Java I/O and Files

Objectives:

- Learn the basic facts about Java's IO package
- Understand the concept of an input or output "stream"
- Learn a about exceptions in I/O
- Understand the concept of files in Java

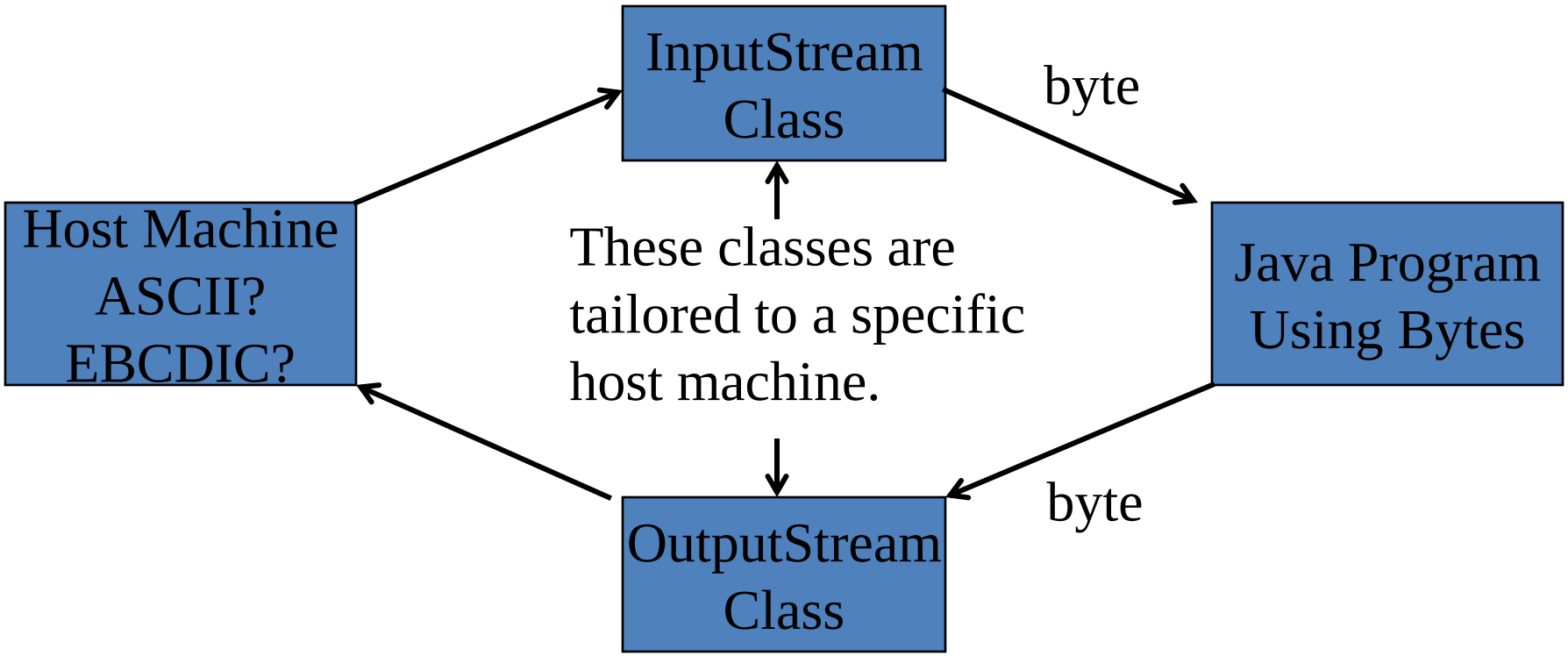
Why Is Java I/O Hard?

- Java is intended to be used on many very different machines, having
  - different character encodings (ASCII, EBCDIC, 7- 8- or 16-bit…)
  - different internal numerical representations
  - different file systems, so different filename & pathname conventions
  - different arrangements for EOL, EOF, etc.
- The Java I/O classes have to “stand between” your code and all these different machines and conventions.

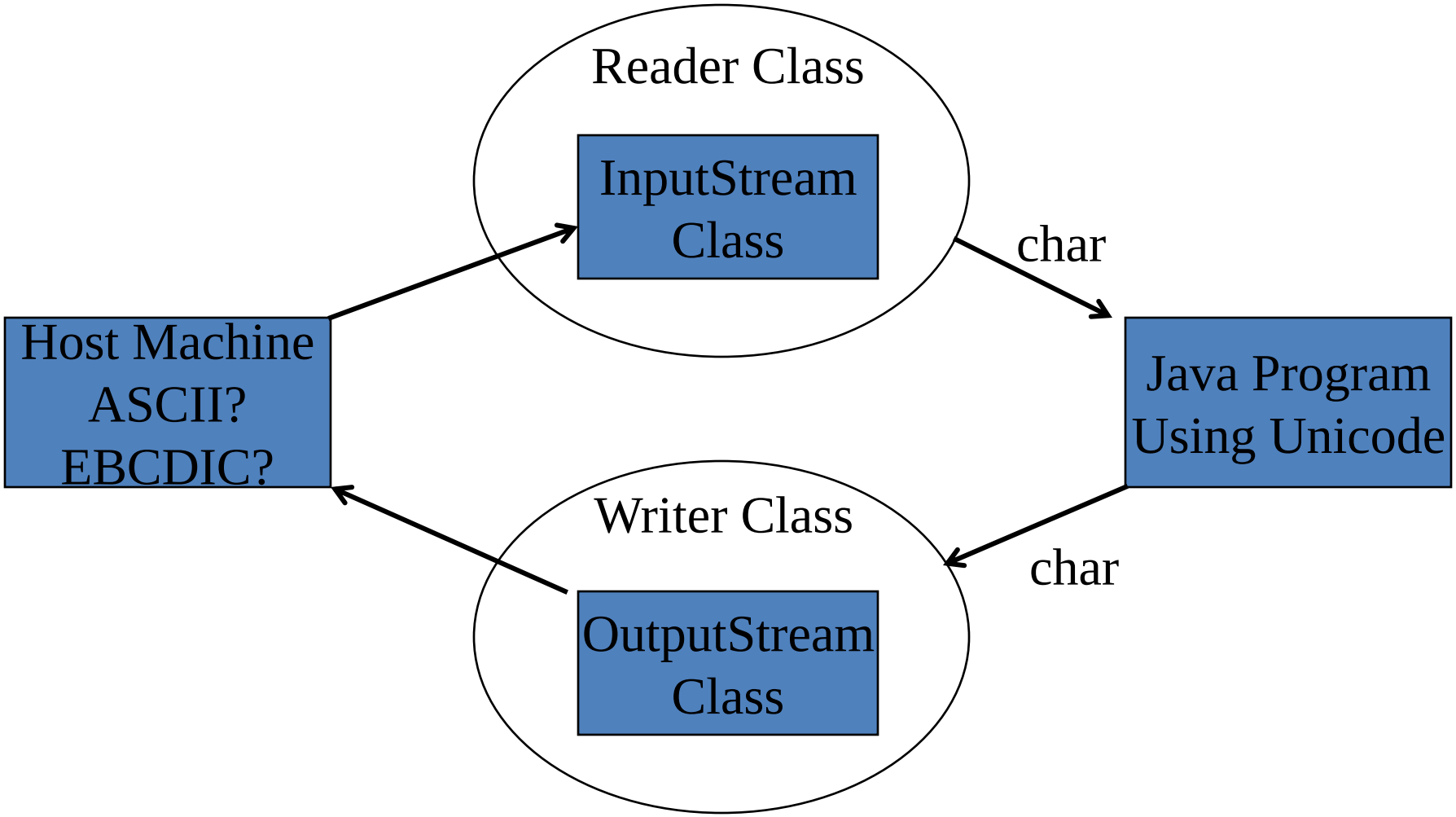
Java's Internal Characters

- Unicode. 16-bit. Good idea.
- So, the primitive type **char** is 16-bit.
- Reading from a file using 8-bit ASCII characters (for example) requires conversion.
- Same for writing.
- But binary files (e.g., graphics) are "byte-sized", so there is a primitive type **byte**.
- So Java has two systems to handle the two different requirements.
- Both are in **java.io**, so import this *always*!

Streams


InputStream Class
byte
Host Machine ASCII? EBCDIC?
These classes are tailored to a specific host machine.
Java Program Using Bytes
byte
OutputStream Class

Readers and Writers



Streams

- A “stream” is an abstraction derived from sequential input or output devices.
- An input stream produces a stream of characters; an output stream receives a stream of characters, “one at a time.”
- Streams apply not just to files, but also to actual IO devices, Internet streams, and so on.

Streams

- A file can be treated as an input or output stream.
- In reality file streams are ***buffered*** for efficiency: it is not practical to read or write one character at a time from or to mass storage.

java.io

BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter
ByteArrayInputStream
ByteArrayOutputStream
CharArrayReader
CharArrayWriter
DataInputStream
DataOutputStream
File
FileDescriptor
FileInputStream
FileOutputStream
FilePermission
FileReader
FileWriter
FilterInputStream
FilterOutputStream
FilterReader
FilterWriter
InputStream
InputStreamReader
LineNumberInputStream
LineNumberReader
ObjectInputStream
ObjectInputStream.GetField
ObjectOutputStream
ObjectOutputStream.PutField
ObjectStreamClass
ObjectStreamField
OutputStream
OutputStreamWriter
PipedInputStream
PipedOutputStream
PipedReader
PipedWriter
PrintStream
PrintWriter
PushbackInputStream
PushbackReader
RandomAccessFile
Reader
SequenceInputStream
SerializablePermission
StreamTokenizer
StringBufferInputStream
StringReader
StringWriter
Writer

java.io

- Uses four hierarchies of classes rooted at Reader, Writer, InputStream, OutputStream.
- Has a special stand-alone class RandomAccessFile.

java.io

- BufferedReader and RandomAccessFile are the only classes that have a method to read a line of text, **readLine.**
- readLine returns a String or null if the end of file has been reached.

What Are The Input Sources?

- **System.in**, which is an **InputStream** connected to your keyboard. (**System** is **public**, **static** and **final**, so it's always there).
- A file on your local machine. This is accessed through a **Reader** and/or an **InputStream**, usually using the **File** class.
- Resources on another machine through a **Socket**, which can be connected to an **InputStream**, and through it, a **Reader**.

Why Can't We Read Directly From These?

- We can, but Java provides only "low-level" methods for these types. For example, **InputStream.read()** just reads a byte…
- It is assumed that in actual use, we will "wrap" a basic input source within another class that provides more capability.
- This "wrapper" class provides the methods that we actually use.

“Wrapping”

- Input comes in through a stream (bytes), but usually we want to read characters, so “wrap” the stream in a Reader to get characters.

```
public static void main(String[] args) {
    InputStreamReader isr = new InputStreamReader(System.in);
    int c;
    try {
       while ((c = isr.read()) != -1)
       System.out.println((char) c);
    }
    catch(IOException e) {
    }
}
```

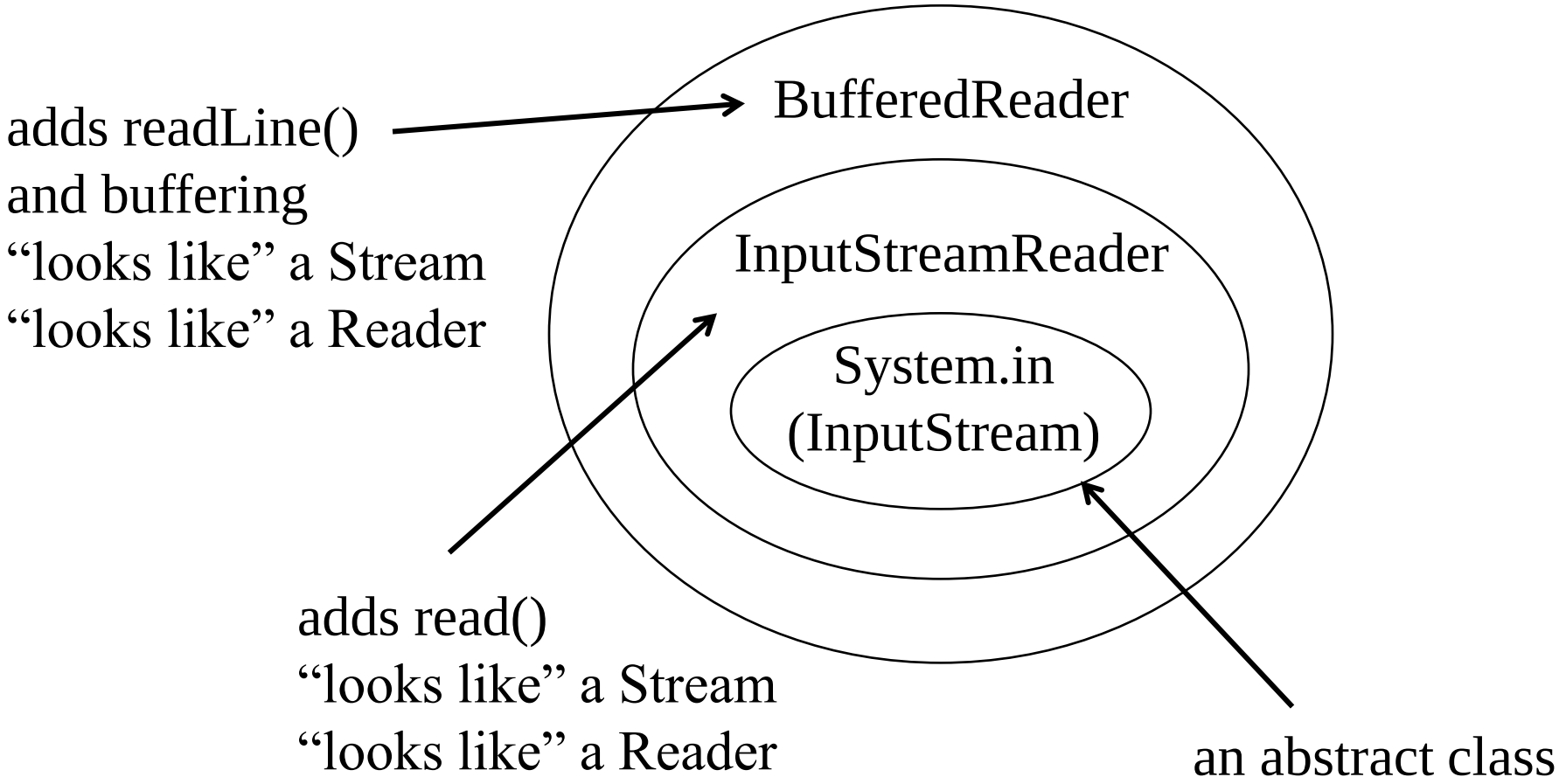
InputStreamReader

- This is a bridge between bytes and chars.
- The **read()** method returns an **int**, which must be cast to a **char**.
- **read()** returns -1 if the end of the stream has been reached.
- We need more methods to do a better job!

Use a **BufferedReader**

```
public static void main(String[] args) {
    BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
    String s;
    try {
        while ((s = br.readLine()).length() != 0)
        System.out.println(s);
    }
    catch(IOException e) {
    }
}
```

“Transparent Enclosure”



java.io

- "Throws" ***checked exceptions*** when anything goes wrong (e.g., a program fails to open a file or encounters the end of file).
- try-catch statement should be used to handle code that throws checked exceptions.
- **There are no convenient methods for reading an int or a double from an ASCII file.**

The I/O package - overview

- The `java.io` package defines I/O in terms of *streams* – ordered sequences of data that have a *source* (input streams) or a *destination* (output streams)
- Two major parts:
  1. *byte streams*
     - 8 bits, data-based
     - input streams and output streams
  2. *character streams*
     - 16 bits, text-based
     - readers and writers

Byte streams

- Two parent abstract classes: **InputStream and OutputStream**
- **Reading bytes:**
  - **InputStream** class defines an abstract method

    public abstract int read() throws IOException
    - Designer of a concrete input stream class overrides this method to provide useful functionality.
    - E.g. in the **FileInputStream** class, the method reads one byte from a file
  - InputStream class also contains nonabstract methods to read an array of bytes or skip a number of bytes

Byte streams

- Writing bytes:
  - **OutputStream** class defines an abstract method

    public abstract void write(int b) throws IOException
  - OutputStream class also contains nonabstract methods for tasks such as writing bytes from a specified byte array
- Close the stream after reading or writing to it to free up limited operating system resources by using close()

Example code1:

```
import java.io.*;
class CountBytes {

  public static void main(String[] args)
   throws IOException {
     FileInputStream in = new
                  FileInputStream(args[0]);
     int total = 0;
     while (in.read() != -1)
        total++;
     in.close();//Always close streams
    System.out.println(total + "bytes");
  }
}
```

Example code2:

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

      byte from = (byte)args[0].charAt(0);
      byte to = (byte)args[1].charAt(0);
      byte x;

      while((x = System.in.read()) != -1)
         System.out.write(x == from ? to :
  x);

   }
}
```

If you run "`java TranslateByte b B`" and enter text `bigboy` via the keyboard the output will be: `BigBoy`

Character streams

- Two parent `abstract` classes for characters: **`Reader`** and **`Writer`**.
- Each support similar methods to those of its byte stream counterpart–`InputStream` and `OutputStream`, respectively
- The standard streams—`System.in`, `System.out` and `System.err`—existed before the invention of character streams. So they are byte streams though logically they should be character streams.

Stream Objects

All Java programs make use of standard stream objects

- System.in
  - To input bytes from keyboard
- System.out
  - To allow output to the screen
- System.err
  - To allow error messages to be sent to screen

Conversion between byte and character streams

- The conversion streams `InputStreamReader` and `OutputStreamReader` translate between character and byte streams

  - **`public InputStreamReader(InputStream in)`**
  - **`public OutputStreamWriter(OutputStream out)`**

- `read` method of `InputStreamReader`
  - read bytes from their associated `InputStream` and convert them to characters
- `write` method of `OutputStreamWriter`
  - take the supplied characters, convert them to bytes and write them to its associated `OutputStream`

Reading Characters

```
Import java.io.*;
class  Reading{
  public static void main(String a[])throws IOException
  {
       char c;
       BufferedReader br = new BufferedReader(new
    InputStreamReader(System.in))
      do{
      c=(char)br.read();
      System.out.println(c);
      } while(c!='q');
  }
}
```

Files

- A file is a collection of data in mass storage.
- A data file is not a part of a program's source code.
- The same file can be read or modified by different programs.
- The program must be aware of the format of the data in the file.

Files (cont'd)

- The file system is maintained by the operating system.
- The system provides commands and/or GUI utilities for viewing file directories and for copying, moving, renaming, and deleting files.
- The system also provides "core" functions, callable from programs, for reading and writing directories and files.

Text Files

- A computer user distinguishes text ("ASCII") files and "binary" files. This distinction is based on how you treat the file.
- A text file is assumed to contain lines of text (e.g., in ASCII code).
- Each line terminates with a "newline" character (or a combination, carriage return plus line feed).

Text Files

- Examples:
  - Any plain-text file, typically named *something*.txt
  - Source code of programs in any language (e.g., *Something*.java)
  - HTML documents
  - Data files for certain programs, (e.g., fish.dat; any file is a data file for some program.)

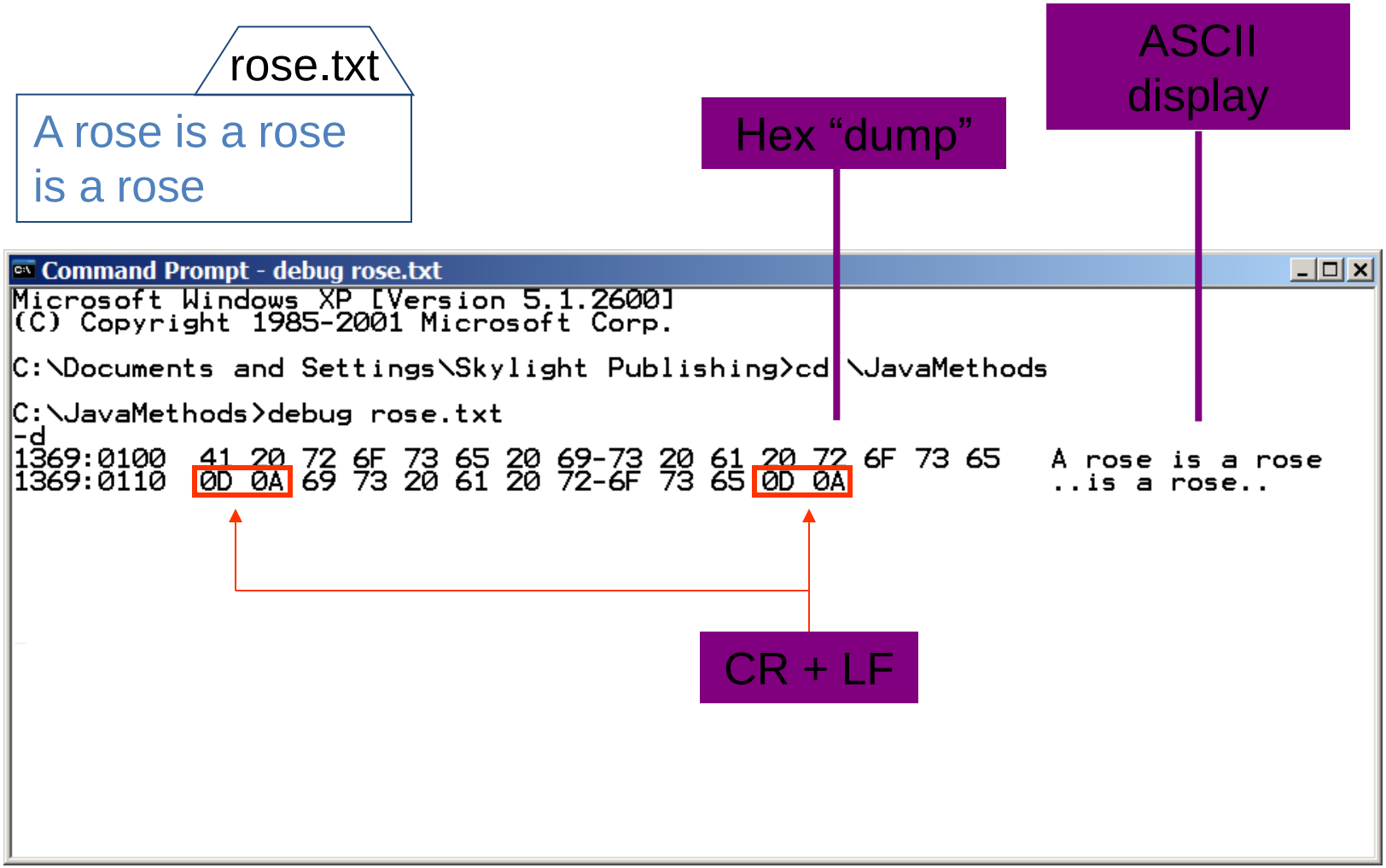
Binary Files

- A “binary” file contains any information, any combination of bytes.
- Only a programmer / designer knows how to interpret it.
- Different programs may interpret the same file differently (e.g., one program displays an image, another extracts an encrypted message).

Binary Files

- Examples:
  - Compiled programs (e.g., *Something*.class)
  - Image files (e.g., *something*.gif)
  - Music files (e.g., *something*.mp3)
- Any file can be treated as a binary file (even a text file, if we forget about the special meaning of CR-LF).

Text as Binary:



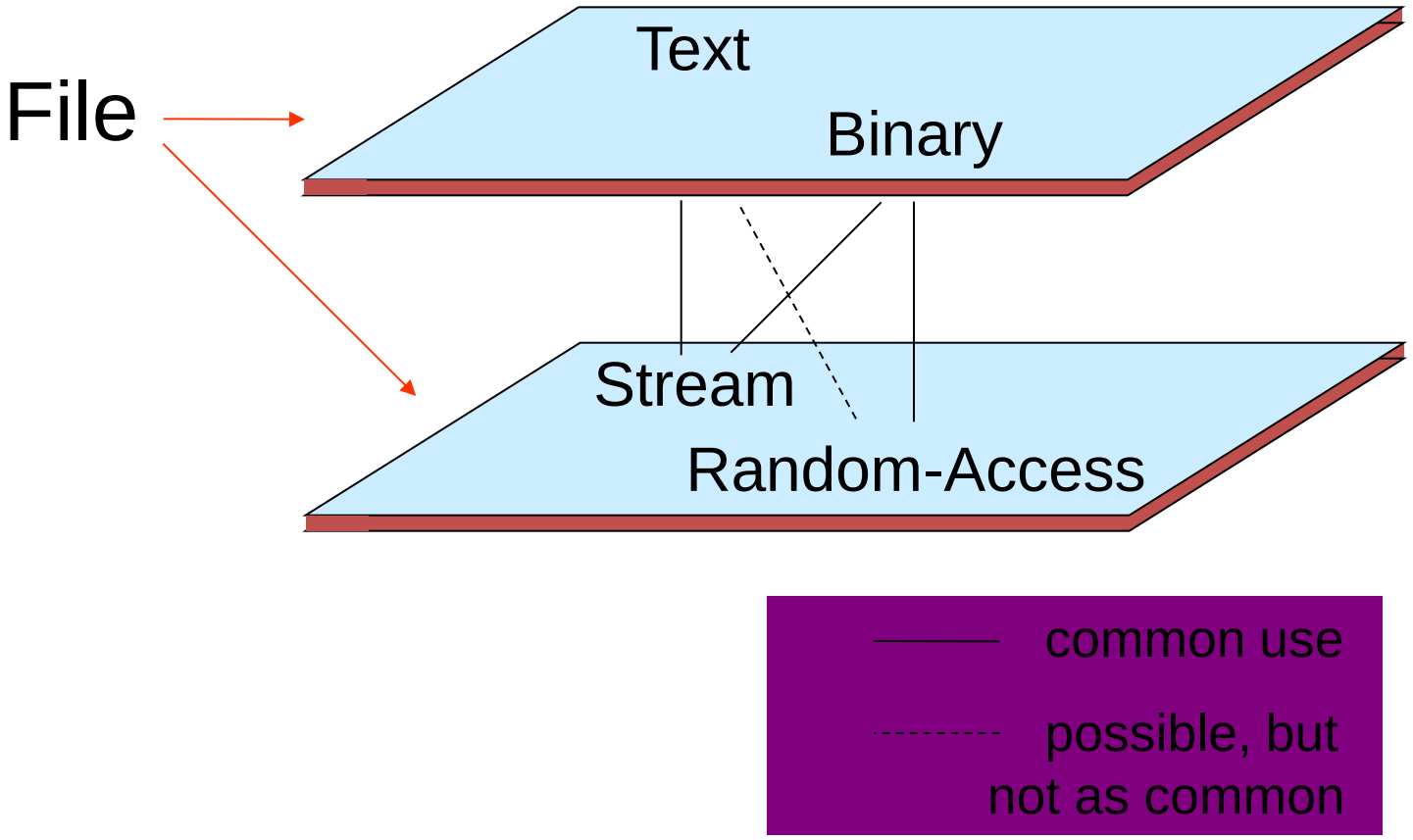
Random-Access Files

- A program can start reading or writing a random-access file at any place and read or write any number of bytes at a time.
- “Random-access file” is an abstraction: any file can be treated as a random-access file.
- You can open a random-access file both for reading and writing at the same time.

Random-Access Files (cont'd)

- A binary file containing fixed-length data records is suitable for random-access treatment.
- A random-access file may be accompanied by an "index" (either in the same or a different file), which tells the address of each record.
- Tape : CD == Stream : Random-access

File Types: Summary



Some Classes for File Handling

- **`FileInputStream`** and **`FileOutputStream`** perform file input and output respectively
- **`FileReader`** and **`FileWriter`**
  - are used to read and write characters to a file
- **`DataInputStream`** and **`DataOutputStream`**
  - allow a program to read and write binary data using an **`InputStream`** and **`OutputStream`** respectively
- **`ObjectInputStream`** and **`ObjectOutputStream`**
  - deal with Objects implementing **`ObjectInput`** and **`ObjectOutput`** interfaces respectively

Reading From a File: **FileInputStream**

- Its constructor takes a string containing the file pathname.

```
public static void main(String[] args) throws IOException {
   InputStreamReader isr = new
      InputStreamReader(new FileInputStream("FileInput.java"));
   int c;
   while ((c = isr.read()) != -1)
      System.out.println((char) c);
   isr.close();
}
```

Reading From a File (cont.)

- Here we check for a -1, indicating we've reached the end of the file.
- This works just fine if the file to be read is in the same directory as the class file, but an absolute path name is safer.
- The **read()** method can throw an **IOException**, and the **FileInputStream** constructor can throw a **FileNotFoundException**
- Instead of using a try-catch construction, this example shows main() declaring that it throws **IOException**.

The **File** Class

- Think of this as holding a file *name*, or a list of file *names* (as in a directory).
- You create one by giving the constructor a pathname, as in

  File f = new File("d:/www/java/week10/DirList/.");

- This is a directory, so now the **File f** holds a list of (the names of) files in the directory.
- It's straightforward to print them out.

Listing Files

```
import java.io.*;
import java.util.*;
public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        System.out.println(path.getAbsolutePath());
        list = path.list();
        for (int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
}
```

java.io

- Uses “wrapper” classes (a.k.a “decorators”): a “more advanced” object is constructed around a simpler object, adding features.

```
import java.io.*;
...
  BufferedReader inputFile = new
BufferedReader (
                        new  FileReader (inFileName));

  PrintWriter outputFile = new PrintWriter (
              new BufferedWriter (
                  new FileWriter (outFileName)));
```

Working with files

- Sequential-Access file: the `File` streams—`FileInputStream`, `FileOutputStream`, `FileReader` and `FileWriter`—allow you to treat a file as a stream to input or output sequentially
  - Each file stream type has three types of constructors
    - A constructor that takes a `String` which is the name of the file
    - A constructor that take a `File` object which refers to the file
    - A constructor that takes a `FileDescriptor` object

Working with files

- Random-Access file: `RandomAccessFile` allow you to read/write data beginning at the a specified location
  - a *file pointer* is used to guide the starting position
  - It's not a subclass of `InputStream`, `OutputStream`, `Reader` or `Writer` because it supports both input and output with both bytes and characters

Example of RandomAccessFile

```
import java.io.*;
class Filecopy {
   public static void main(String args[]) {
      RandomAccessFile fh1 = null;
      RandomAccessFile fh2 = null;
      long filesize = -1;
      byte[] buffer1;

      try {
         fh1 = new RandomAccessFile(args[0],
                                    "r");
         fh2 = new RandomAccessFile(args[1],
                                    "rw");

      } catch (FileNotFoundException e) {
         System.out.println("File not found");
         System.exit(100);
      }
```

Example of RandomAccessFile (Continued)

```
 try {
    filesize = fh1.length();
    int bufsize = (int)filesize/2;
    buffer1 = new byte[bufsize];
    fh1.readFully(buffer1, 0, bufsize);

    fh2.write(buffer1, 0, bufsize);

  } catch (IOException e) {
     System.out.println("IO error
                              occurred!");
     System.exit(200);
     }
   }
}
```

Important Point

Data must be read in in the same form that it is written out to a file

Writing

```
output = new ObjectOutputStream
          ( new FileOutputStream(filename ));
output.writeObject( objectname );
output.close( );
```

Reading

```
input = new ObjectInputStream
           new FileInputStream( filename ) );
record = ( ObjectType ) input.readObject( );
input.close( );
```

The `File` class

- The `File` class is particularly useful for retrieving information about a file or a directory from a disk.
  - A `File` object actually represents a path, not necessarily an underlying file
  - A `File` object doesn't open files or provide any file-processing capabilities
- Three constructors
  - `public File( String name)`
  - `public File( String pathToName, String name)`
  - `public File( File directory, String name)`

Methods in the `File` class

- `boolean canRead() / boolean canWrite()`
- `boolean exists()`
- `boolean isFile() / boolean isDirectory() / boolean isAbsolute()`
- `String getAbsolutePath() / String getPath()`
- `String getParent()`
- `String getName()`
- `long length()`
- `long lastModified()`