



# Java's Input/Output

---

Java's Hierarchy Input/Output Classes and  
Their Purpose

# Introduction

---

- These slides introduce several input and output classes for special purposes, such as reading or writing a file.
- Some of these slides use the concept of **inheritance**.

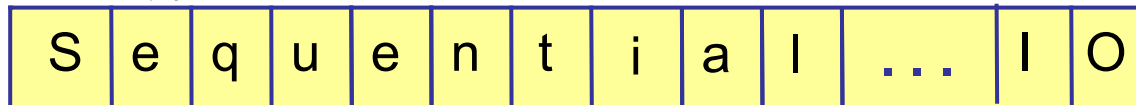
.

# Types of Input/Output

	Data in text format (data as characters)	Data in binary format
Sequential Access	<ul style="list-style-type: none"><li>■ write to terminal</li><li>■ text, html files</li><li>■ printf( ), format( )</li></ul>	<ul style="list-style-type: none"><li>■ not human readable</li><li>■ efficient for space and computer read</li><li>■ image, MP3, Word</li></ul>
Random Access		

# Sequential Access

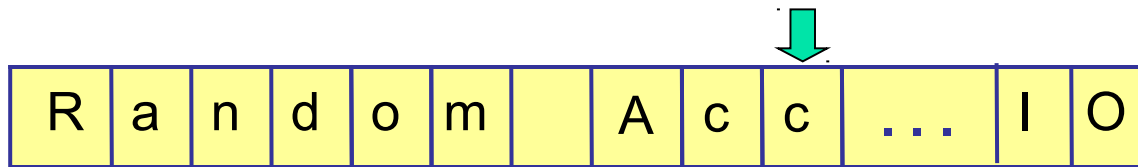
- Read/write everything starting from the beginning.
- Sequential:
  - Cannot "back up" and reread or rewrite something.
  - Cannot "jump" to arbitrary location in stream.
- InputStream and OutputStream use sequential I/O.
- InputStream has a **skip(n)**, but it is still sequential.



```
int a = instream.read( ); // read a = 'S'  
byte [ ] b = new byte[10];  
int count = instream.read( b ); // read next 10 bytes
```

# Random Access

- Can move to any location using **seek ( )** method.
- Can move forward and backward.
- Only makes sense for files.



```
File file = new File( "c:/data/myfile.txt" );  
RandomAccessFile rand =  
    new RandomAccessFile(file, "r");  
rand.seek( 9L ); // goto byte 9  
int b = rand.read( );
```

# Basic Input Classes

---

`InputStream`

read input as bytes

`Reader`

read input as characters

`InputStreamReader`

`BufferedReader`

read Strings, read entire lines

# InputStream

Reads input as bytes -- one byte at a time.

If no more data, read() returns -1.

```
buffer = new StringBuffer( );  
while ( true ) {  
    int c = inputStream.read( );  
    if ( c < 0 ) break; // end of input  
    buffer.append( (char)c );  
}
```

# Do & test programming Idiom

This kind of code is common in C.

```
buffer = new StringBuffer( );  
int c = 0;  
while ( (c=inputStream.read( )) >=0 ) {  
    buffer.append( (char)c );  
}
```



# InputStream with array of byte

It can be more efficient to read many bytes at one time.

```
byte [] buff = new byte[80];  
while ( true ) {  
    int count = inputStream.read( buff );  
    if ( count <= 0 ) break; // end  
    // process buff  
}
```

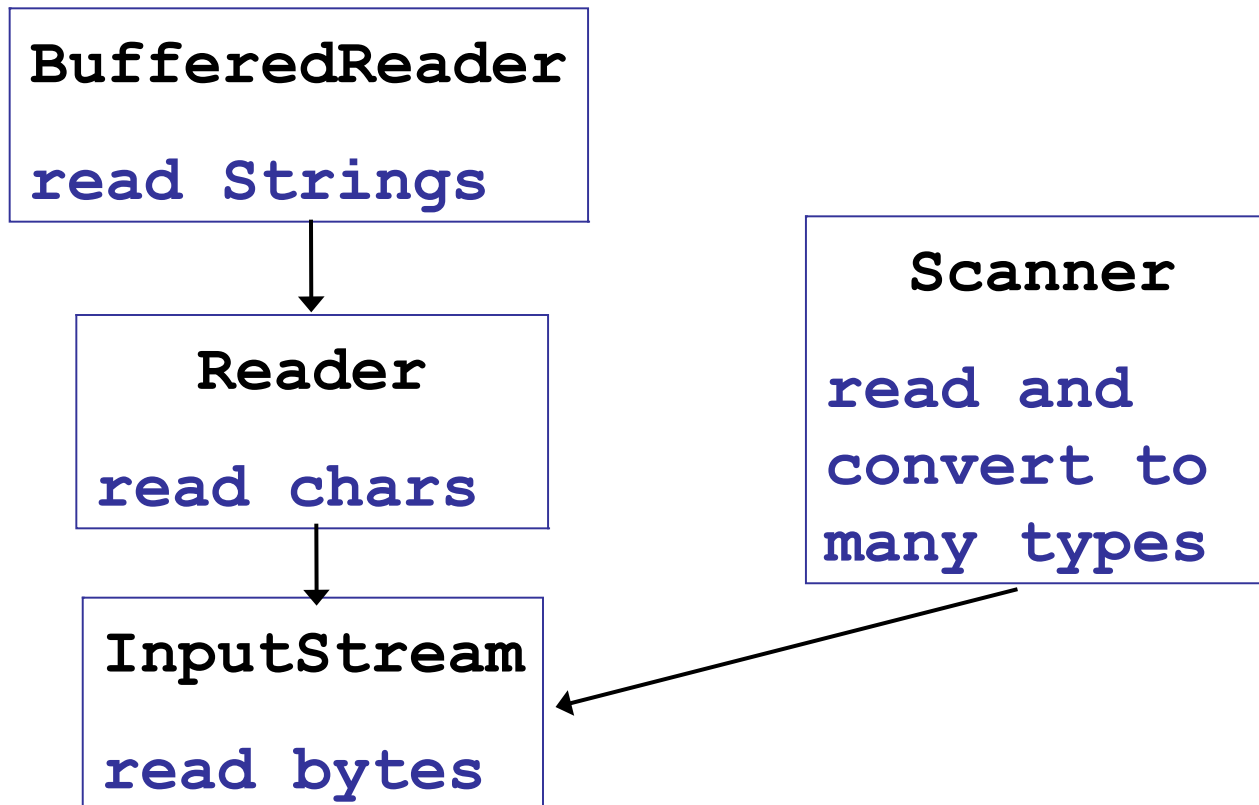
# FileInputStream

- An InputStream connected to a file.
- Has many constructors.
- Works just like InputStream!

```
FileInputStream inputStream =  
    new FileInputStream("c:/test.dat");  
  
while ( true ) {  
    int c = inputStream.read( );  
    if ( c < 0 ) break; // end of input  
    buffer.append( (char)c );  
}  
inputStream.close( );
```

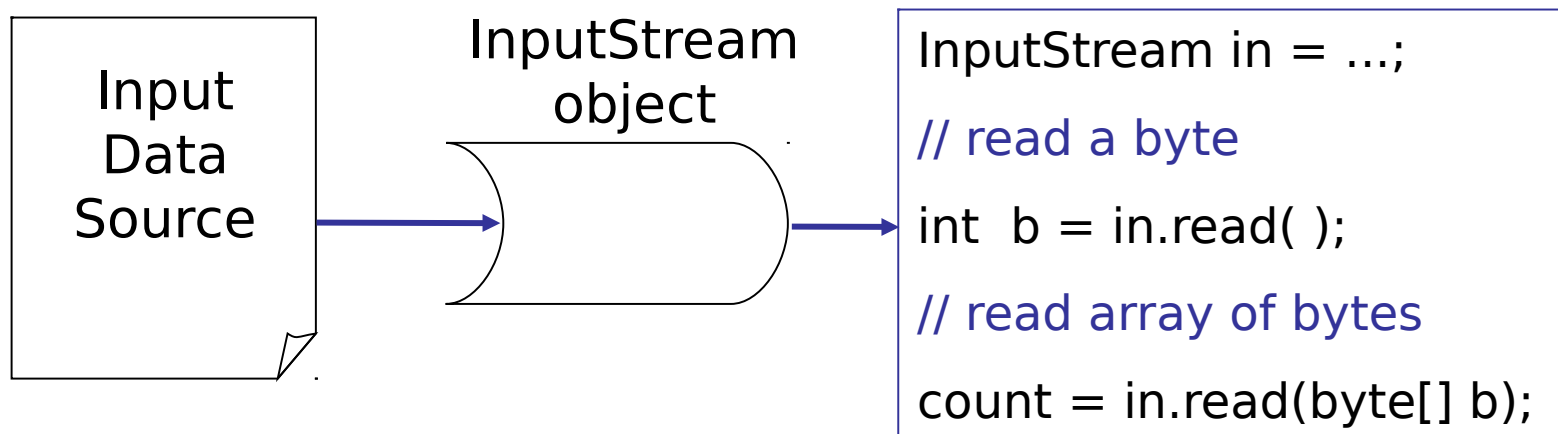
# Input Classes Hierarchy

- Each layer "adapts" a lower layer to provide a different interface. They are **adaptors**.



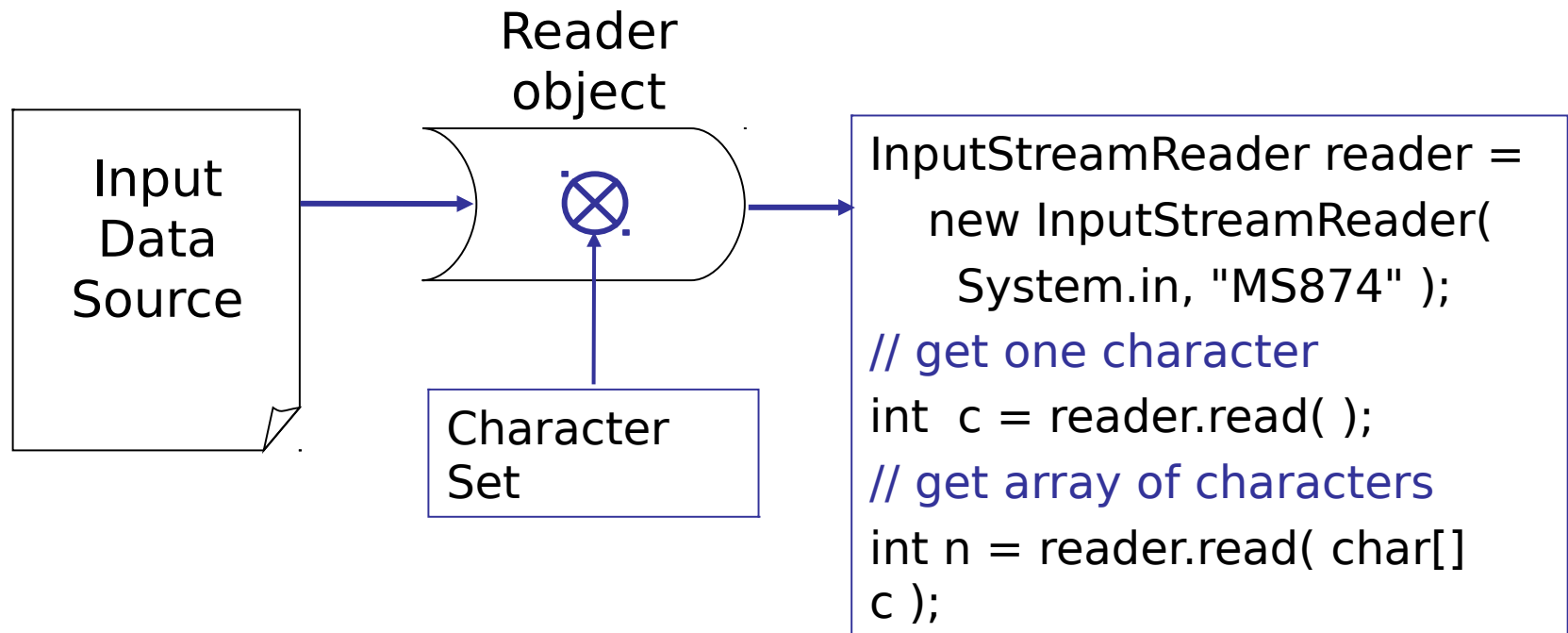
# InputStream

- InputStream reads bytes and returns them.
- **No interpretation** of character sets.
- OK for raw data.
- **Not good** for character data using character set.



# Reader

- Reader: reads bytes and converts to chars.
- Interprets bytes according to **Character Set Encoding**.
- Can handle any language... if you know the charset.



# InputStreamReader class

InputStreamReader is a kind of Reader.

It converts bytes into characters.

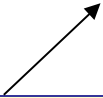
```
InputStream in = new FileInputStream( "test" );
InputStreamReader reader =
    new InputStreamReader(in);
// read a character
char b = (char) reader.read( );
// read several characters
char [ ] buff = new char[100];
int nchars = reader.read( buff, 0, 100);
// close the input stream
reader.close( );
```

# Character Sets

Java API docs list names of character sets.

InputStreamReader reader

= new InputStreamReader( inputStream, "charset" );



Charset Name	Description
ASCII	ASCII 7-bit encoding
ISO8859_1	ISO 8859-1 Latin alphabet No. 1
Cp874	IBM (DOS) Thai encoding
MS874	MS Windows Thai encoding
TIS620	TIS-620, Thai encoding
<b>UTF-8</b>	<b>8-bit UTF encoding</b>
UTF-16	16-bit UTF encoding

# BufferedReader class

BufferedReader reads input as Strings.

It uses a Reader to read characters

```
BufferedReader breader = new BufferedReader(  
    new InputStreamReader( System.in ) );  
// read a line  
String s = breader.readLine( );
```

Buffered Reader methods:

int read( ) - read next char

int read(char[ ], start, count) - read chars into array

String readLine( ) - return a string containing rest of the line

close( ) - close the reader



# BufferedReader for File Input

To read from a file, create a `BufferedReader` around a `FileReader`. The `ready()` method returns true if (a) input buffer contains data (e.g. reading from `System.in` or a pipe) or (b) underlying data source is not empty (reading from file).

```
String filename = "mydata.txt";
BufferedReader bin = new BufferedReader(
    new FileReader( filename ) );
// read some lines
while( bin.ready( ) )
{
    String s = bin.readLine( );
    // do something with the string
}
bin.close( );
```

# Input Streams and Readers

Java has a Reader class corresponding to common InputStream classes.

## **InputStream**

InputStream

LineNumberInputStream

FilterInputStream

FileInputStream

PipedInputStream

## **Reader**

InputStreamReader

LineNumberReader

FilterReader

FileReader

PipedReader

## **Reading Binary Data**

DataInputStream

use readChar() method  
of DataInputStream to  
interpret data as characters

# InputStream Hierarchy

Java provides a *hierarchy* of classes for processing input from different sources and types.

## Java Input Stream Class Hierarachy

InputStream

    ByteArrayInputStream

    FileInputStream

    PipedInputStream

    ObjectInputStream

    SequenceInputStream

    FilterInputStream

        DataInputStream (binary input)

        BufferedInputStream

        LineNumberInputStream

        PushbackInputStream

These are  
"wrappers"  
for another  
input stream.

# How to Read without Blocking

InputStream has an `available( )` method that returns the number of bytes waiting to be read.

Use this to read without blocking.

Reader classes have a `ready( )` method.

```
InputStream in = System.in;  // or whatever

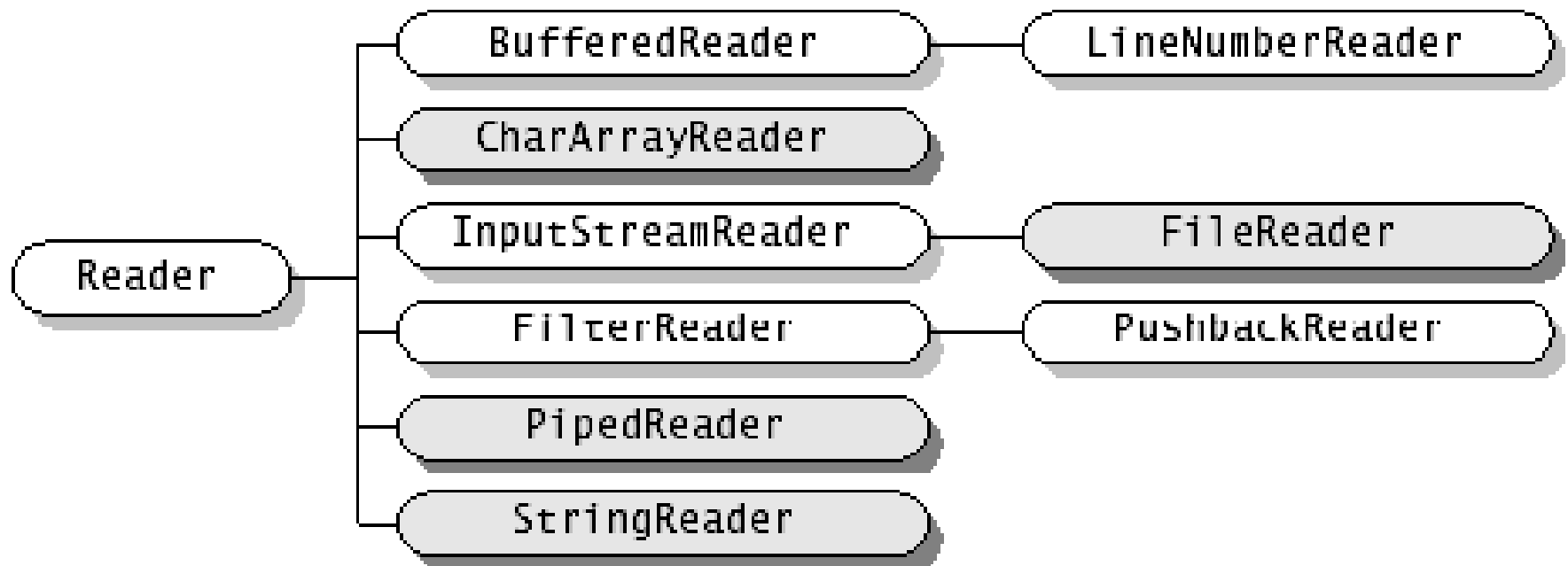
// read whatever bytes are available
int size = in.available();
if ( size > 0 ) {
    byte [ ] b = new byte[size];
    in.read( b ); // this should not block
}
```

# BufferedReader and End-of-Data

The `readLine( )` method returns `null` if the end of input stream is encountered. You can use this to read all data from a file.

```
String filename = "mydata.txt";
BufferedReader bin = new BufferedReader(
    new FileReader( filename ) );
// read all data
String s;
while( ( s = bin.readLine() ) != null )
{
    // process data in String s
    System.out.println( s.toUpperCase() );
}
file.close( );
```

# Reader Class Hierarchy



# Reading Binary Data

Examples:

- MP3 file, image file

Advantages:

- space efficient, can read quickly (little conversion)

```
InputStream instr = new FileInputStream( "mydata" );
DataInputStream data = new DataInputStream( instr );

try {
    int n = data.readInt( );           // 4 bytes
    double x = data.readDouble( );    // 8 bytes
    char c = data.readChar( );         // 2 bytes
} catch ( IOException e ) { ... }
```

# End-of-File for DataInputStream

- Throws EOFException if end of input encountered while reading.

```
InputStream fin = new FileInputStream( "mydata" );
DataInputStream data = new DataInputStream( fin );

double sum = 0;
while( true ) {
    try {
        double x = data.readDouble( ); // 8 bytes
        sum += x;
    } catch ( IOException e ) { ... }
    catch ( EOFException e ) { break; } // EOF
}
data.close( );
```



# Scanner

`java.util.Scanner` is newer than the other classes.

`Scanner` "wraps" an `InputStream` or a `String` and provides parsing and data conversion.

```
// scanner wraps an InputStream
InputStream in = new FileInputStream(...);
Scanner scanner = new Scanner( in );

// scanner to parse a String
String s = "Peanuts 10.0 Baht";
Scanner scan = new Scanner( s );
```

# Reading with Scanner

Can test for presence of data.

Convert next token into any primitive or get entire line as String.

```
Scanner scanner = new Scanner("3 dogs .5");  
if ( scanner.hasNextInt() )  
    n = scanner.nextInt();  
if ( scanner.hasNext() )  
    word = scanner.next();  
if ( scanner.hasNextDouble() )  
    x = scanner.nextDouble();  
// read and discard rest of this line  
scanner.nextLine();
```

# Parsing with Scanner

Can change separator character.

Can search using regular expressions.

```
Scanner scanner = new Scanner("aa,bb,999");  
scanner.useDelimiter(",");  
String word = scanner.next(); // = "aa"  
String w = scanner.findInLine("\\d\\d\\d");  
// w is "999"  
\\d is a regular expression for a digit 0-9
```

# Output Classes

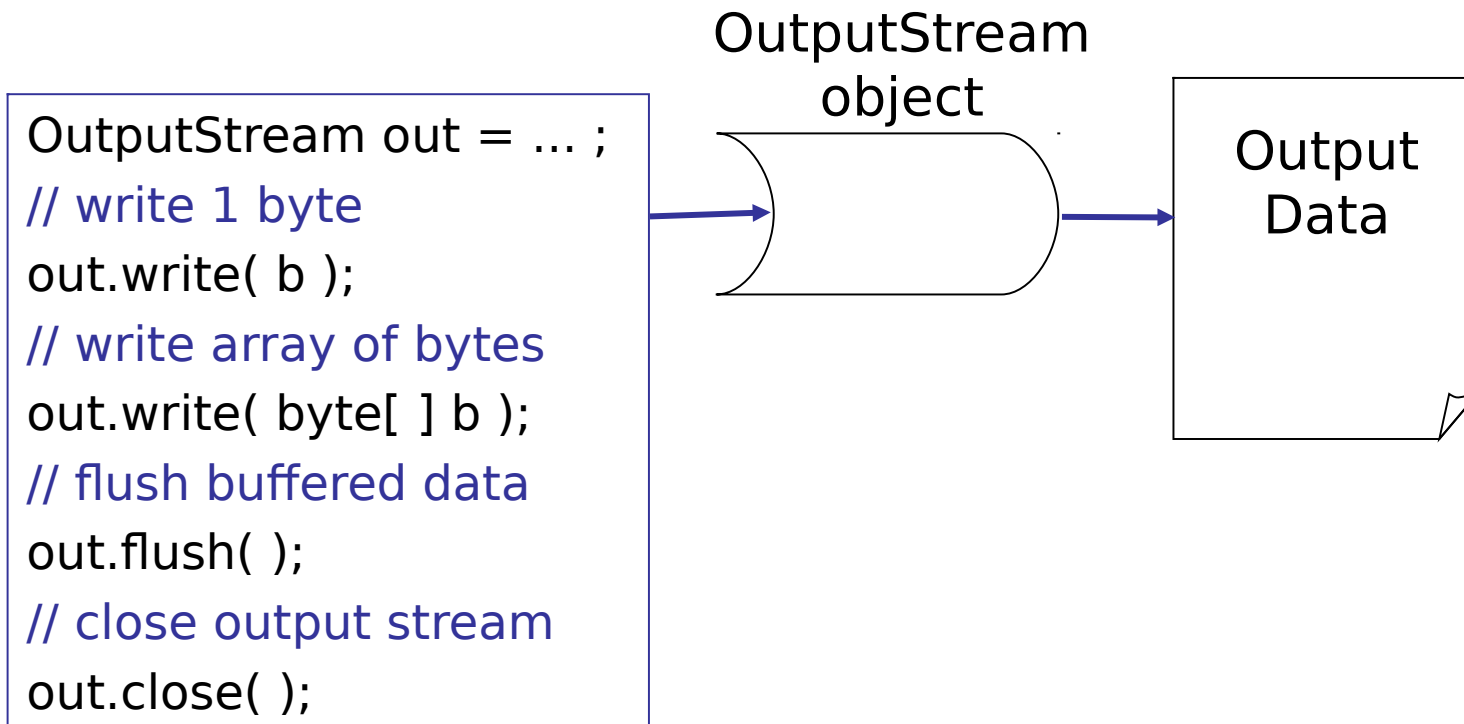
Three layers, just like Input hierarchy

- **OutputStream**: outputs bytes (low level)
- **Writer**: outputs characters (convert to bytes)
- **BufferedWriter**: outputs strings and lines. buffers data

**Formatter**: utility for creating formatted output. Can be used as a pre-filter for an output stream or output to any *Appendable* object.

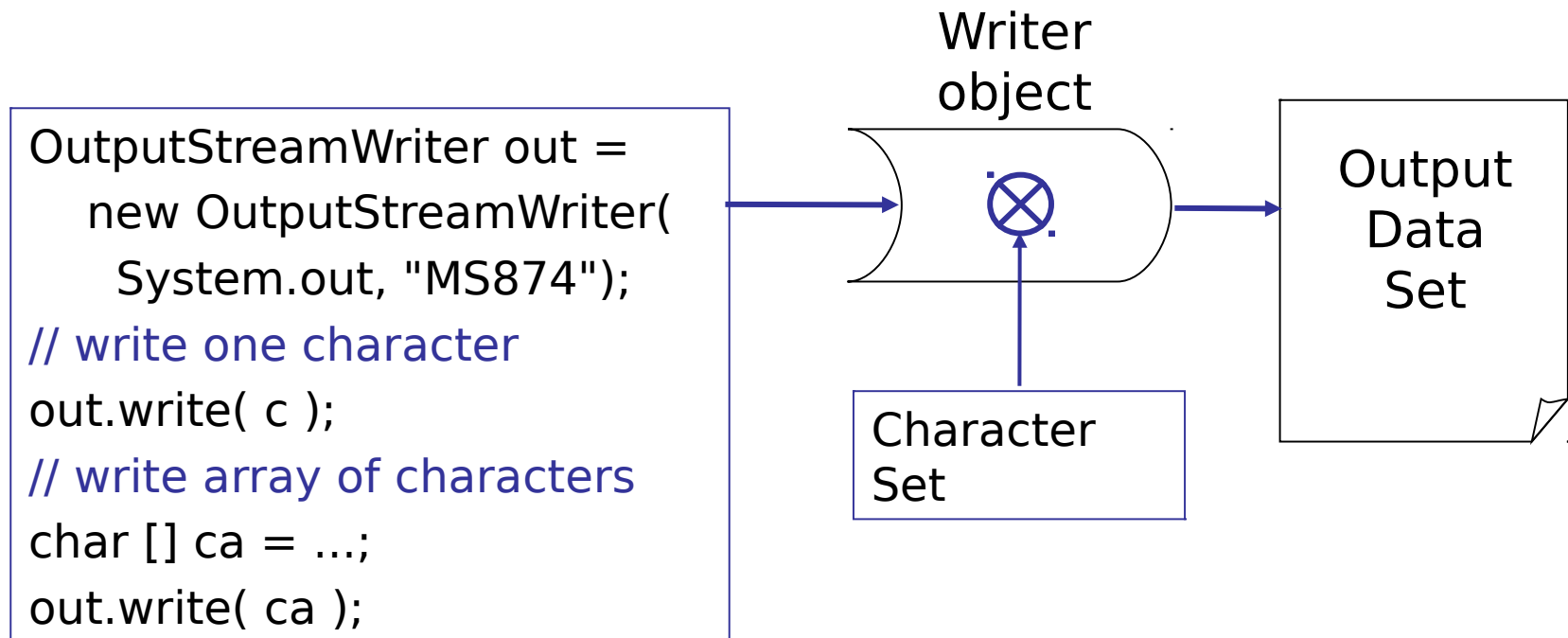
# OutputStream

- OutputStream writes bytes to some output sink.
- No interpretation of character sets.
- Works OK for text in system's default character set.



# Writer

- Writer converts UNICODE characters to bytes.
- Interprets chars according to character set encoding.
- Can handle any language (if you know the charset).



# Output Streams and Writers

Java has several classes derived from `OutputStream` and `Writer`. Each class handles a particular output sink.

## **OutputStream**

`OutputStream`

`FilterOutputStream`

`FileOutputStream`

`PipedOutputStream`

## **Writer**

`OutputStreamWriter`

`FilterWriter`

`FileReader`

`PipedWriter`

`StringWriter`

## **Writing Binary Data**

`DataOutputStream`

use `writeChar()` or  
`writeChars()` methods to  
output UNICODE characters

# Handling Exceptions

The Java input and output methods will throw an `IOException` if there is an error in any input/output operation such `read()`, `write()`, or `print()`. Your program must deal with this exception in one of two ways:

1. Throw the exception..

```
public void myMethod throws IOException( ) {  
    // read and process the input  
}
```

2. Catch the exception and take some action. This is illustrated on the next slide.



# Catching an Exception

```
BufferedReader myfile;
try {
    myfile = new BufferedReader(
        new FileReader( filename ) );
} catch (IOException e) {
    System.out.println(
        "Couldn't open file" + filename);
    return;
}
// read a line from file
try {
    String s = myfile.readLine( );
    // do something with string
} catch (IOException e) {
    System.out.println("Exception " + e
        + " while reading file.");
}
```

# Using Files

The `FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter` classes operate on `File` objects.

Create a `File` object by specifying the filename (and optional path):

```
File file1 = new File("input.txt"); // in "current" directory
File file2 = new File("/temp/input.txt"); // in temp dir
File file3 = new File("\\temp\\input.txt"); // same thing
File file4 = new File("/temp", "input.txt"); // same thing
File dir = new File("/temp"); // open directory as file
```

These commands **do not create a file** in the computer's **file system**. They only create a `File` object in Java.

# Testing Files

The File class has methods to:

- test file existence and permissions
- create a file, delete a file
- get file properties, such as path

```
File file = new File( "/temp/input.txt" ); // file object
```

```
if ( file.exists( ) && file.canRead( ) ) // OK to read  
    FileInputStream fin = new FileInputStream(file);
```

```
if ( ! file.exists( ) ) file.createNewFile( ); // create a file!  
if ( file.canWrite( ) ) // OK to write  
    FileOutputStream fout = new FileOutputStream(file);
```

# More File Operations

File objects can tell you their size, location (path), modification time, etc. See the Java API for File.

```
File file = new File("/temp/something.txt"); // file object
```

```
if ( file.isFile() ) {  
    /* this is an ordinary file */  
    long length = file.length( );  
    long date = file.lastModified( );  
}
```

```
if ( file.isDirectory() ) {  
    /* this is a directory */  
    File files [] = file.listFiles(); // read directory  
}
```

# File Copy Example

Copy a file. Realistically, you should test file existence and permissions, catch IOException, etc.

```
File infile = new File("/temp/old.txt");
File outfile = new File("/temp/new.txt");
if ( outfile.exists( ) ) outfile.delete( );
outfile.createNewFile( );

FileReader fin = new FileReader( infile );
FileWriter fout = new FileWriter( outfile );
// reading char at a time is very inefficient
int c;
while ( (c = fin.read()) >= 0 ) fout.write(c);
fin.close();
fout.flush();
fout.close();
```

# Pipes

Reading and writing pipes: one method writes data into the pipe, another method reads data from the pipe.

Very useful for multi-threaded applications.

```
PipedOutputStream pout = new PipedOutputStream();  
PipedInputStream pin = new PipedInputStream(pout);
```



```
PipedOutputStream pout = new PipedOutputStream();  
PipedInputStream pin = new PipedInputStream( pout );  
PrintStream out = new PrintStream( pout );  
BufferedInputStream in = new BufferedInputStream( pin );  
out.println("data into the pipe"); // write to the pipe  
String s = in.readLine(); // read from the pipe
```

# RandomAccessFile

- Random Access I/O means you can move around in the file, reading/writing at any place you want.
- For output, you can even write *beyond* the end of file.
- Use **seek ( )** to move current position.

```
RandomAccessFile ra = new RandomAccessFile("name", "rw");  
ra.seek( 100000L ); // go to byte #100000  
  
byte [ ] b = new byte[1000];  
  
// all "read" methods are binary, like DataInputStream  
ra.readFully( b ); // read 1000 bytes  
  
ra.seek( 200000L ); // go to byte #200000  
  
ra.write( b );
```

# More Information

---

In the **Sun Java Tutorials** (online)

**I/O: Reading and Writing**

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

**Handling Errors with Exceptions**

<http://java.sun.com/docs/books/tutorial/essential/exceptions/>