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

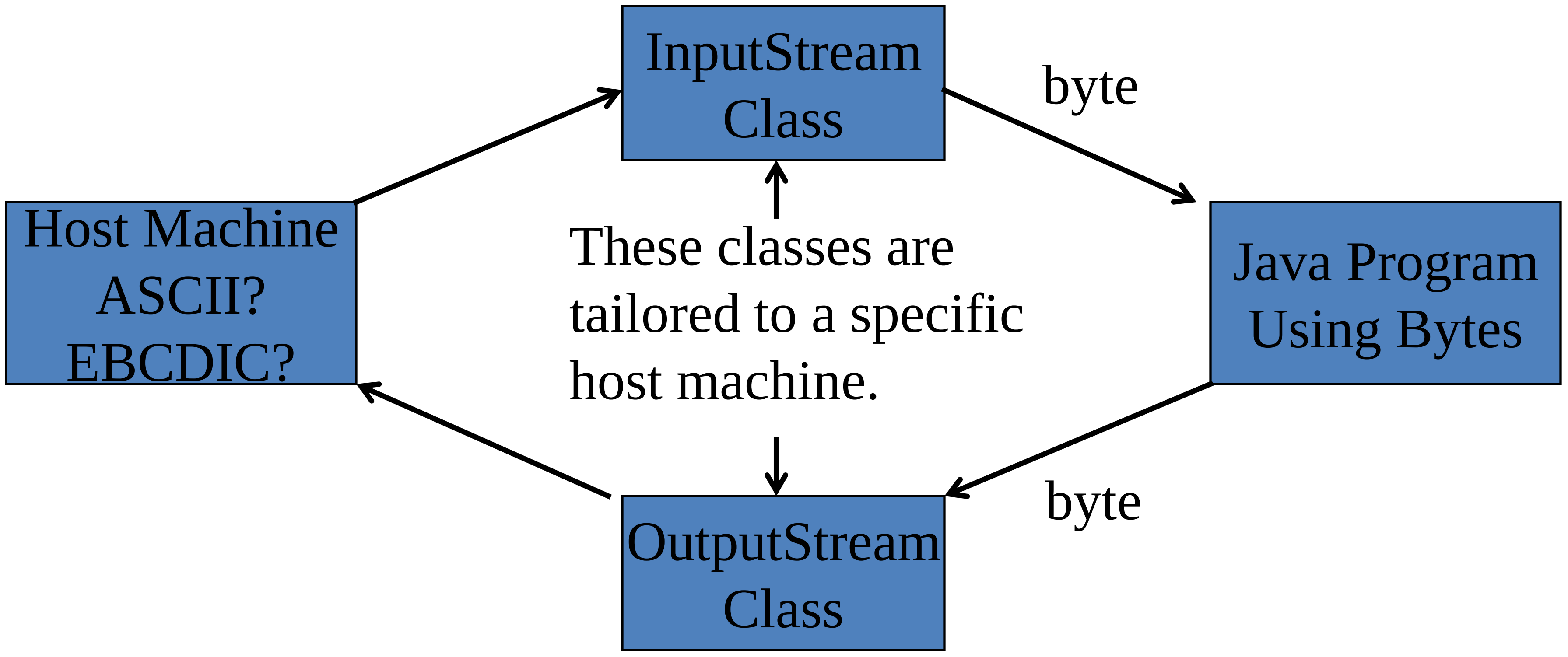
Why Is Java I/O Hard?

- 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.

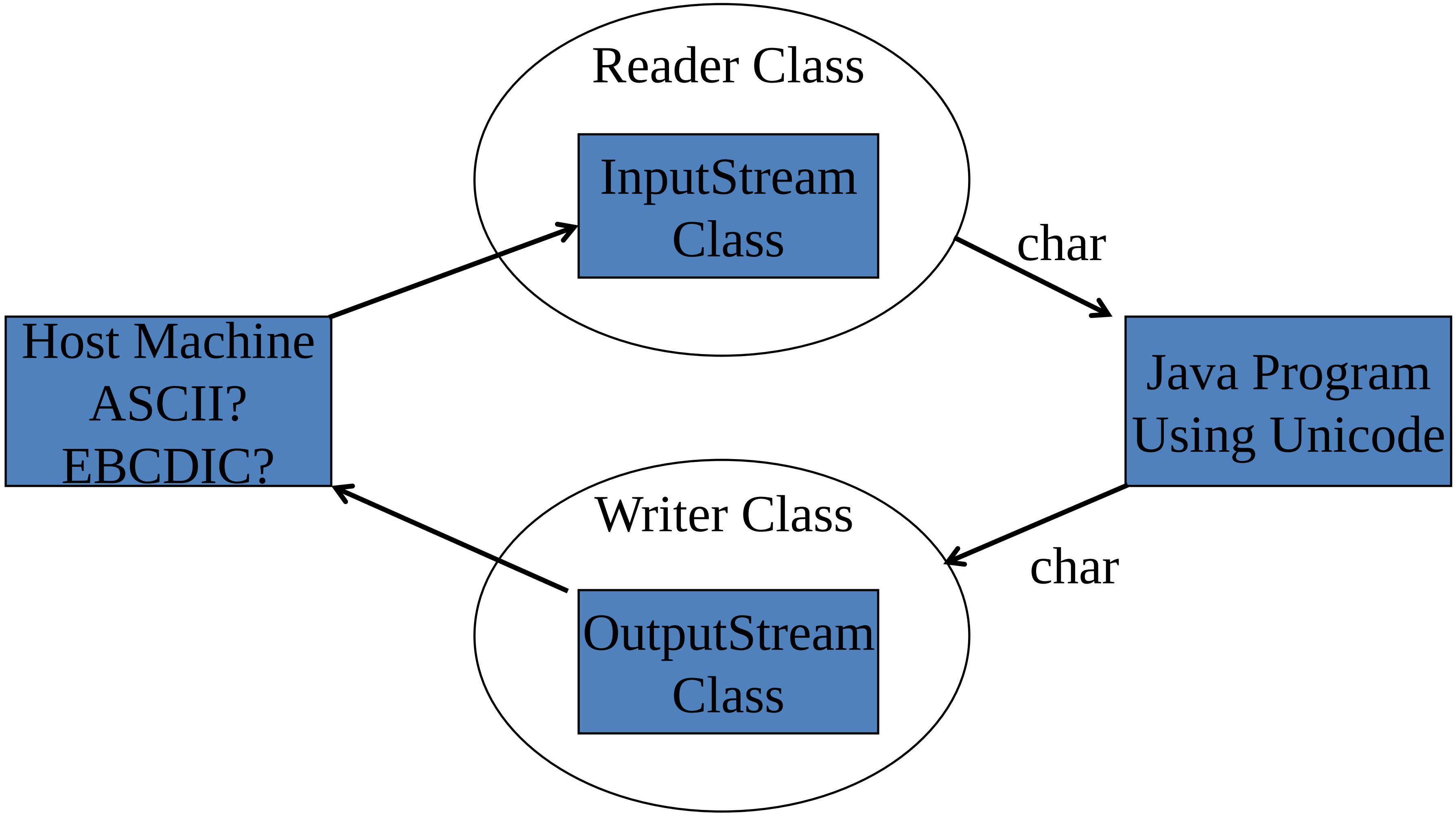
Java’s Internal Characters

- 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) {
    }
}
```

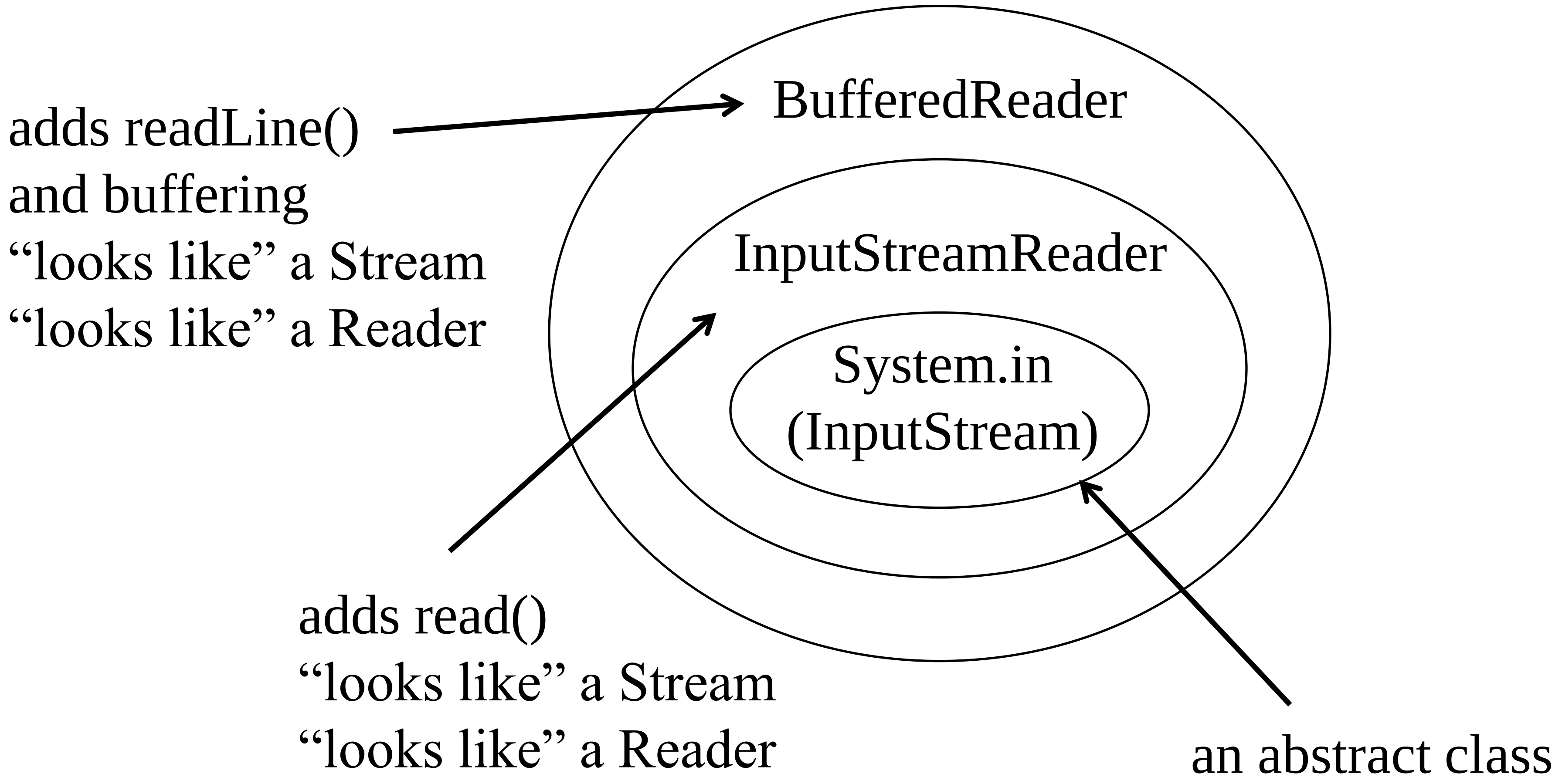
InputStreamReader

- 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()

<u>Example code1:</u>

```
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”);
   }
}
```

<u>Example code2:</u>

```
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');
  }
}
```