Java I/O and Files

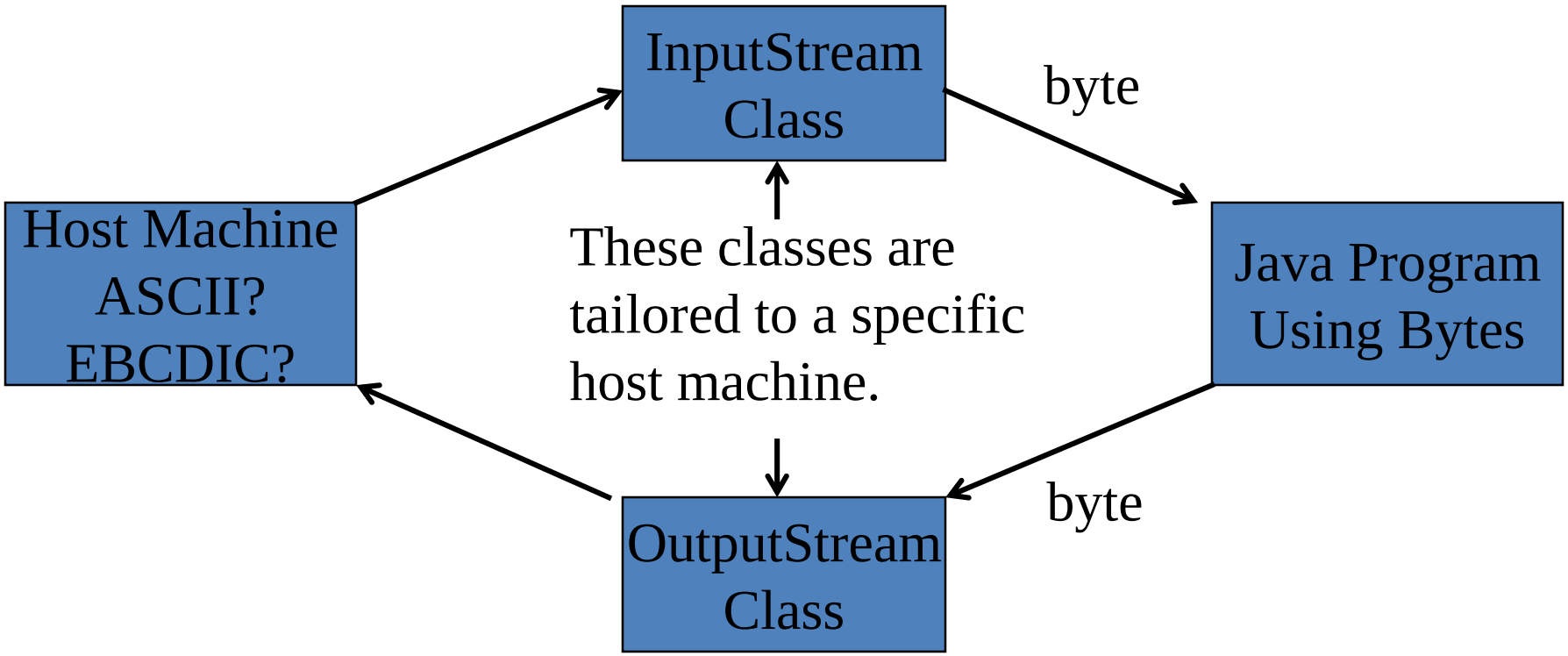
Why Is Java I/O Hard?

- 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.

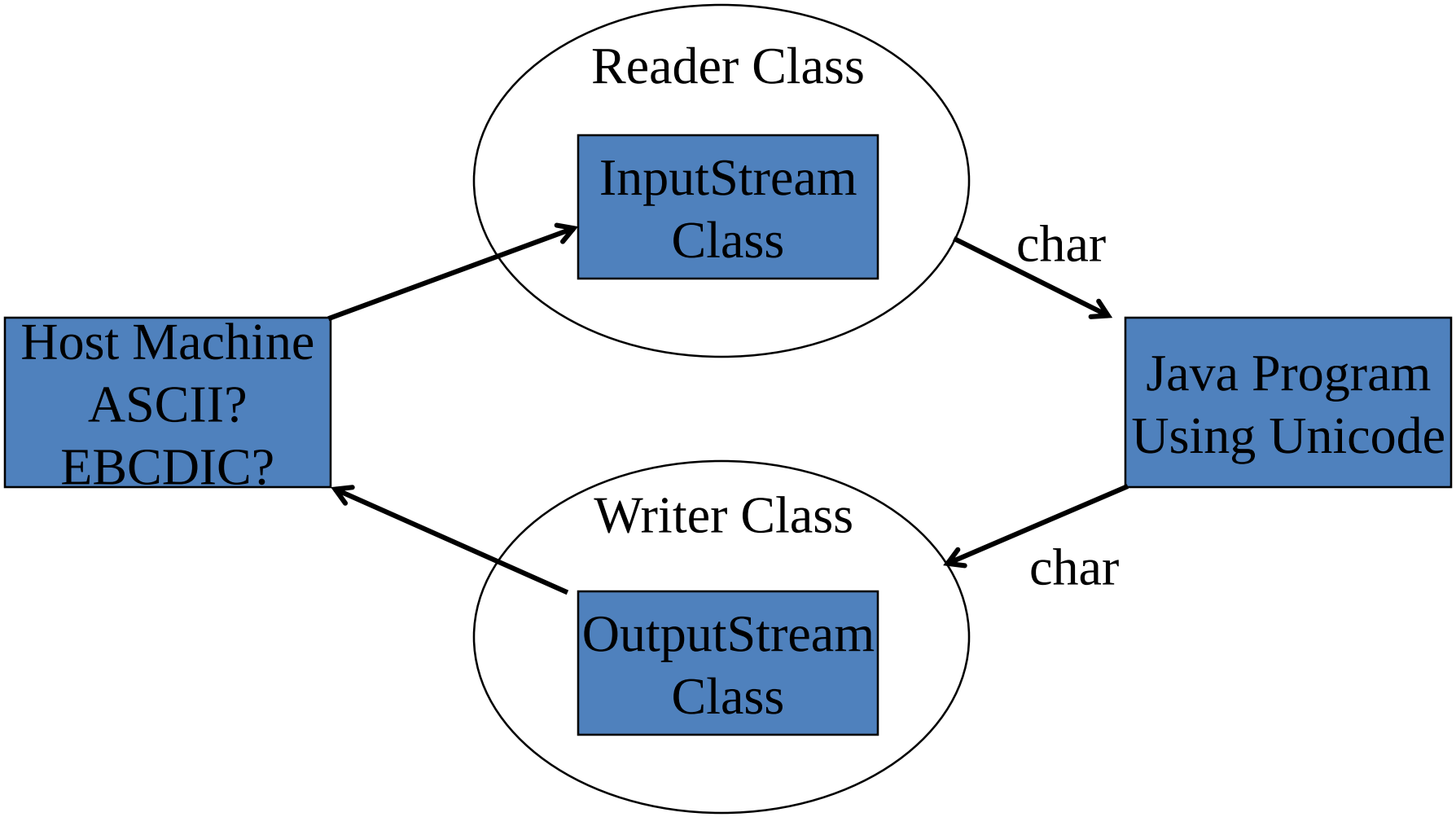
Java's Internal Characters

- Unicode. 16-bit. Good idea.
- primitive type **char** is 16-bit.
- Reading /writing from a file using 8-bit ASCII characters (for example) requires conversion.
- 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.
- Streams apply not just to files, but also to actual IO devices, Internet streams, and so on.

- In reality streams are ***buffered*** : it is not practical to read or write one character at a time.

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.

“Wrapping”

- Input comes in through a stream (bytes)
- 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) {
    }
}
```

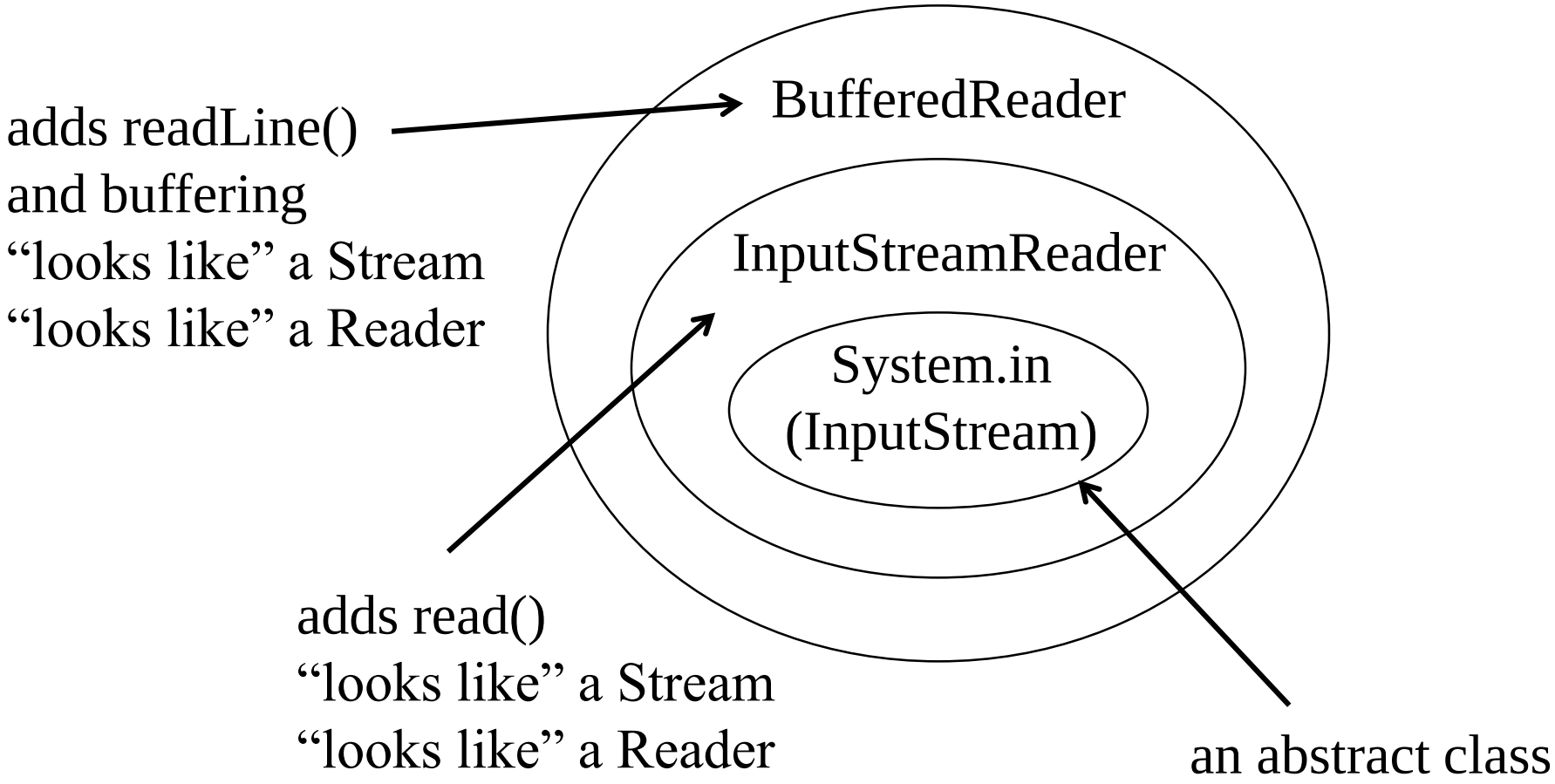
InputStreamReader

- 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*** try-catch statement should be used to handle code that throws checked exceptions.

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
  - **OutputStream** class defines an abstract method

    public abstract void write(int b) throws IOException

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`**.
- 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

- **`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

- 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.

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.)

- -1: indicates end of the file.
- absolute path name is safer.
- The **read()** method can throw an **IOException**, and the **FileInputStream** constructor can throw a **FileNotFoundException**

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]);
    }
}
```

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()`