Core Java

IO Streams

Objective

At the end of this session, you will be able to:

- Write programs using Byte Streams
- Serialize Objects
- Externalize Objects
- Write programs using Character Streams

Agenda

- Introduction to I/O Streams
- Types of Streams
- Byte Streams
- Object Serialization
- Object Externalization
- Character Streams

File class:

- **File** class does not operate on streams.
- It deals directly with files and the file system.
- It describes the properties of a file.

- File is an object which is a representation of file and directory pathnames.
- 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*)

directoryPath - path name of the file,

filename - name of the file or subdirectory,

dirObj - **File** object that specifies a directory,

uriObj - **URI** object that describes a file.

- Eg.: File f1 = new File("/");
  File f2 = new File("/",“fn.bat");

- **File** defines many methods that obtain the standard properties of a **File** object.

```
f2.list();          f2.length();
f2.exists();        f2.canWrite();
f2.isFile();        f2.getParent();
f2.isDirectory();   f2.canRead();
f2.getName();           f2.getPath();
```

- **// 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 size: " + f1.length() + " Bytes");
    }
}
```

Output:

```
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 size: 695 Bytes

- **File** includes two useful utility methods.
  - **renameTo( ) =>** boolean renameTo(File *newName*);
    - It will return **true** upon success and **false** if the file cannot be renamed
  - **delete( ) =>** boolean delete( );
    - **delete( )** returns **true** if it deletes the file and **false** if the file cannot be removed

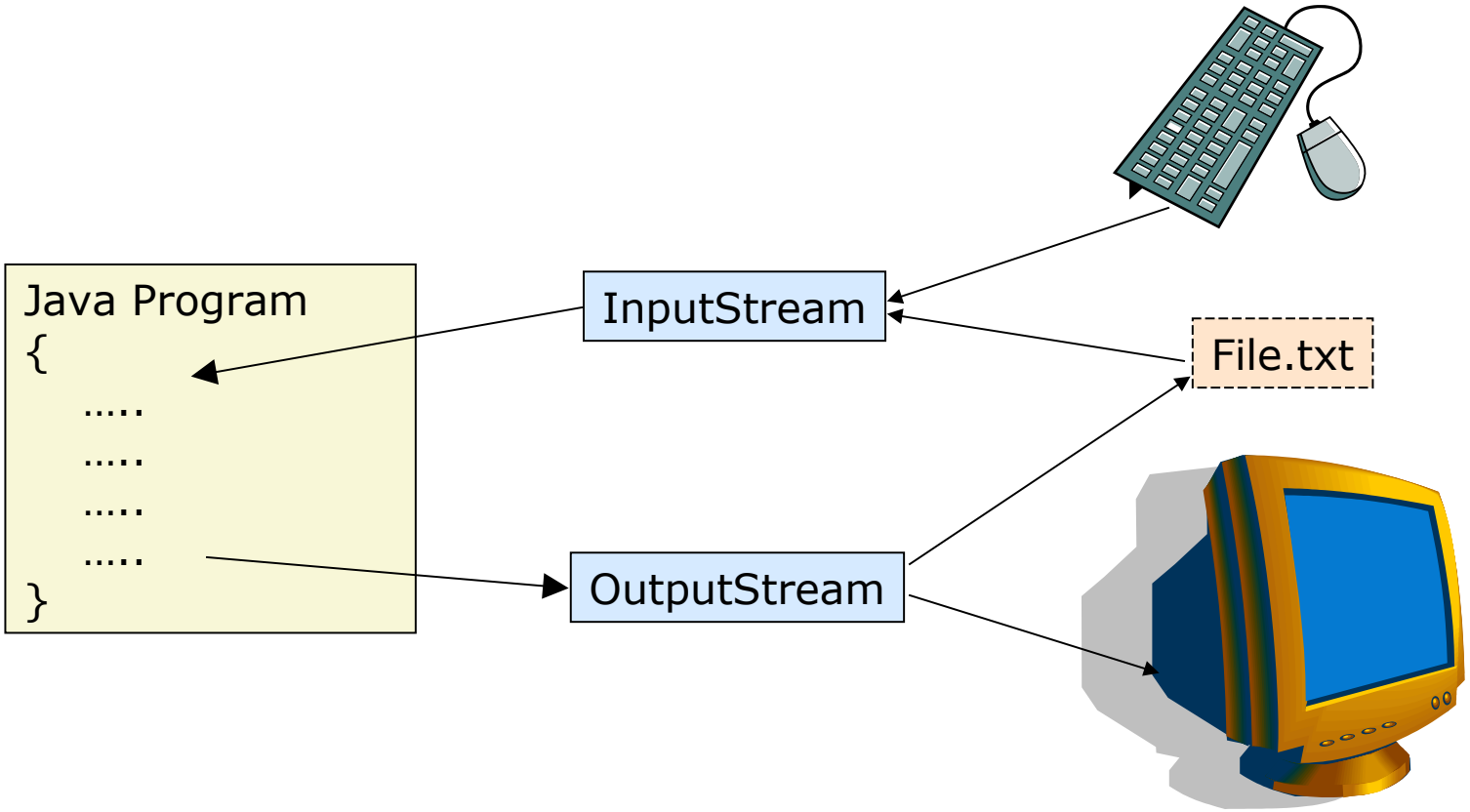
Introduction to Streams

How do we take an input from keyboard?

- Input Streams are used to read data from any data source like keyboard, socket, file etc.
- Output Streams are used to write data to any data destination like console, socket, file etc.

What is I/O Stream?

- An abstract representation of data connected to some input or output device is called as Stream

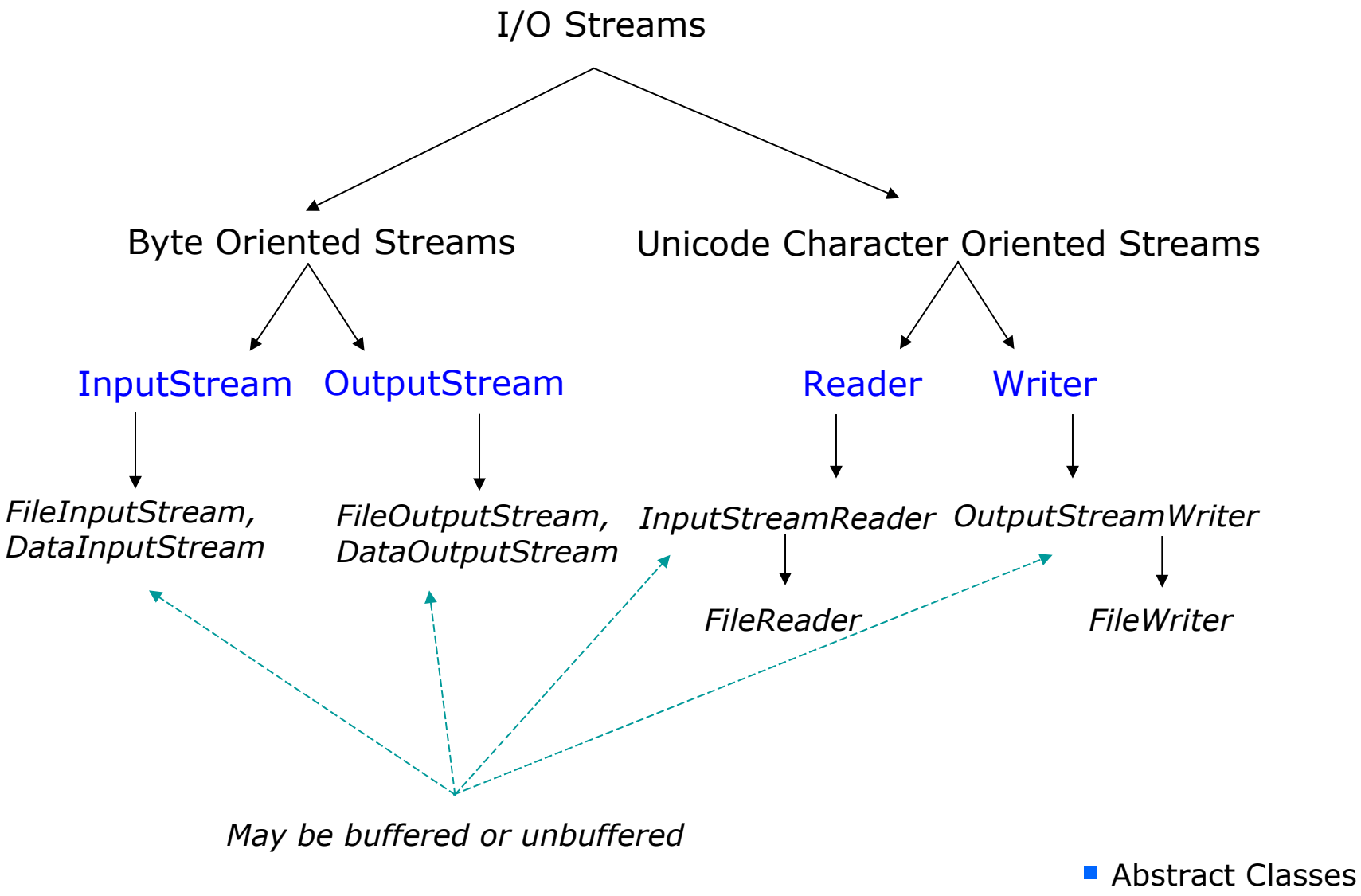


Types of Streams

Two Types of Stream Classes:

- Byte Stream
  - Usually works for bytes & binary objects
  - *InputStream* and *OutputStream* are the abstract classes which represent Byte Streams
- Character Stream
  - Usually works for Characters & Strings
  - Follows the Unicode
  - Introduced to cater to the needs of internationalization
  - *Reader* and *Writer* are the abstract classes which represents Character Streams

Types of Streams (Contd…)



Give this a Try…

1. Character Stream uses _____ standard to represent characters.

2. To write the data to the file which stream is to be used?

Byte Streams

- *FileOutputStream* & *FileInputStream* classes:
  - These are sub classes of *OutputStream* and *InputStream* classes respectively
  - Used to write & read binary data and /or binary object to and from the data source

```
FileOutputStream fos = new FileOutputStream("abc.txt");

FileInputStream fis = new FileInputStream("abc.txt");
```

Byte Streams (Contd…)

- *FileInputStream* object is used to read data from the file

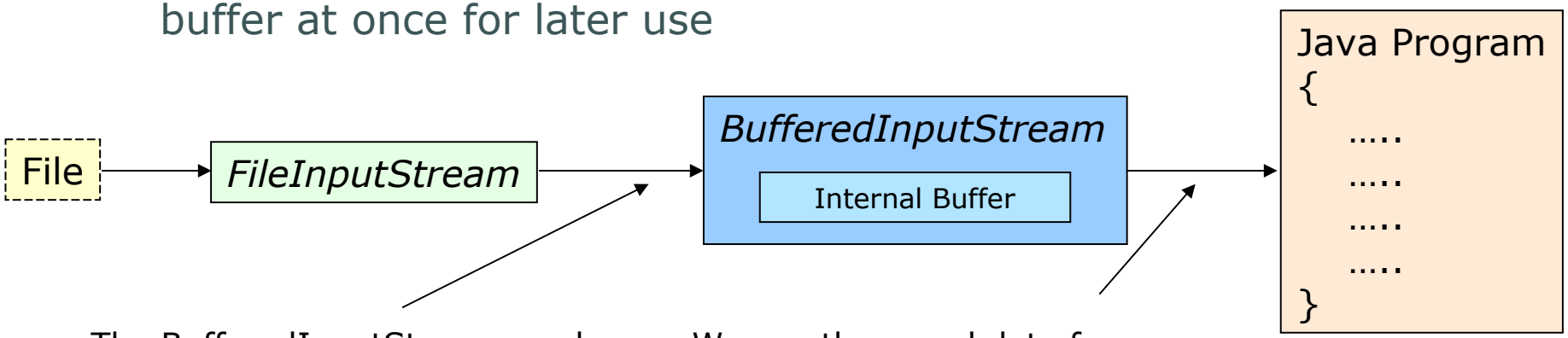
```
FileInputStream testFile;

try {
        testFile = new FileInputStream("test.dat");

         while((nextByte = testFile.read()) != -1)
        {
        System.out.println(nextByte);
        }
      }
     catch(IOException e)
     {
    System.out.println("Error reading file + e ");
     }
```

read() returns -1 on encountering end of file

ReadWriteFile.java

Byte Streams (Contd…)

- *BufferedInputStream* & *BufferedOuputStream* classes:
  - Subclasses of *InputStream* & *OutputStream* classes respectively
  - We can wrap a *BufferedInputStream* around the *FileInputStream* for reading & storing large chunks of data in a buffer at once for later use



The BufferedInputStream reads data from the File in large chunks and stores the data in an internal buffer

We can then read data from the *BufferedInputStream*. Data is read from the buffer instead of directly from the file on each read

Byte Streams (Contd…)

BufferedInputStream adds buffering to *FileInputStream* object

```
BufferedInputStream bufferedFile;

try {
      bufferedFile = new BufferedInputStream(new
      FileInputStream("test.dat");

       while((nextByte = bufferedFile.read()) != -1)
      {
            System.out.println(nextByte);
      }
    }
   catch(IOException e)
   {
  System.out.println("Error reading file + e ");
    }
```

BufferedTest.java

Give this a Try…

1. Buffer Stream is used to read large amount of data as compare to File Stream? State True / False

1. Can we append the data in the existing file?

1. If the file to be read does not exists, which exception gets thrown?

Byte Streams (Contd…)

- Data I/O Streams
  - We may want an even higher level of abstraction and wish to read & write data to and from streams in the form of primitive data variables (rather than just bytes or characters)
  - Java has built in stream classes to automatically handle converting this information into the necessary raw bytes that a stream can use

Byte Streams (Contd…)

- *DataInputStream* & *DataOutputStream* classes:
  - Allow to read and write primitive data types to input and output streams respectively

```
BufferedOutputStream bufStream;
DataOutputStream dataStream;
try {
   bufStream = new BufferedOutputStream(new
         FileOutputStream("file.out");
   dataStream = new DataOutputStream(bufStream);
   dataStream.writeInt(5);
} catch(IOException e) {
   System.out.println("Error writing to file " + e );
} finally {
  // Write code in try/catch to close the streams
}
```

DataStream.java

Object Serialization

- The process of writing the state of an object to a byte stream
- Saves the state of an Object to any data destination like file
- This may later be restored by the process of Deserialization
- Only an object that implements the *Serilizable* interface can be saved & restored by the serilization facilities
- The *Serializable* interface defines no members; It is simply used to indicate that a class may be serialized
- All subclasses of a *serializable* class are also *serializable*
- transient declared & static variables are not saved by this

Object Serialization (Contd…)

- *ObjectOutputStream* & *ObjectInputStream* classes
  - Subclasses of *InputStream* & *OutputStream* classes
  - Same functionality as *DataInputStream* & *DataOutputStream*
  - Also include support for reading and writing objects data via the *readObject()* & *writeObject()* methods

```
ObjectOutputStream oos = new ObjectOutputStream(
new FileOutputStream("abc.txt"));
oos.writeObject();

ObjectInputStream ois = new ObjectInputStream(
new FileInputStream("abc.txt"));
ois.readObject();111
```

Serial.java

Object Externalization

- Does some processing before storing & after retrieving objects, if desired

  e.g. Encrypting passwords before storing

- We can control the process of serialization by implementing the *Externalizable* interface instead of *Serializable*

- *Externalizable* extends the original Serializable interface & adds *writeExternal()* and *readExternal(),* which must be implemented

- These two methods are called automatically after object's serialization & deserialization

Object Externalization (Contd…)

```
public class ExternalData implements Externalization
{
        public ExternalData()
        {
        System.out.println("In the constr..");
        }
        public void writeExternal(ObjectOuput out)
        throws IOException
         {
          System.out.println("In the WriteExternal");
         }
        public void readExternal(ObjectInput in)
         throws IOException,ClassNotFoundException
         {
          System.out.println("In the ReadExternal");
         }
}
```

Give this a Try…

1. To read the specific primitive data type which Stream class we have to use?

1. Which interface is used to control the Serialization?

Character Streams

- Two types of Character Stream classes:
  1. Reader
  2. Writer

- *FileReader* and *FileWriter* classes:

  - Subclasses of *Reader* & *Writer* class

  - Used to read and write characters or strings from a data source like file

```
FileWriter fw = new FileWriter("abc.txt",true);

FileReader fr = new FileReader("abc.txt") throws
IOException
```

Character Streams (Contd…)

- *BufferedReader* & *BufferedWriter* classes:
  - Provides buffering to Character streams
  - Subclasses of *Reader* & *Writer* classes
  - *BufferedReader* is used to read data from console & files
- *InputStreamReader* class:
  - Serves as a wrapper for any *InputStream* object
  - Converts the raw bytes as they are read from the *InputStream* and serves them to the user as Unicode characters

Character Streams (Contd…)

- Reading data from console:
  - We can wrap *InputStreamReaders* around *InputStreams* to make them useful in reading character data
  - *BufferedReader* provides a *readLine()* method for additional functionality

```
BufferedReader stdin = new BufferedReader(new
        InputStreamReader(System.in));
try {
   String input = stdin.readLine();
   System.out.println("The input from the command line

                        was " + input);
} catch(IOException e) {
   System.err.println("Error reading data");
}
```

BRReadLines.java

Give this a Try…

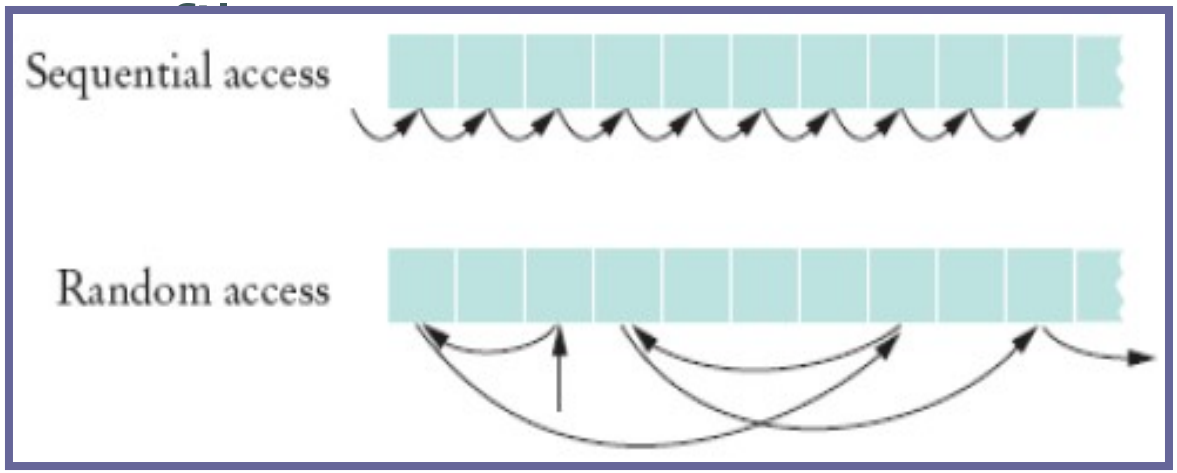
1. Which method is used to read the data line by line from the console?

1. Reader class deals with which kind of encoding?

Random Access Files

Random Access Files

- Random access files are files in which records can be accessed in any order
  - Also called direct access files
  - More efficient than sequential access

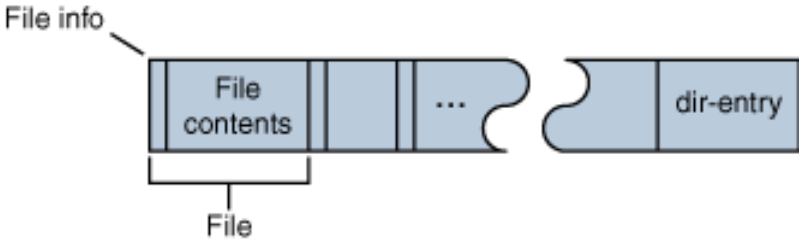


Need for Random Access Files

- Real-time applications require immediate response
  - Example: respond to customer query about a bill
  - Sequencing through records for account is time-intensive
- Random (immediate) access meets real-time need
  - Directly read from or write to desired record

Example

- Consider the zip format. A ZIP archive contains files and is typically compressed to save space. It also contain a directory entry at the end that indicates where the various files contained within the ZIP archive begin



Accessing a specific file using sequential access

- Open the ZIP archive.
- Search through the ZIP archive until you locate the file you want to extract.
- Extract the file.
- Close the ZIP archive.

On an average, we have to read half of the zip archive to find the required file

Accessing a specific file using random access

- Open the ZIP archive.
- Seek to the directory entry and locate the entry for the file you want to extract from the ZIP archive.
- Seek (backward) within the ZIP archive to the position of the file to extract.
- Extract the file.
- Close the ZIP archive.

  This is more efficient as you read only the directory entry and file that you want to extract.

RandomAccessFiles class

- The RandomAccessFile class contains the same read(), write() and close() methods as Input and OutputStream.
- Also contains seek() that lets you select a beginning position within the file before reading or writing data.
- Includes capabilities for reading and writing primitive-type values, `byte` arrays and strings

RandomAccessFile Class

- NOT compatible with the stream/reader/writer models
- With a random-access file, you can seek to the desired position and then read and write an amount of bytes
- Only support seeking relative to the beginning of the file
  - Not relative to current position of file pointer
  - However there are methods that report the current position

Methods to support seeking

long **getFilePointer**()
Returns the current offset in this file.

long **length**()
Returns the length of this file.

void **seek**(long pos)
Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.

Constructor Summary

RandomAccessFile(File file, String mode)

Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.

RandomAccessFile(String name, String mode)

Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

- The mode should be either “r” or “rw”
  - No “w”

Constructor Summary

- When a RandomAccessFile is created in read-only mode a FileNotFoundException is generated

- When a RandomAccessFile is created in read-write a zero length file will be created

File Pointer

- RandomAccessFile supports *file pointer* which indicates the current location in the file.
- When the file is first created, the file pointer is set to 0, indicating the beginning of the file.
- Calls to the read and write methods adjust the file pointer by the number of bytes read or written.

Manipulate file pointer

- RandomAccessFile contains three methods for explicitly manipulating the file pointer.
- int skipBytes(int) — Moves the file pointer forward the specified number of bytes
- void seek(long) — Positions the file pointer just before the specified byte
- long getFilePointer() — Returns the current byte location of the file pointer

- To move the file pointer to a specific byte

```
f.seek(n);
```

- To get current position of the file pointer.

```
long n = f.getFilePointer();
```

- To find the number of bytes in a file

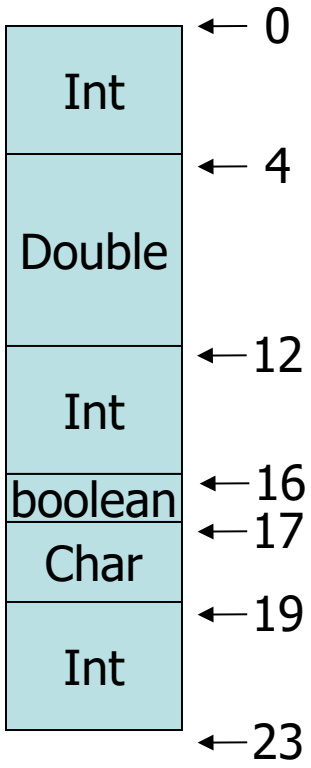
```
long filelength = f.length();
```

Writing Example

```
import java.io.*;
public class RandomAccess {
  public static void main(String args[]) throws
   IOException {
   RandomAccessFile myfile = new
   RandomAccessFile("rand.dat", "rw");
   myfile.writeInt(120);
   myfile.writeDouble(375.50);
    myfile.writeInt('A'+1);
    myfile.writeBoolean(true);
    myfile.writeChar('X');
```

```
// set pointer to the beginning of file and read
  next two items
        myfile.seek(0);
        System.out.println
       (myfile.readInt());
        System.out.println

(myfile.readDouble());
//set pointer to the 4th item and read it
        myfile.seek(16);
        System.out.println
```

```
(myfile.readBoolean());
// Go to the end and "append"

an integer 2003
        myfile.seek(myfile.length());
        myfile.writeInt(2003);
        // read 5th and 6th items
        myfile.seek(17);
        System.out.println a
```



```
(myfile.readChar());
        System.out.println

(myfile.readInt());
        System.out.println("File length:
  "+myfile.length());
        myfile.close();
    }
}
```

Output

```
C:\WINDOWS\system32\cmd.exe

C:\Program Files\Java\jdk1.6.0\bin>javac RandomAccess.java

C:\Program Files\Java\jdk1.6.0\bin>java RandomAccess
120
375.5
true
x
2003
File length: 27

C:\Program Files\Java\jdk1.6.0\bin>
```

Reading Example

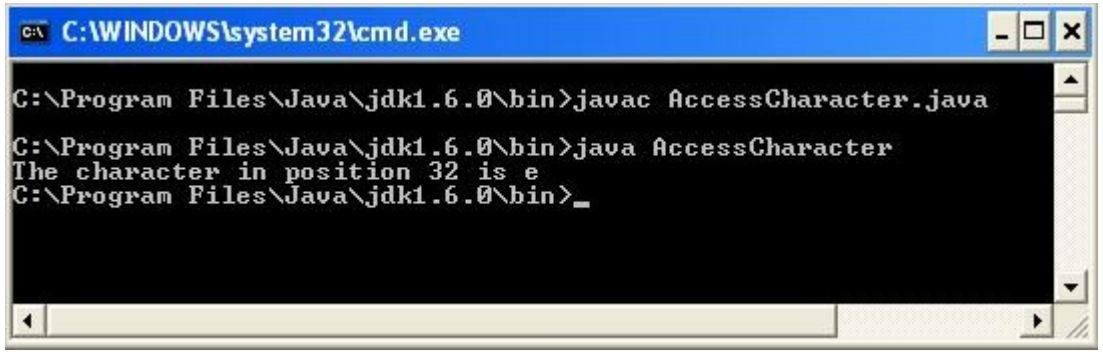
- RandomAccessFile creates random access files
  - Contains familiar read( ), write( ), open( ), close( )
  - seek( ) method selects start position within a file before read or write
  - Places a file pointer at the selected location
  - File pointer holds byte number of next file position
  - Ex: locate 32nd character in file, which may then be read

Reading Example

```
import java.io.*;
public class AccessCharacter
{
   public static void main(String[] args) throws
  IOException
   {
    OutputStream ostream;
    int c;
    RandomAccessFile inFile = new
  RandomAccessFile("AQuote.txt","r");
    ostream = System.out;
    int pos = 32;
```

```
try
    {  inFile.seek(pos);
       c = inFile.read();
       System.out.print("The character in position
  " + pos + " is ");
       ostream.write(c);
    }a
    catch (IOException e)
    { System.out.println();
      inFile.close();
      ostream.close();
    }
  }
}
```

Output


C:\WINDOWS\system32\cmd.exe
C:\Program Files\Java\jdk1.6.0\bin>javac AccessCharacter.java
C:\Program Files\Java\jdk1.6.0\bin>java AccessCharacter
The character in position 32 is e
C:\Program Files\Java\jdk1.6.0\bin>


AQuote.txt - Notepad
File Edit Format View Help
The aim of life is self-development. To realize one's nature perfectly -
that is what each of us is here for.

References

- Object Oriented Software Development using Java- 2nd edition - Xiaoping Jia
- Java Programming 3rd edition – Joyce Farrell
- www.java.sun.com
- www.javaworld.com

Summary

In this session, we have covered:

- What is I/O Stream
- Types of Streams
- Byte Streams
- Object Serialization
- Object Externalization
- Character Streams

Thank You