Core Java

Java I/O & File Handling

Lesson Objectives

- To understand the following topics:
  - Streams
  - Byte Stream I/O hierarchy
  - Character Streams: Readers and Writers
  - The File Class
  - Object Serialization
  - Scanning and Formatting

Overview of I/O Streams

- Most programs need to access external data.
- Data is retrieved from an input source.
  - Program results are sent to output destination.

Figure 8-1: A program uses an input stream to read data from a source, one item at a time

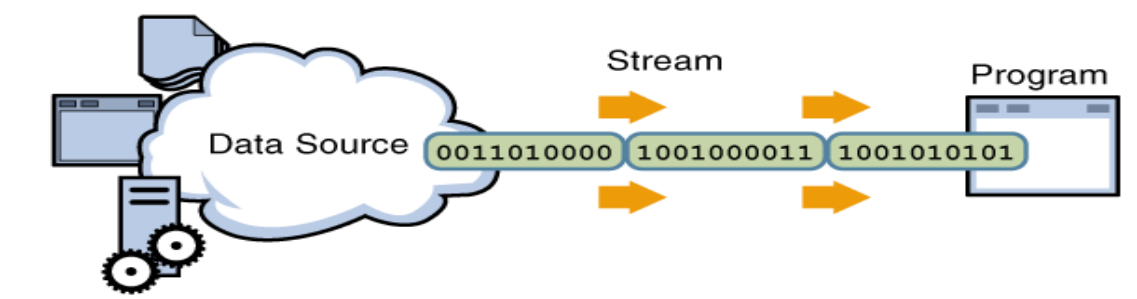
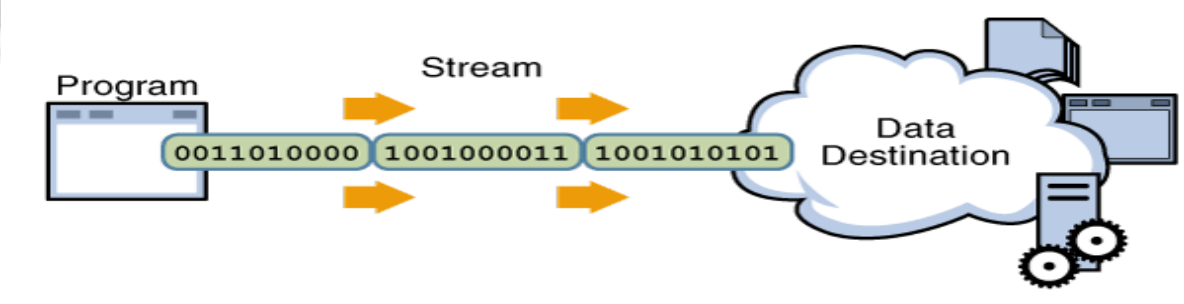


Figure 8-2:A program uses an output stream to write data to a destination, one item at time



What is a Stream?

- Abstraction that consumes or produces information.
- Linked to source and destination.
- Java implements streams within class hierarchies defined in the *java.io* package.
- An *input* stream acts as a *source* of data.
- An *output* stream acts as a *destination* of data.

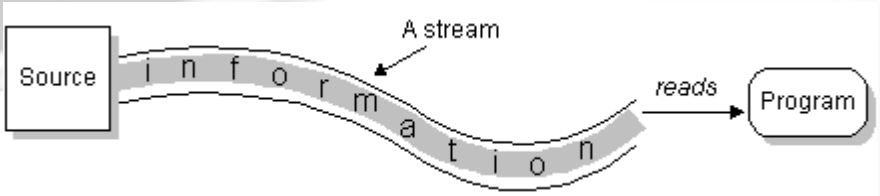


Figure 8-3: (a) Input Stream

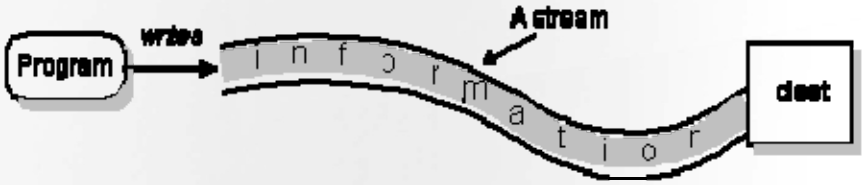


Figure 8-3:(b) Output stream

Types of Streams

- *Byte Streams:* Handle I/O of raw binary data.
- *Character Streams:* Handle I/O of character data. Automatic translation handling to and from a local character.
- *Buffered Streams:* Optimize input and output with reduced number of calls to the native API.
- *Data Streams:* Handle binary I/O of primitive data type and String values.
- *Object Streams:* Handle binary I/O of objects.
- *Scanning and Formatting:* Allows a program to read and write formatted text.

Byte Stream I/O Hierarchy

InputStream Class - Methods

Method	Description
close()	Closes this input stream and releases any system resources associated with the stream.
int read()	Reads the next byte of data from the input stream.
int read(byte[] b)	Reads some number of bytes from the input stream and stores them into the buffer array *b*.
int read(byte[] b, int off, int len)	Reads up to *len* bytes of data from the input stream into an array of bytes.

OutputStream Class - Methods

Method	Description
close()	Closes this output stream and releases any system resources associated with this stream.
flush()	Flushes this output stream and forces any buffered output bytes to be written out.
write(byte[] b)	Writes *b.length* bytes from the specified byte array to this output stream.
write(byte[] b, int off, int len)	Writes *len* bytes from the specified byte array starting at offset off to this output stream.
write(int b)	Writes the specified byte to this output stream.

InputStream Subclasses

Classname	Description
DataInputStream	A filter that allows the binary representation of java primitive values to be read from an underlying inputstream
BufferedInputStream	A filter that buffers the bytes read from an underlying input stream. The buffer size can be specified optionally.
FilterInputStream	Superclass of all input stream filters. An input filter must be chained to an underlying inputstream.
ByteArrayInputStream	Data is read from a byte array that must be specified
FileInputStream	Data is read as bytes from a file. The file acting as the input stream can be specified by File object, or as a String
PushBackInputStream	A filter that allows bytes to be “unread “ from an underlying stream. The number of bytes to be unread can be optionally specified.
ObjectInputStream	Allows binary representation of java objects and java primitives to be read from a specified inputstream.
PipedInputStream	It reads many bytes from PipedOutputStream to which it must be connected.
SequenceInputStream	Allows bytes to be read sequentially from two or more input streams consecutively.

FileInputStream and FileOutputStream Example

```
import java.io.*;
class CopyFile  {
                FileInputStream fromFile;
                FileOutputStream toFile;
      public void init(String arg1, String arg2) {
                //Assign the files
      try{
                fromFile = new FileInputStream(arg1);
                toFile = new FileOutputStream(arg2);
      } catch (FileNotFoundException fnfe) {
                System.out.println("Exception: " + fnfe);
      } catch (IOException ioe) {
                System.out.println("Exception: " + ioe);
      } catch (ArrayIndexOutOfBoundsException aioe) {
                  System.out.println("Exception: " + aioe);
      }
}
```

FileInputStream and FileOutputStream Example

```
public void copyContents() {// copy bytes
    try {
            int i = fromFile.read();
            while ( i != -1)
               { //check the end of file
                 toFile.write(i);
                 i = fromFile.read();
                }
               } catch (IOException ioe)
           { System.out.println("Exception: " + ioe);}
}
```

FileInputStream and FileOutputStream Example

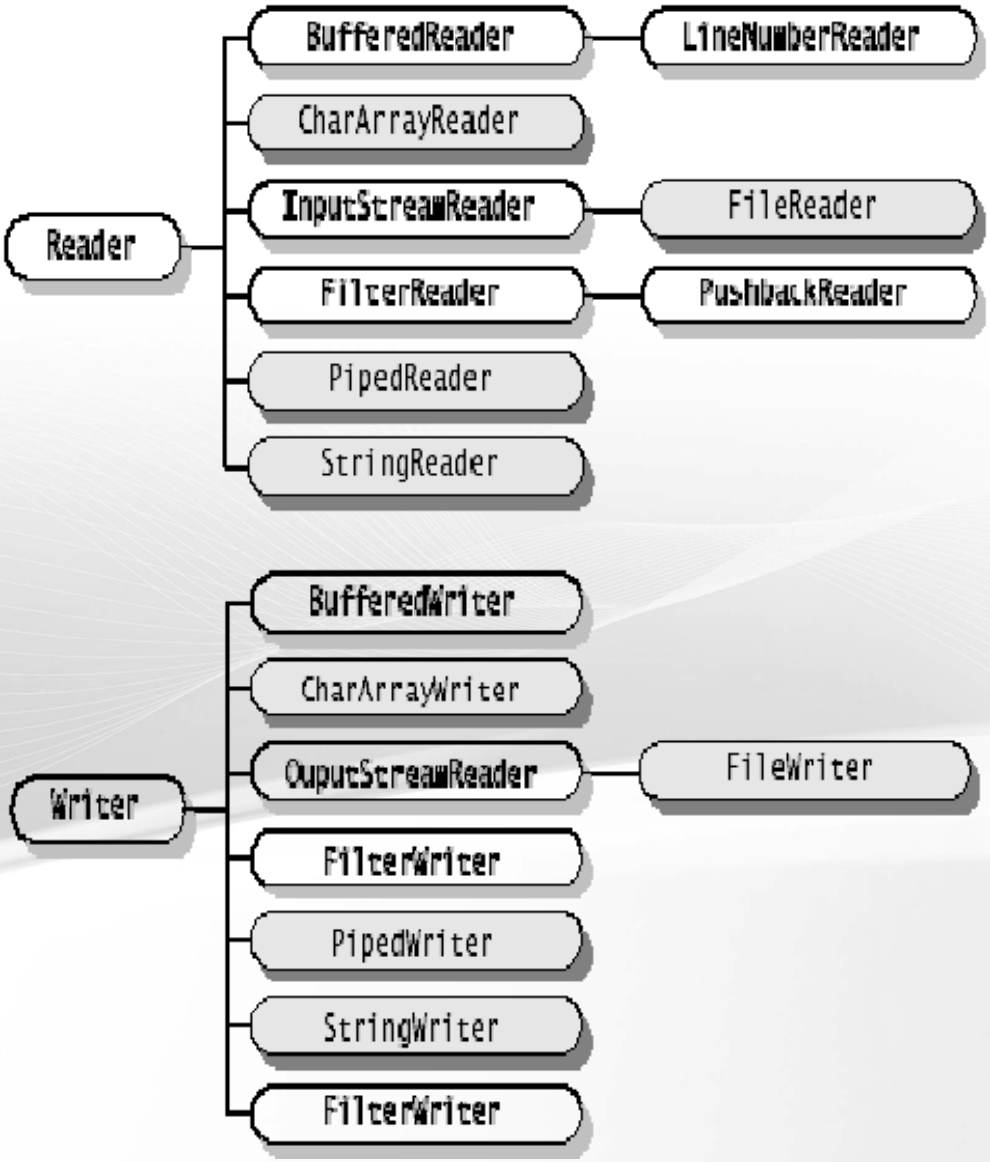
```
public void closeFiles() {//close the files
                                try{
                    fromFile.close();
                    toFile.close();
                    } catch (IOException ioe)
                                { System.out.println("Exception: " + ioe);
                }
        }

public static void main(String[] args){
                CopyFile c1 = new CopyFile();
                c1.init(args[0], args[1]);
                c1.copyContents();
                c1.closeFiles(); } }
```

Demo : Input/Output Streams

Demo:

CopyFile.java

Character Stream I/O Hierarchy



Character Streams Classes - Readers and Writers

- All Character Stream classes are descended from *Reader* and *Writer* Abstract classes.
- Java represents characters internally in the 16-bit Unicode character encoding.
- *Reader* is an input character stream that reads a sequence of Unicode characters.
- *Writer* is an output character stream that writes a sequence of Unicode characters

Reader Class - Methods

Method	Description
int read() throws IOException	reads a byte and returns as an int
int read(char b[])throws IOException	reads into an array of chars b
int read(char b[], int off, int len) throws IOException	reads *len* number of characters into char array *b*, starting from offset *off*
long skip(long n) throws IOException	Can skip n characters.

Writer Class - Methods

Method	Description
void write(int c)throws IOException	writes a byte.
void write(char b[])throws IOException	writes from an array of chars b
void write(char b[], int off, int len) throws IOException	writes *len* number of characters from char array *b*, starting from offset *off*
void write(String b, int off, int len) throws IOException	writes *len* number of characters from string *b*, starting from offset *off*

FileReader and FileWriter Example

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;
        try {
            inputStream = new FileReader(“sampleinput.txt");
            outputStream = new FileWriter(“sampleoutput.txt");
            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
```

FileReader and FileWriter Example

```
} finally {
   if (inputStream != null) {
 inputStream.close();
 }
    if (outputStream != null)
 {

outputStream.close();
 }

 }
 } }
```

Buffered Input Output Stream

- An unbuffered I/O means each read or write request is handled directly by the underlying OS.
  - Makes a program less efficient.
    - Each such request often triggers disk access, network activity, or some other relatively expensive operation.
  - Java implements buffered I/O Streams to reduce this overhead.
    - Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty.
    - Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

Buffered Stream

- A program can convert a un-buffered stream into buffered using the *wrapping idiom:*
  - Un-buffered stream object is passed to the constructor of a buffered stream class.
  - Example

```
inputStream =
new BufferedReader(new FileReader(“sampleinput.txt"));

outputStream =
new BufferedWriter(new FileWriter(“sampleoutput.txt"));
```

Demo : Buffered Stream

- **Demo:**

  CharEncode.java

DataInputStream / DataOutputStream Classes

- Data Streams:
  - Support binary I/O of primitive data type values:
    - boolean, char, byte, short, int, long, float, and double as well as String.
  - All data streams implement either the *DataInput* interface or the *DataOutput* interface.
    - DataInputStream and DataOutputStream are most widely-used implementations.

DataInputStream / DataOutputStream Methods

Read	Write	Type
readBoolean	writeBoolean	boolean
readChar	writeChar	char
readByte	writeByte	byte
readShort	writeShort	short
readInt	writeInt	int
readLong	writeLong	long
readFloat	writeFloat	float
readDouble	writeDouble	double
readUTF	writeUTF	String (in UTF format)

DataInputStream / DataOutputStream Classes Demo

```
public static void writeData(double[] data, String file)throws
    IOException {
    OutputStream fout = new FileOutputStream(file);
    DataOutputStream out = new DataOutputStream(fout);
    out.writeInt(data.length);
    for (double d : data) {     out.writeDouble(d); }
    out.close();
 }
public static double[] readData(String file) throws IOException {
    InputStream fin = new FileInputStream(file);
    DataInputStream in = new DataInputStream(fin);
    double[] data = new double[in.readInt()];
    for (int i = 0; i < data.length; i++) {      data[i] = in.readDouble(); }
    in.close();
    return data;
 }
```

DataInputStream / DataOutputStream Classes Demo

- **Demo:**

JavaPrimitiveValues.java

STDIO.java

ObjectInputStream / ObjectOutputStream

- Object streams support I/O of objects:
  - Support I/O of primitive data types.
  - Object has to be *Serializable* type.
  - *Object Classes:* ObjectInputStream, ObjectOutputStream
    - Implement ObjectInput and ObjectOutput, which are sub interfaces of DataInput and DataOutput.
  - An object stream can contain a mixture of primitive and object values.

Object Serialization

- *Object Serialization:*
  - Object Serialization allows an object to be transformed into a sequence of bytes that can be later re-created (deserialized) into an original object.
  - Process to read and write objects.
  - Provides ability to read or write a whole object to and from a raw byte stream.
  - Use object serialization in the following ways:
    - *Remote Method Invocation (RMI):* Communication between objects via sockets.
    - *Lightweight persistence:* Archival of an object for use in a later invocation of the same program.

Object Serialization - Example

```
import java.io.*;
// This is a serializable object
class Employee implements Serializable  {
        String name;
        int age;
        double salary;
        Employee(String name, int age, double salary) {
                this.name = name;
                this.age = age;
                this.salary = salary;
        }
        public void showDetails() {
                System.out.println("Name          :" + name);
                System.out.println("Age           :" + age);
                System.out.println("Salary        :" + salary);
        }
}
```

Object Serialization - Example

```
class ObjectSerializationDemo {
      void writeData() {
        Employee db[] = {
                        new Employee("Sachin",25,12000.56),
                        new Employee("Rahul",24,12670.78),
                        new Employee("Hritik",28,16000.89)
                        };
       try  {
            FileOutputStream out = new FileOutputStream("emp-obj.dat");
            ObjectOutputStream sout = new ObjectOutputStream(out);
                        for (int i = 0; i < db.length ; i++ ) {
                                    sout.writeObject(db[i]);
                        }
                        sout.close();
                        } catch (IOException ioe) {
                                    ioe.printStackTrace();
```

Object Serialization - Example

```
        }
            }
            void readData() {
try         {
                    FileInputStream in = new FileInputStream("emp-obj.dat");
                    ObjectInputStream sin = new ObjectInputStream(in);
                    Employee e = (Employee) sin.readObject();
                    e.showDetails();
                    e = (Employee) sin.readObject();
                    e.showDetails();
                    e = (Employee) sin.readObject();
                    e.showDetails();
                    sin.close();
```

Object Serialization - Example

```
            } catch (IOException ioe) {
                        ioe.printStackTrace();
            } catch (ClassNotFoundException cnfe) {
                        cnfe.printStackTrace();
    }
        }
        public static void main(String  args[]) {
            ObjectSerializationDemo impl = new ObjectSerializationDemo();
                impl.writeData();
                impl.readData();
        }
}
```

Demo : Object Serialization

- **Demo:**

ObjectSerializationDemo.java

Scanning and Formatting

- Java platform provides two APIs to translate to and from neatly formatted data.
  - Scanner API
    - Breaks input into individual tokens associated with bits of data.
  - Formatting API
    - Assembles data into nicely formatted, human-readable form.

Scanner Class

- Prior to Java 1.5 getting input from the console involved multiple steps.
- Java 1.5 introduced the *Scanner* class to simplify console input.
- Also reads from files and Strings (among other sources).
- Used for powerful pattern matching.
- Scanner is in the Java.util package.
  - Hence, type the following code:

```
import java.util.Scanner;
```

Scanner Class

- Scanner(File source):
  - Constructs a new Scanner that produces values scanned from the specified file.
- Scanner(InputStream source):
  - Constructs a new Scanner that produces values scanned from the specified input stream.
- Scanner(Readable source):
  - Constructs a new Scanner that produces values scanned from the specified source.
- Scanner(String source):
  - Constructs a new Scanner that produces values scanned from the specified string.

How to use Scanner class?

- Scanner class basically parses input from the source into tokens by using delimiters to identify the token boundaries.
- The default delimiter is whitespace.
- Example:

```
Scanner sc = new Scanner (System.in);
int i = sc.nextInt();
System.out.println("You entered" + i);
```

Demo : Scanner class

- **Demo:**

ParseString.java

Scanner Methods

- String next()
- boolean nextBoolean()
- byte nextByte()
- double nextDouble()
- float nextFloat()
- int nextInt()
- String nextLine()
- long nextLong()
- short nextShort()

- boolean hasNext()
- boolean hasNextBoolean()
- boolean hasNextByte()
- boolean hasNextDouble()
- boolean hasNextFloat()
- boolean hasNextInt()
- boolean hasNextLong()
- boolean hasNextShort()

Format Method

Formats multiple arguments based on a format string.
Example:

```
public class Root2 {
   public static void main(String[] args) {
            int i = 2;
 double r = Math.sqrt(i);
System.out.format("The square root of %d is %f.%n", i,
r);
}

}
```

Output: The square root of 2 is 1.414214.