

# CHAPTER 11

## Basic I/O

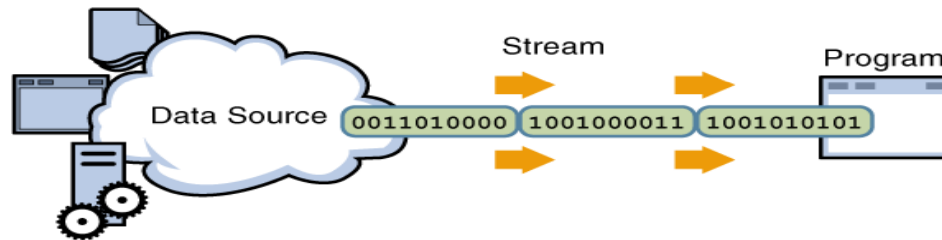
# Contents

- ◆ I/O Streams
  - Byte Streams
  - Character Streams
  - Buffered Streams
  - Scanning and Formatting
  - I/O from the Command Line
  - Data Streams
  - Object Streams
- ◆ File I/O
  - File Objects
  - Random Access Files

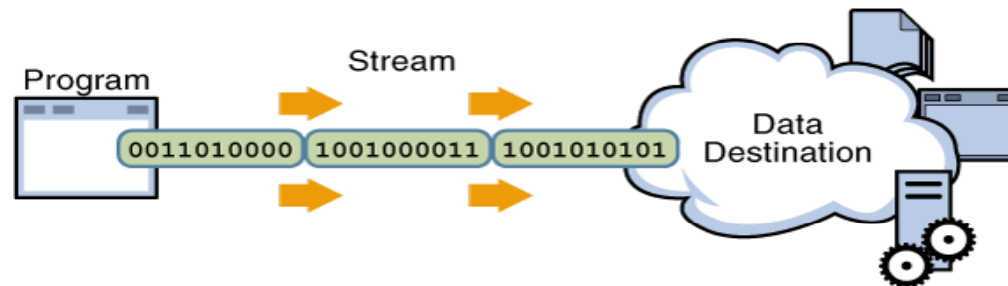
# I/O Streams

## ◆ I/O Stream

- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.



Reading information into a program



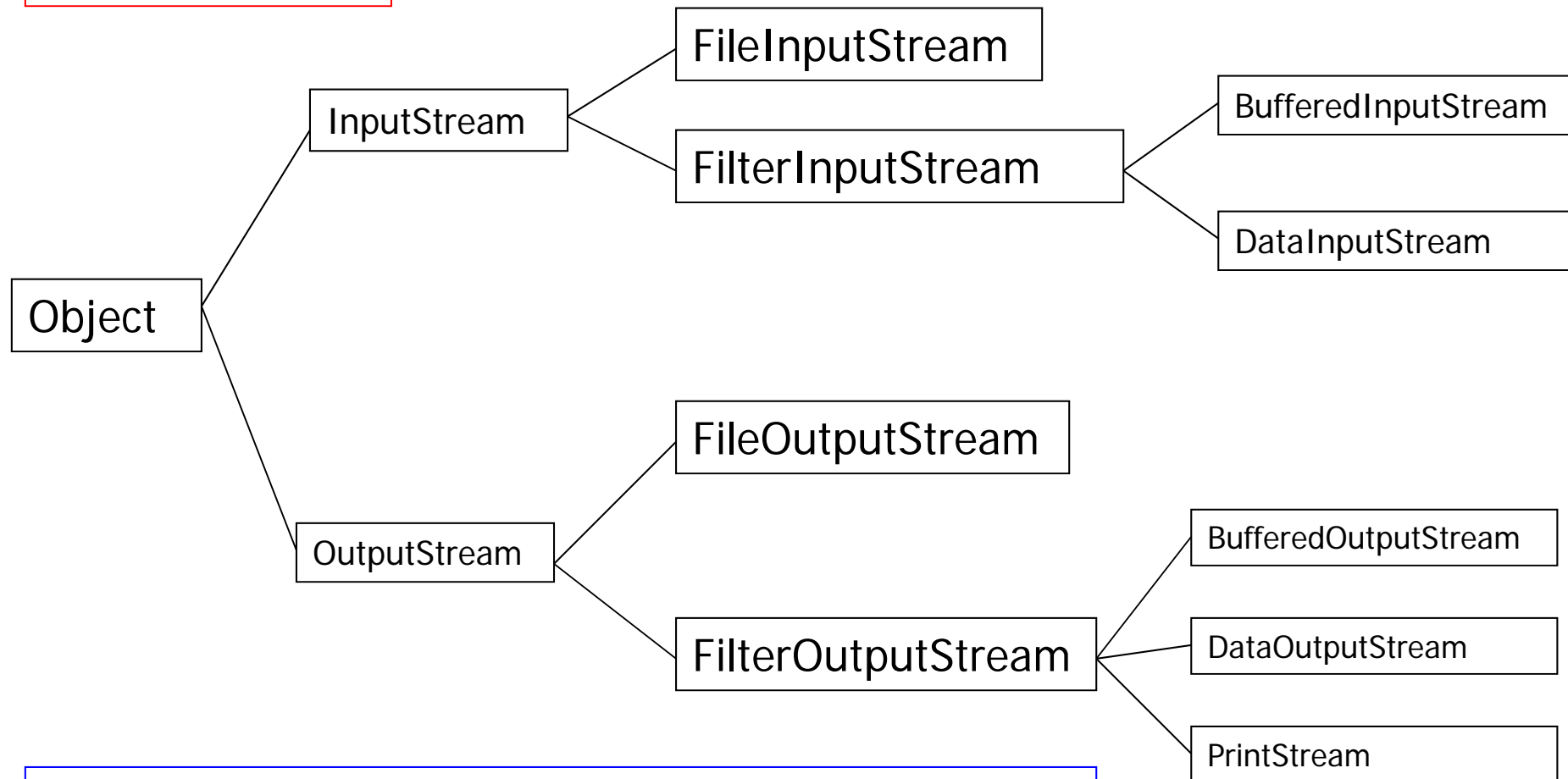
Writing information from a program

# Streams Overview

- ◆ Two major parts in the java.io package: character (16-bit UTF-16 characters) streams and byte (8 bits) streams
- ◆ I/O is either text-based or data-based (binary)
- ◆ Input streams or output streams → byte stream
- ◆ Readers or Writers → character streams
- ◆ Five group of classes and interfaces in java.io
  - The general classes for building different types of byte and character streams
  - A range of classes that define various types of streams – filtered, piped, and some specific instances of streams
  - The data stream classes and interfaces for reading and writing primitive values and strings
  - For Interacting with files
  - For object serialization

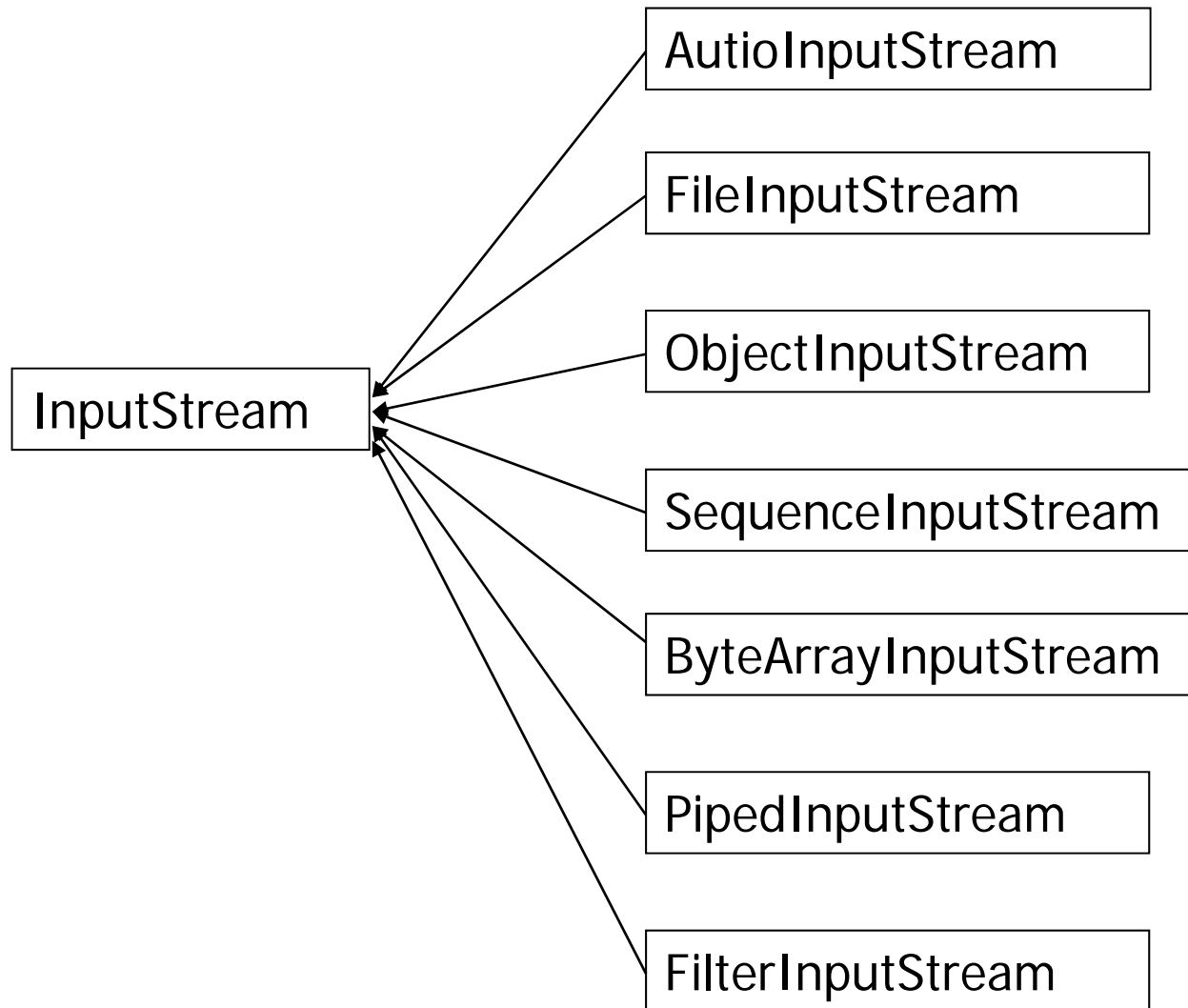
# Byte Streams (Binary Streams)

**For low-level I/O**

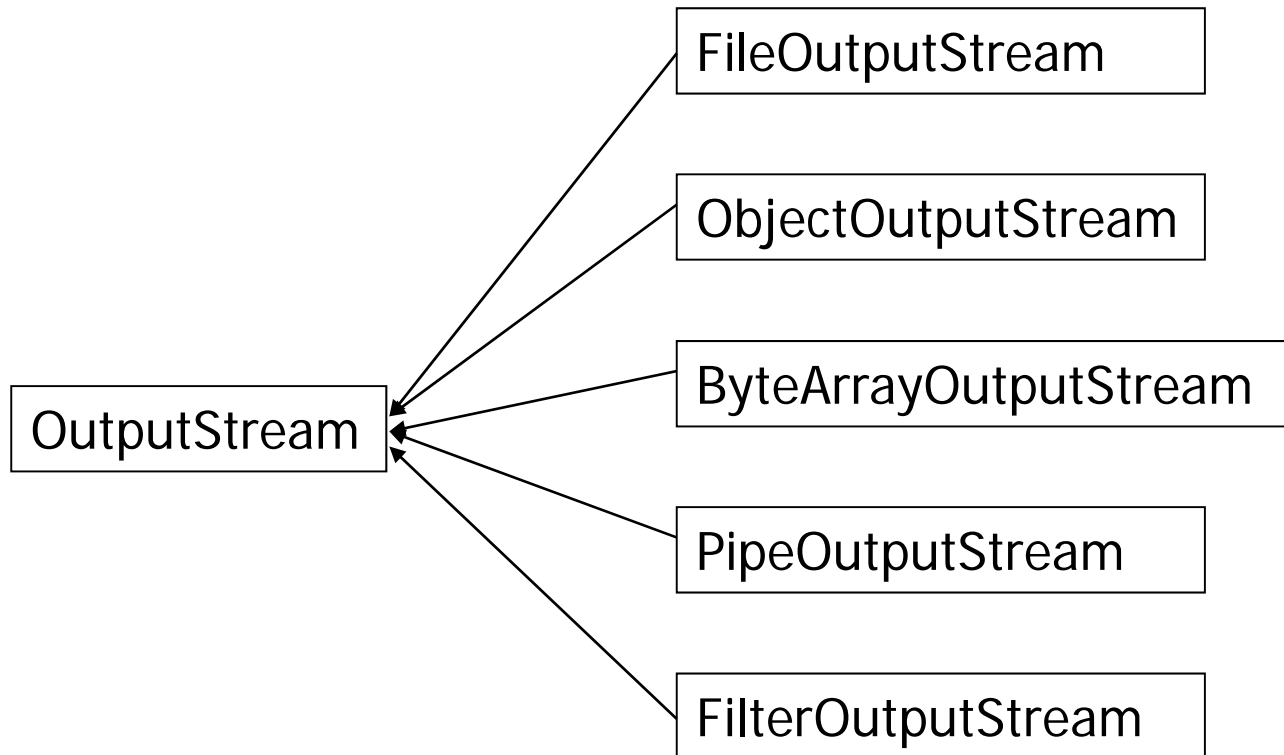


Programs use byte streams to perform input and output of 8-bit bytes (octets).

# Byte Streams



# Byte Streams



# Byte Streams

## ◆ CopyBytes.java

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

### URL for this code

<http://java.sun.com/docs/books/tutorial/essential/io/examples/CopyBytes.java>

FileInputStream  
(byte stream)  
object from the  
input file

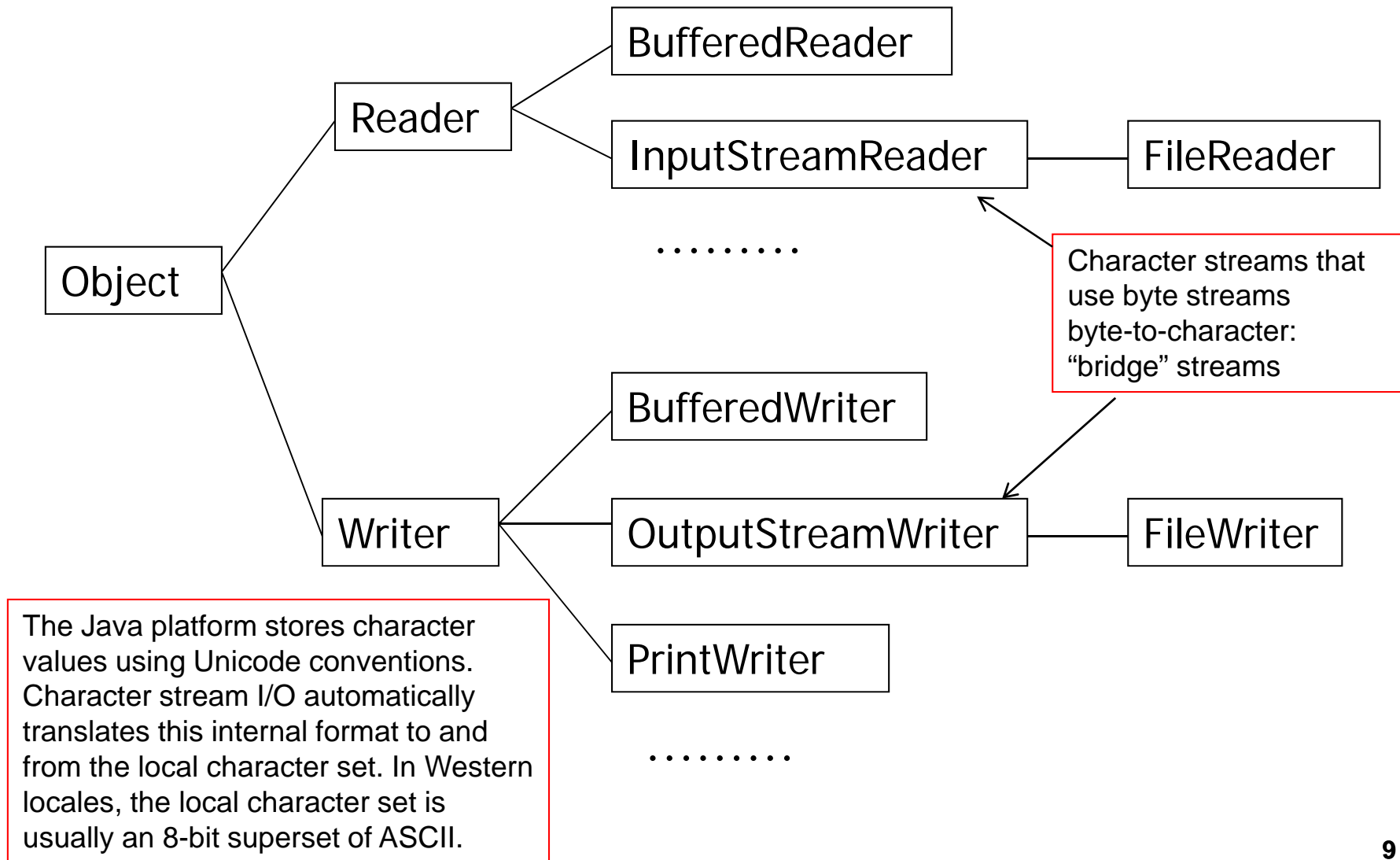
Reads a byte of  
data from this  
input stream.

### Input Example (xanadu.txt)

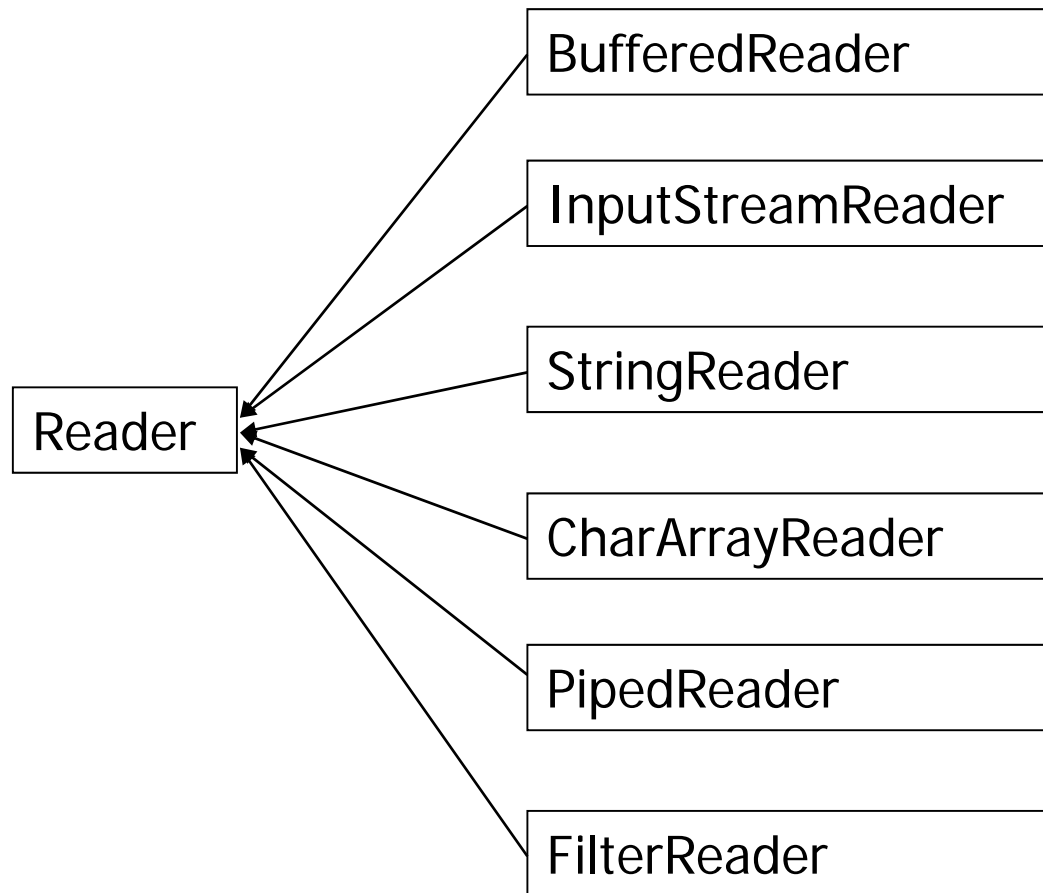
In Xanadu did Kubla Khan  
A stately pleasure-dome decree:  
Where Alph, the sacred river, ran  
Through caverns measureless to man  
Down to a sunless sea.



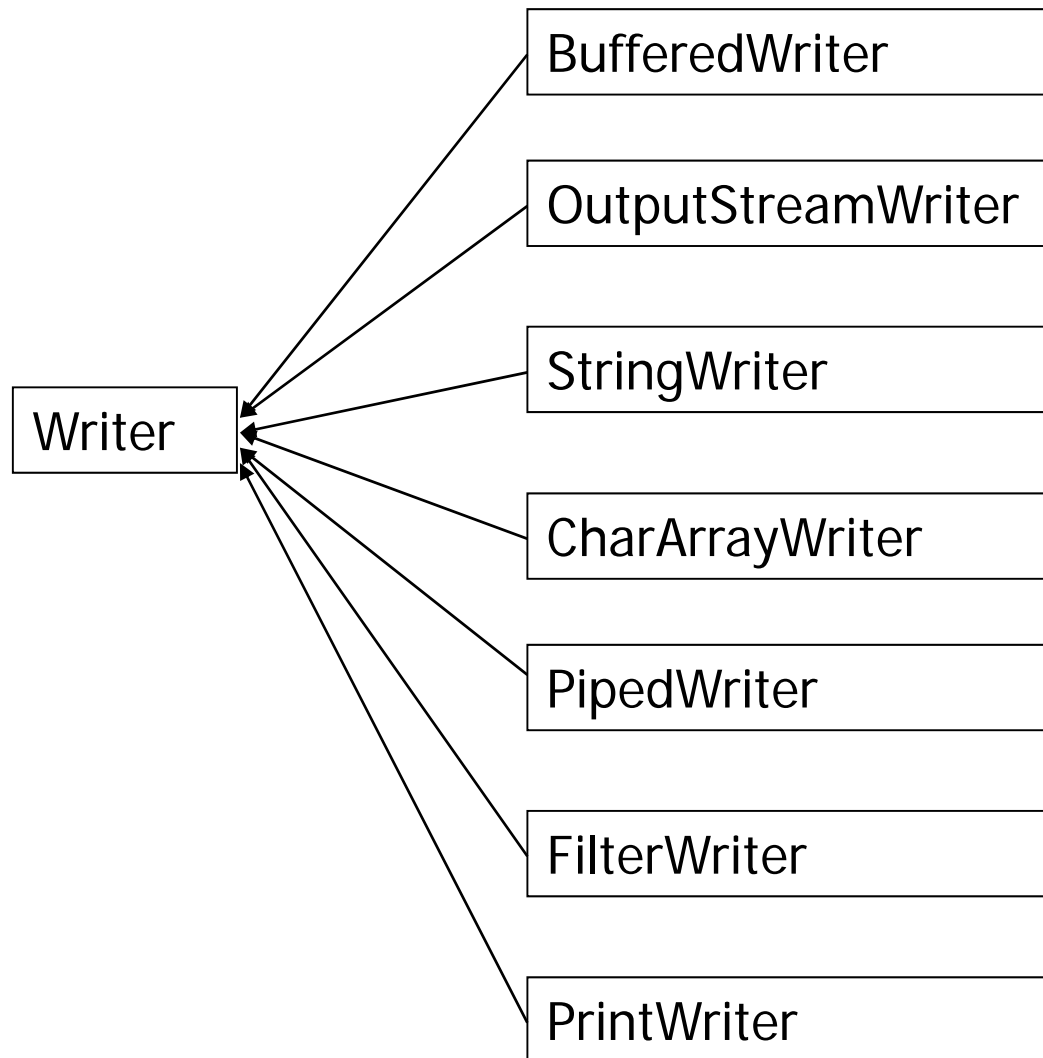
# Character Streams



# Character Streams



# Character Streams



# Character Streams

## ◆ CopyCharacters.java

```
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("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");
            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

What is the difference between the byte streams and character streams?

FileReader  
(character stream)  
object from the  
Input file

Reads a single  
character.

### URL for this code

<http://java.sun.com/docs/books/tutorial/essential/io/examples/CopyCharacters.java>

# Character Streams

## ◆ CopyLines.java of Line-Oriented I/O

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;
```

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.

```
public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;
        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));
            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

### URL for this code

<http://java.sun.com/docs/books/tutorial/essential/io/examples/CopyLines.java>

# Filter Streams

```
import java.io.*;
```

abstract class

```
public class UppercaseConvertor extends  
    FilterReader {  
    public UppercaseConvertor(Reader in) {  
        super(in);  
    }  
}
```

```
public int read() throws IOException {  
    int c = super.read();  
    return (c == -1 ? c :  
        Character.toUpperCase((char)c));  
}
```

```
public int read(char[] buf, int offset, int count)  
    throws IOException  
{  
    int nread = super.read(buf, offset, count);  
    int last = offset + nread;  
    for (int i = offset; i < last; i++)  
        buf[i] = Character.toUpperCase(buf[i]);  
    return nread;  
}
```

Overloaded read method: for reading in  
buffer which is array of character.

```
public static void main(String[] args)  
    throws IOException  
{  
    StringReader src = new StringReader(args[0]);  
    FilterReader f = new UppercaseConvertor(src);  
    int c;  
    while ( (c=f.read()) != -1)  
        System.out.print((char)c);  
    System.out.println();  
}
```

Function of the read() method was changed with  
filtering.

Filter streams help to chain streams to  
produce composite streams of greater utility.  
They get their power from the ability to filter-  
process what they read or write, transforming  
the data in some way.

**Run:**

% java UppercaseConvertor "no lowercase"

**Result:**

NO LOWERCASE

# Scanning and Formatting

◆ Programming I/O often involves translating to and from the neatly formatted data humans like to work with. To assist with these chores, the Java platform provides two APIs. The scanner API breaks input into individual tokens associated with chunks of data. The formatting API assembles data into nicely formatted, human-readable form.

## ◆ ScanXan.java of Scanning

```
import java.io.*;
import java.util.Scanner;
public class ScanXan {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        try {
            s = new Scanner(new BufferedReader(new
                FileReader("xanadu.txt"))));
            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

See next Slide

### Output of ScanXan:

In  
Xanadu  
did  
Kubla  
Khan  
A  
stately  
pleasure-dome  
...


### To use a different token separator: For Example

```
s.useDelimiter(",¥¥s*");
```

# Scanning and Formatting

## ◆ ScanSum.java for Parsing Individual Tokens

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;
public class ScanSum {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        double sum = 0;
        try {
            s = new Scanner(
                new BufferedReader(new FileReader("usnumbers.txt")));
            s.useLocale(Locale.US);
            while (s.hasNext()) {
                if (s.hasNextDouble()) {
                    sum += s.nextDouble();
                } else {
                    s.next();
                }
            }
        } finally {
            s.close();
        }
        System.out.println(sum);
    }
}
```



### The Scanner class:

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. This instantiation uses the Scanner (Readable source) constructor, because the BufferedReader class implements the Readable interface.

### Input of ScanSum:

8.5  
32,767  
3.14159  
1,000,000.1

### Output of ScanSum:

1032778.74159



# Scanning and Formatting

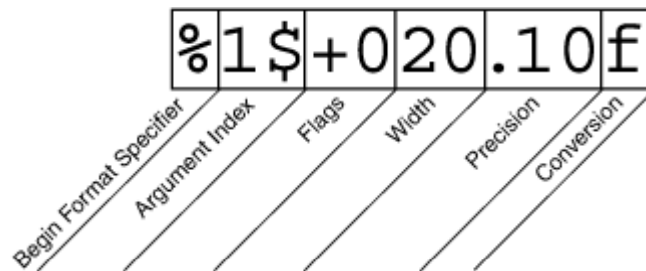
## ◆ Two Levels of formatting:

- “print” and “println” format individual values
- “format” formats

```
public class Format {  
    public static void main(String[] args)  
    {  
        System.out.format("%f, %1$+020.10f %n", Math.PI);  
        System.out.format("%f, %2$+020.10f %n", Math.PI, 2.17);  
    }  
}
```

Output:

```
3.141593, +000000003.1415926536  
3.141593, +000000002.1700000000
```



- **Precision.** For floating point values, this is the mathematical precision of the formatted value. For s and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary.
- **Width.** The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks.
- **Flags** specify additional formatting options. In the Format example, the + flag specifies that the number should always be formatted with a sign, and the 0 flag specifies that 0 is the padding character. Other flags include - (pad on the right) and , (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions.
- **The Argument Index** allows you to explicitly match a designated argument. You can also specify < to match the same argument as the previous specifier.

# I/O from the Command Line

- ◆ Standard Streams: System.in, System.out, System.err
- ◆ For character stream: InputStreamReader cin = new InputStreamReader(System.in);
- ◆ The Console has most of the features provided by the Standard Streams, and others besides. The Console is particularly useful for secure password entry.

## ◆ Example “Password.java” of the Console

```
import java.io.Console;
import java.util.Arrays;
import java.io.IOException;
public class Password {
    public static void main (String args[]) throws
IOException {
        Console c = System.console();
        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }
        String login = c.readLine("Enter your login: ");
        char [] oldPassword =
            c.readPassword("Enter your old password: ");
        if (verify(login, oldPassword)) {
            boolean noMatch;
            do {
                char [] newPassword1 =
                    c.readPassword("Enter your new password: ");
                char [] newPassword2 =
                    c.readPassword("Enter new password again: ");
```

```
noMatch = ! Arrays.equals(newPassword1, newPassword2);
                if (noMatch) {
                    c.format("Passwords don't match. Try again.%n");
                } else {
                    change(login, newPassword1);
                    c.format("Password for %s changed.%n", login);
                }
                Arrays.fill(newPassword1, ' ');
                Arrays.fill(newPassword2, ' ');
            } while (noMatch);
        }
        Arrays.fill(oldPassword, ' ');
    }
    //Dummy verify method
    static boolean verify(String login, char[] password) {
        return true;
    }
    //Dummy change method
    static void change(String login, char[] password) {}
}
```

# Data Streams

◆ Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values. All data streams implement either the `DataInput` interface or the `DataOutput` interface.

◆ Read / Write methods

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

# Data Streams

## ◆ Example “DataStreams.java”

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.BufferedReader;
import java.io.BufferedOutputStream;
import java.io.IOException;
import java.io.EOFException;
public class DataStreams {
    static final String dataFile = "invoicedata";
    static final double[] prices =
        { 19.99, 9.99, 15.99, 3.99, 4.99 };
    static final int[] units = { 12, 8, 13, 29, 50 };
    static final String[] descs = { "Java T-shirt",
        "Java Mug",
        "Duke Juggling Dolls",
        "Java Pin",
        "Java Key Chain" };
    public static void main(String[] args) throws
    IOException {
        DataOutputStream out = null;
        try {
            out = new DataOutputStream(new
                BufferedOutputStream(new
                    FileOutputStream(dataFile)));
            Open an
            output
            stream.
```

```
        }
        for (int i = 0; i < prices.length; i++) {
            out.writeDouble(prices[i]);
            out.writeInt(units[i]);
            out.writeUTF(descs[i]);
        }
        } finally { out.close(); }
        DataInputStream in = null;
        double total = 0.0;
        try {
            in = new DataInputStream(new
                BufferedInputStream(new FileInputStream(dataFile)));
            double price;
            int unit;
            String desc;
            try {
                while (true) {
                    price = in.readDouble();
                    unit = in.readInt(); desc = in.readUTF();
                    System.out.format(
                        "You ordered %d units of %s at $%.2f%n", unit, desc, price);
                    total += unit * price;
                }
            } catch (EOFException e) { }
            System.out.format("for a TOTAL of: $%.2f%n", total);
        }
        finally { in.close(); }
    }
}
```

writeUTF method  
writes String values  
in a modified form of  
UTF-8

DataStreams can  
read each record in  
the stream

# Object Streams (Object Serialization)

## ◆ What is Object Serialization?

- Serialization: process of converting an object's representation into a stream of bytes
- Deserialization: reconstituting an object from a byte stream
- Process of reading and writing objects
- Writing an object is to represent its state in a serialized form sufficient to reconstruct the object as it is read.

## ◆ Providing Object Serialization for Your Classes

- Implementing the Serializable Interface
- Customizing Serialization
- Implementing the Externalizable Interface
- Protecting Sensitive Information

# Creating a Class to be Serialized

## ◆ Implementing the Serializable interface

```
import java.io.Serializable;
import java.util.Date;
import java.util.Calendar;
public class PersistentTime implements
    Serializable
{
    private Date time;
    public PersistentTime()
    { time =
      Calendar.getInstance().getTime(); }
    public Date getTime()
    { return time; }
}
```

## ◆ How to Write to an ObjectOutputStream

```
public class FlattenTime {
    public static void main(String [] args) {
        String filename = "time.ser";
        if(args.length > 0) { filename = args[0]; }
        PersistentTime time = new PersistentTime();
        FileOutputStream fos = null;
        ObjectOutputStream out = null;
        try
        {
            fos = new FileOutputStream(filename);
            out = new ObjectOutputStream(fos);
            out.writeObject(time);
            out.close();
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

# Deserialization and Nonserializable Objects

## ◆ How to Write to an ObjectOutputStream

```
public class InflateTime {
    public static void main(String [] args)
    {
        String filename = "time.ser";
        if(args.length > 0)
        {
            filename = args[0];
        }
        PersistentTime time = null;
        FileInputStream fis = null;
        ObjectInputStream in = null;
        try {
            fis = new FileInputStream(filename);
            in = new ObjectInputStream(fis);
            time = (PersistentTime)in.readObject();
            in.close();
        } catch(IOException ex) {
            ex.printStackTrace();
        }
        // print out restored time
        System.out.println("Flattened time: " +
            time.getTime());
        System.out.println();
        // print out the current time
        System.out.println("Current time: " +
            Calendar.getInstance().getTime());
    }
}
```

## ◆ Nonserializable Objects

The object to be serialized must mark all nonserializable fields transient

```
import java.io.Serializable;
public class PersistentAnimation
implements Serializable, Runnable {
    transient private Thread animator;
    private int animationSpeed;
    public PersistentAnimation(int
        animationSpeed) {
        this.animationSpeed = animationSpeed;
        animator = new Thread(this);
        animator.start();
    }
    public void run() {
        while(true) {
            // do animation here
        }
    }
}
```

### Another Example

/home/course/javaone/sources/IO/ObjectSerial/  
ObjectFileTest.java

# File I/O

## ◆ Streams so far:

- Provide a simple model for reading and writing data
- Work with a large variety of data sources and destinations, including disk files
- However, streams don't support all the operations that are common with disk files.

## ◆ Focus on non-stream file I/O. There are two topics:

- File is a class that helps one write platform-independent code that examines and manipulates files and directories.
- Random access files support nonsequential access to disk file data.



# File I/O

## ◆ File Objects

- The **File** class makes it easier to write platform-independent code that examines and manipulates files.
- The file corresponding to the file name might not even exist. A program can use the object to parse a file name. Also, the file can be created by passing the **File** object to the constructor of some classes, such as `FileWriter`.

## ◆ A File Has Many Names

`File a = new File("xanadu.txt");`

- The program invokes a number of methods to obtain different versions of the file name. The program is then run both on a Microsoft Windows system (in directory `c:\java\examples`) and a Solaris system (in directory `/home/cafe/java/examples`).

# File I/O

## ◆ File Method Examples

```
File a = new File("xanadu.txt");
```

- a.toString()
- a.getName()
- a.getParent()
- a.getAbsolutePath()
- a.getCanonicalPath()

```
File b = new File("../" + File.separator + "examples" + File.separator  
+ "xanadu.txt");
```

- b.toString()
- b.getName()
- b.getParent()
- b.getAbsolutePath()
- b.getCanonicalPath()

# File I/O

## ◆ FileStuff.java using the File class

```
import java.io.File;
import java.io.IOException;
import static java.lang.System.out;
public class FileStuff {
    public static void main(String args[]) throws
                                IOException
    {
        out.print("File system roots: ");
        for (File root : File.listRoots()) {
            out.format("%s ", root);
        }
        out.println();
        for (String fileName : args) {
            out.format("%n-----%nnew File(%s)%n",
                                fileName);

            File f = new File(fileName);
            out.format("toString(): %s%n", f);
            out.format("exists(): %b%n", f.exists());
            out.format("lastModified(): %tc%n",
                                f.lastModified());
            out.format("isFile(): %b%n", f.isFile());
            out.format("isDirectory(): %b%n",
                                f.isDirectory());
```

'¥n' – back slash  
is OK too.

```
            out.format("isHidden(): %b%n", f.isHidden());
            out.format("canRead(): %b%n", f.canRead());
            out.format("canWrite(): %b%n", f.canWrite());
            out.format("canExecute(): %b%n",
                                f.canExecute());
            out.format("isAbsolute(): %b%n", f.isAbsolute());
            out.format("length(): %d%n", f.length());
            out.format("getName(): %s%n", f.getName());
            out.format("getPath(): %s%n", f.getPath());
            out.format("getAbsolutePath(): %s%n",
                                f.getAbsolutePath());
            out.format("getCanonicalPath(): %s%n",
                                f.getCanonicalPath());
            out.format("getParent(): %s%n", f.getParent());
            out.format("toURI: %s%n", f.toURI());
        }
    }
}
```

## ◆ Manipulating Files

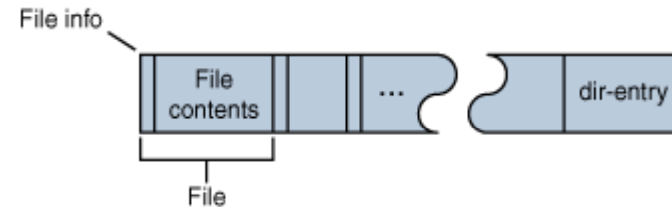
- The delete method deletes the file immediately, while the deleteOnExit method deletes the file when the virtual machine terminates.
- `new File("xanadu.txt").setLastModified(new Date().getTime());`

# Random Access Files

- ◆ Random access files permit nonsequential, or “random,” access to a file’s contents.

- ◆ Example: To extract a ZIP file

- 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.



- ◆ Methods for explicitly manipulating the file pointer in the `RandomAccessFile` class
  - `int skipBytes(int)`
  - `void seek(long)`
  - `long getFilePointer()`

# Using the RandomAccessFile

```
import java.io.*;
class FilePrint {
    public static void main (String args[]) {
        int i;
        RandomAccessFile file;
        try {
            // For writing data
            file = new RandomAccessFile(args[0], "w");
            for(i=0; i < 3; i++) {
                file.seek((long)(i * 4));
                file.writeInt(i);
            }
            file.close();
            // For reading data
            file = new RandomAccessFile(args[0], "r");
            for(i=0; i < 3; i++) {
                file.seek((long)(i * 4));
                System.out.println("data["+i+"] = " + file.readInt());
            }
            file.close();
        }
        catch (Exception e) {
            System.out.println("Error: " + e.toString());
        }
    }
}
```

Open the file  
for writing data

Put the file pointer at the  
value of the argument, and  
write the data

# Data Format of the datafile.txt (Problem 2)

The data in the file “datafile.txt” was created using the “writeLong”, “seek”, and “writeInt” methods of the “RandomAccessFile” class, and cannot be seen as their real values by text editors. Please refer to the figure below for format of the file.

