development. Its use can cut down the time required for certain common I/O operations

Another aspect of NIO is its attention to application expressivity. NIO is platform dependent. Its ability to enhance application performance depends on following OS, Specific JVM, Mass storage characteristics, Data, and Host virtualization context

Central features of the NIO APIs

- Charsets and their Associated Decoders and Encoders
- Buffers
- Channels of Various Types
- Selectors and Selection Keys

File Systems, Paths, and Files

A file system stores and organizes files on media, typically hard drives. Such files can be easily retrieved. Every file has a path through the file system.

File Systems

- Typically, files are stored in a hierarchical structure, where there is a root node.
- File systems can have one or more root directories. File systems have different characteristics for path separators.

Path

- Every file is identified through its path. In other words, the path specifies a unique location in the file system.
- It starts from the root node. Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as D:\.
- The character that separates the directory names is called the delimiter.
- This is specific to the file system.

Path Interface

`java.nio.file.Path` interface object can help to locate a file in a file system.

Typically, the interface represents a system dependent file path.

It provides the entry point for the file and directory manipulation.

A Path is hierarchical. It includes a sequence of directory and file name elements.

These are separated by a delimiter.

There could be a root component. This represents a file system hierarchy.

Name of a file or directory is the name element that is extreme far from the root of the directory hierarchy.

Other name elements include directory names.

A Path can represent following:

- A root
- A root and a sequence of names
- One or more name elements

Tracking File System and Directory Changes

Static methods in the `java.nio.file.Files` class perform primary functions for the Path objects. The methods in the class can identify and automatically manage symbolic links.

Following are various File operations:

- Copying a file or directory
- Moving a file or directory
- Checking a file or directory
- Deleting a file or directory
- Listing a Directory's Content
- Creating and Reading Directories
- Reading and Writing from Files
- Reading a File by Using Buffered Stream I/O
- Writing a File by Using Buffered Stream I/O

How to Read/Write Strings to and from Files? [1-2]

`FileWriter` class is used to write data in to text files of Java. Both of these are Character Stream classes.

`FileInputStream` and `FileOutputStream` should not be used to read/write textual data since, they are Byte stream classes.

`FileWriter` helps to generate a file and write characters in it.

The `FileWriter` class accepts default character encoding and the default byte-buffer size. `FileOutputStream` is used to write raw bytes.

How to Read/Write Strings to and from Files? [2-2]

FileReader

- `FileReader` reads each character from a text file.
- `FileReader` class assumes that the default character encoding and the byte-buffer size are appropriate.
- `FileReader` is used to read character streams, whereas `FileInputStream` reads raw bytes data streams.

Summary

- A stream is a logical entity that produces or consumes information.
- Data stream supports input/output of primitive data types and String values.
- `InputStream` is an abstract class that defines how data is received.
- The `OutputStream` class defines the way in which output is written to streams.
- File class directly works with files on the file system.
- A buffer is a temporary storage area for data.
- Serialization is the process of reading and writing objects to a byte stream.
- Java provides the Console class to enhance and simplify command line applications.
- The Console class provides various methods to access character-based console device.
- Java in its `java.util.zip` package provides classes that can compress and decompress files.
- The Deflater and Inflater classes extend from the Object class.
- The `DeflaterInputStream` and `DeflaterOutputStream` classes are inherited from the `FilterInputStream` and `FilterOutputStream` classes respectively.
- The `InflaterInputStream` and `InflaterOutputStream` classes are inherited from the `FilterInputStream` and `FilterOutputStream` classes respectively.
- NIO is platform dependent.
- A file system stores and organizes files on media, typically hard drives.