



Input & Output Classes

Java's Hierarchy of Input/Output Classes
and Their Purpose

Introduction

- Java has a hierarchy of input classes for different purposes.
- The base class is `InputStream`, which reads input as bytes. It has many subclasses, like `FileInputStream` and `DataInputStream`.
- Other input classes convert the input into characters, strings, or primitive data types.
 - Those classes (almost) all get the bytes they need from `InputStream`.
- Java's output classes use a similar design.

.

Sources of Input

An application can read **input** from:

- **console** or terminal
- **file** on disk, USB drive, etc.
- **URL** on the internet
- a **sensor**, using its device driver
- a **microphone** or **camera**, using its device driver
- **another process** (output of another application)
- other sources

What's in a File?

A file contains **bytes**, grouped into blocks (usually 512 bytes per block).

Only bytes -- 0's and 1's.

Not text, characters, or images.

Only **Bytes**.

What about Text Files? PNG?

Text, images, sound, etc. depend on how the application interprets the **bytes** in a file.

JPEG, PNG, GIF - a standard for how to read and write **bytes** that represent an image.

MP3 - a standard for how to interpret **bytes** so they can be rendered (by software) as sound.

TXT - **bytes** represent **codes** for characters, using a standard character encoding. Depending on the encoding, one character may be more than one byte in a file.

Text File Example

Suppose we want to write this to a file:

```
cat
```

What data is actually written to the file?

If we use ASCII or Unicode UTF-8 *encoding* for the characters, the file would contain:

binary

```
0110001101100001
0111011100001010
```

hexadecimal

```
6361740A
```

Hex 63 is the code for 'c'. The last byte (hex 0A) is a *newline*. It is added by a text editor or Java's `System.out.println()`.

Each **hex digit** represents 4 bits, e.g. 3=0011, 6=0110, A=1010,

Text File Example Explained

1. "encode" each character according to a **character set** and **encoding**. For ASCII, UTF-8, and many others:
 - 'c' = x63 (hexadecimal 63, or binary 0110 0001)
 - 'a' = x61
 - 't' = x74
2. at the end of each "line" of text is a **line-end character**. This is usually newline (hex 0A):
 - '\n' = x0AOld MacOS apps use a "carriage return" (hex 0D) instead of newline.

Numerical Example

How can we save the value of Math.PI to a file?

3.141592653589793

If we write it as *text (characters)* the result would be
hexadecimal view of file (spaces added for clarity):

```
34312e33 32393531 35333536 39373938  
00000a33
```

20 bytes. But this is a waste!

Math.PI is a **double** value using 8 bytes in memory.

If we write the actual binary value (as in memory) to file, it would only need 8-bytes, and no data lost in converting to decimal digits.

Writing Binary Values

Java has a `DataOutputStream` class for writing primitive values (byte, int, long, double) directly to a file in binary form without converting to text form.

Writing `Math.PI` to a file in binary form, the file contains **hexadecimal view of file:**

```
0940 fb21 4454 182d
```

Only 8 bytes! (2 hexadecimal digits = 1 byte)

MP3, JPEG, PNG, MPEG all contain data in **binary** format, as in the example above. It is faster and smaller than writing numeric data as text.

The Important Lesson

Data on computers is read and written as **bytes**.

"Text" is an interpretation of those bytes as characters.

"Text" requires use of a *character set* and *character encoding* to translate chars to bytes or bytes to chars.

"Binary format" refers to a file where the data is stored directly in a binary encoding, without converting to text.

Classes for Input

`InputStream`

read input as **bytes**

`Reader,`
`InputStreamReader`

read input as **characters**

`BufferedReader`

read entire lines as **Strings**

`DataInputStream`

read data as **primitive** values
(byte, int, float, double)

InputStream

Reads input as bytes. `read()` returns 1 byte.

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 );
}
```

InputStream with array of byte

Reading 1 byte at a time is *slow*.

It is usually faster to read many bytes into an array.

```
byte[] buff = new byte[1024];
while ( true ) {
    int count = inputStream.read( buff );
    if ( count <= 0 ) break; // end
    // process buff[0] ... buff[count-1]
}
```

Why byte[1024]?

You can make the byte array any size you like.

However, at a lower level the operating system reads and "buffers" input in "blocks".

1 block is usually 512, 1024, 2048, or 4096 bytes.

So, its efficient to use code that matches this.

```
int BUFF_SIZE = 512; // or 1024, 2048, 4196, or 8*1024
byte[] buff = new byte[BUFF_SIZE];
int count = in.read(buff);
```

Do & test programming Idiom

This kind of code is common in C.

It calls `read()`, sets the value of `c`, then tests the result.

When there is no more input, `read()` returns -1.

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

InputStream with array error

read(byte[]) may **not** fill the **entire** array (buff)!

Always check **count** of bytes read.

Do not assume that `count == buff.length` !

```
byte[] buff = new byte[512];  
count = inputStream.read(buff);  
  
// ERROR! buff may contain junk  
String data = new String(buff);
```


read() returns actual # bytes read

```
int count = in.read( buff );
```

count is actual number of bytes read.

Do not assume that count == buff.length !

```
byte[] buff = new byte[512];  
count = inputStream.read(buff);  
  
// Correct: only use byte[0] ... [count]  
String data = new String(buff, 0, count);  
// or  
out.write(buff, 0, count);
```

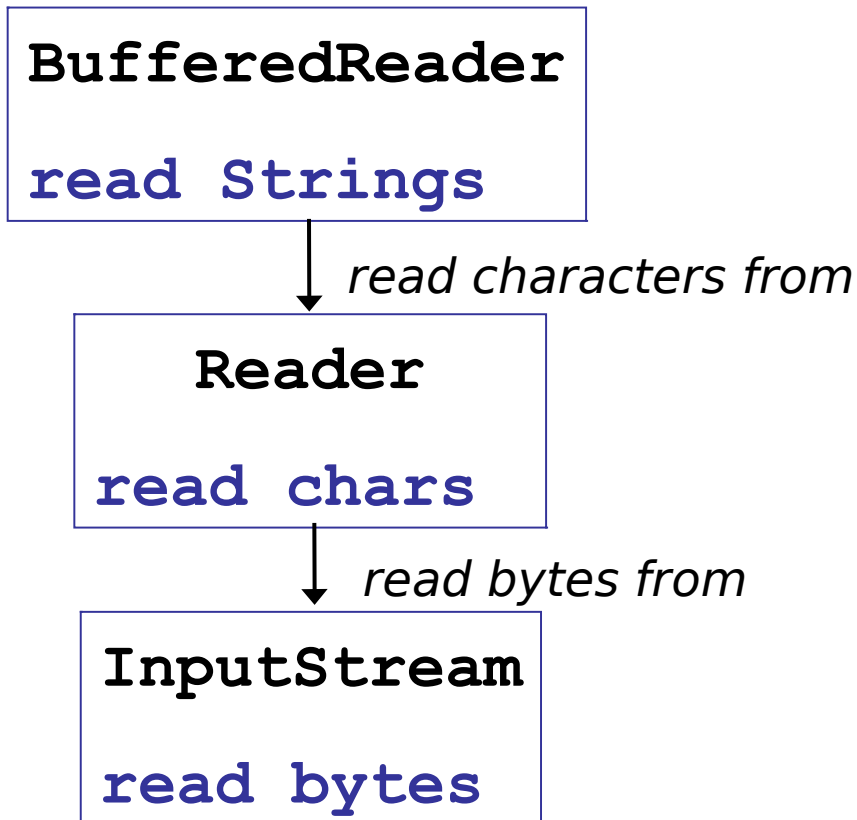
FileInputStream

- An InputStream connected to a file.
- Has many constructors.
- Works just like InputStream! (its a subclass)

```
InputStream 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( );
```

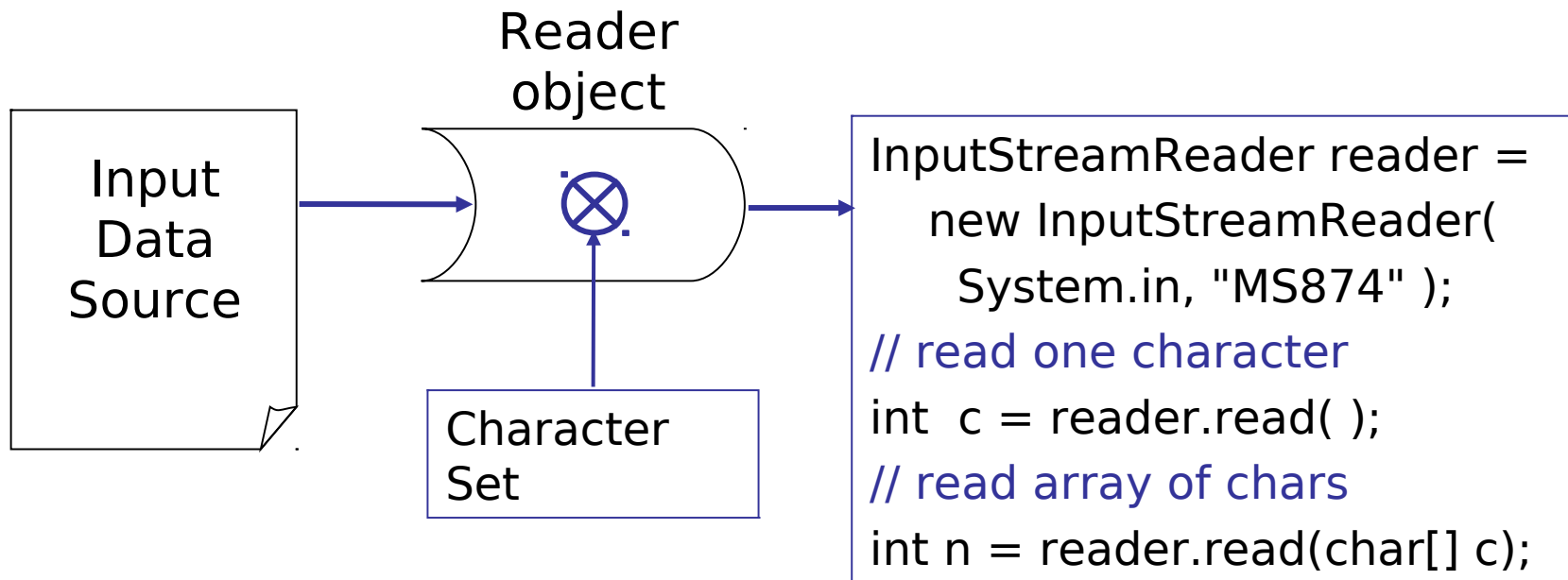
Text Input Class Hierarchy

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



Reader

- Reader: reads bytes and converts to chars.
- Interprets bytes using a **Character Set Encoding**.
- Usually uses an InputStream as input source.



InputStreamReader class

InputStreamReader is a kind of Reader.

It gets data from an InputStream (*any* InputStream object)

```
InputStream in = new FileInputStream( "test" );  
// InputStreamReader "wraps" an InputStream  
InputStreamReader reader =  
    new InputStreamReader(in);  
// read a character  
char b = (char)reader.read( );  
// read several characters together  
char[] buff = new char[1024];  
int nchars = reader.read(buff, 0, buff.length);  
// close the input stream  
reader.close( );
```

Shortcut: FileReader

`FileReader` opens a file and creates `InputStreamReader` for it. (Automatically creates a `FileInputStream`.)

```
InputStreamReader reader =  
    new FileReader("filename");  
// read a character  
char b = (char) reader.read( );
```

BufferedReader class

BufferedReader reads input as Strings.

It uses a [Reader](#) to read characters.

1. Read from System.in

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

2. Read from a file.

```
InputStream in = new FileInputStream("file");  
Reader reader = new InputStreamReader( in );  
BufferedReader br = new BufferedReader(reader);  
// read a line (removes newline char)  
String s = br.readLine( );
```

BufferedReader methods

BufferedReader 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

`ready()` - inquire if there is data ready to be read (avoid blocking)

BufferedReader for File Input

To read from a file, create a `BufferedReader` around a `FileReader`. The `ready()` method returns true if (a) the input buffer contains data or (b) underlying data source is not empty (when reading from file). `ready()` is used to avoid "blocking" the application during a read.

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

BufferedReader and End-of-Data

The `readLine()` method returns `null` if the end of input stream is encountered. Use this to detect the end of input or file.

```
String filename = "mydata.txt";
BufferedReader br = new BufferedReader(
    new FileReader(filename));

// read all data
String line;
while((line=br.readLine()) != null)
{
    // process the data
    System.out.println( line.toUpperCase() );
}
br.close( );
```

Character Sets

There are many **language-specific** character sets:

Extended ASCII - encoding for English, 1 byte per char

ISO-8859-11 - one-byte encoding for Thai chars

TIS-620 - another one-byte encoding for Thai chars

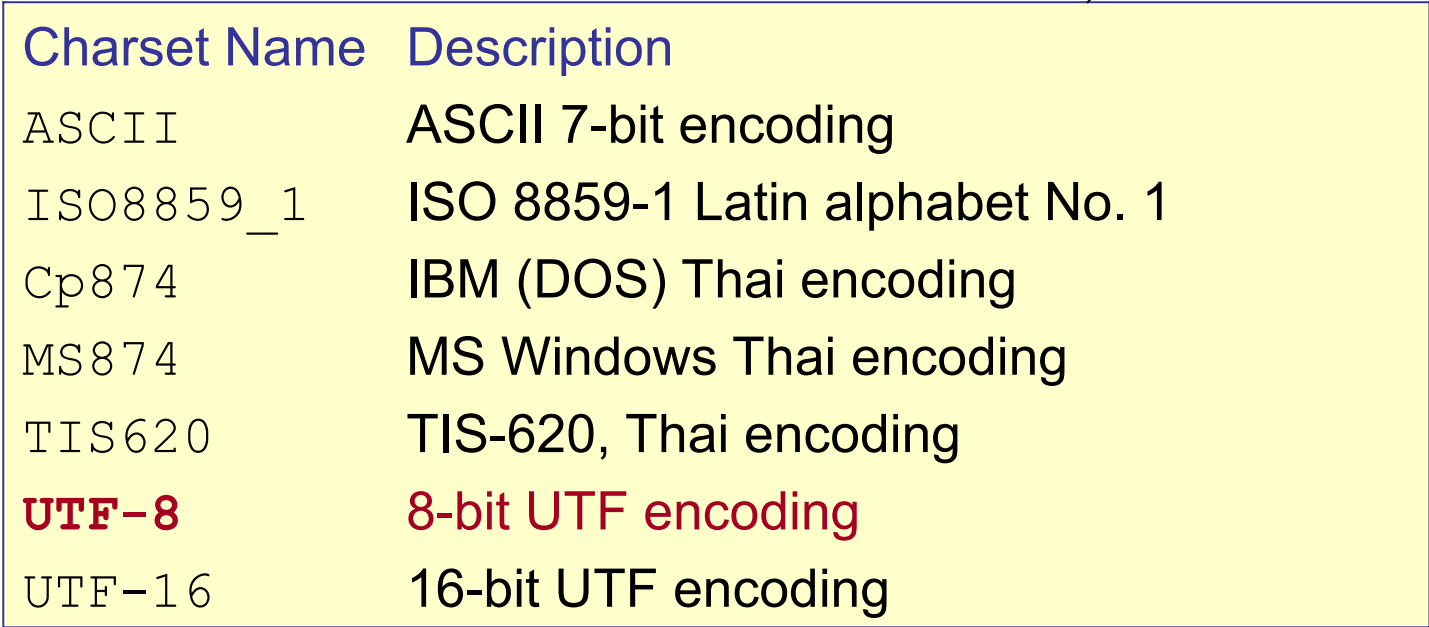
- Having many character sets and encodings means its hard to know what a file contains.
- You have to know the character set and encoding that was used to write text the file.

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 |
| UTF-8 | 8-bit UTF encoding |
| UTF-16 | 16-bit UTF encoding |

Unicode and UTF-8

Unicode is a universal standard that includes character codes for all alphabets. See them all at:

`https://unicode.org`

Most languages use a **variable-length encoding** of Unicode to save space.

By default, Java reads/writes using UTF-8 encoding.

UTF-8 needs 1 byte per char for English alphabet, but 2-3 bytes per char for Thai alphabet.

Unicode Text Example

Write this string to a file (use Java code or a text editor).
If you use a text editor, verify that it uses UTF-8 encoding.

สวัสดี

If you 'type' or 'cat' the file on a terminal it will show
"สวัสดี", because the console also understands UTF-8.

If you look at the **bytes** in the file, it contains:
hexadecimal (2 hex chars = 1 byte)

```
e0b8aae0 b8a7e0b8 b1e0b8aa e0b894e0 b8b5
```

18 bytes! Why?

Using 2-byte per Unicode character, you would expect it to be 12 bytes. UTF-8 makes Thai text longer.

Input Streams & Reader Summary

Java has a Reader class for common InputStream classes.

InputStream

InputStream

FilterInputStream

FileInputStream

PipedInputStream

Reader

InputStreamReader

StringBufferedReader

FilterReader

FileReader

PipedReader

Read Primitive Values in Binary Form

DataInputStream

Scanner

`java.util.Scanner` is a general parser for text files. It also provides data conversion (int, double, String, etc).

`Scanner` reads from an `InputStream` or a `String`.

```
// scanner to read 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 );
```


Using Scanner

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

```
Scanner scanner = new Scanner("3 dogs .5");  
if ( scanner.hasNextInt() )  
    n = scanner.nextInt(); // 3  
if ( scanner.hasNext() )  
    word = scanner.next(); // "dogs"  
if ( scanner.hasNextDouble() )  
    x = scanner.nextDouble(); // 0.5  
// read and discard rest of this line  
scanner.nextLine();
```



Output

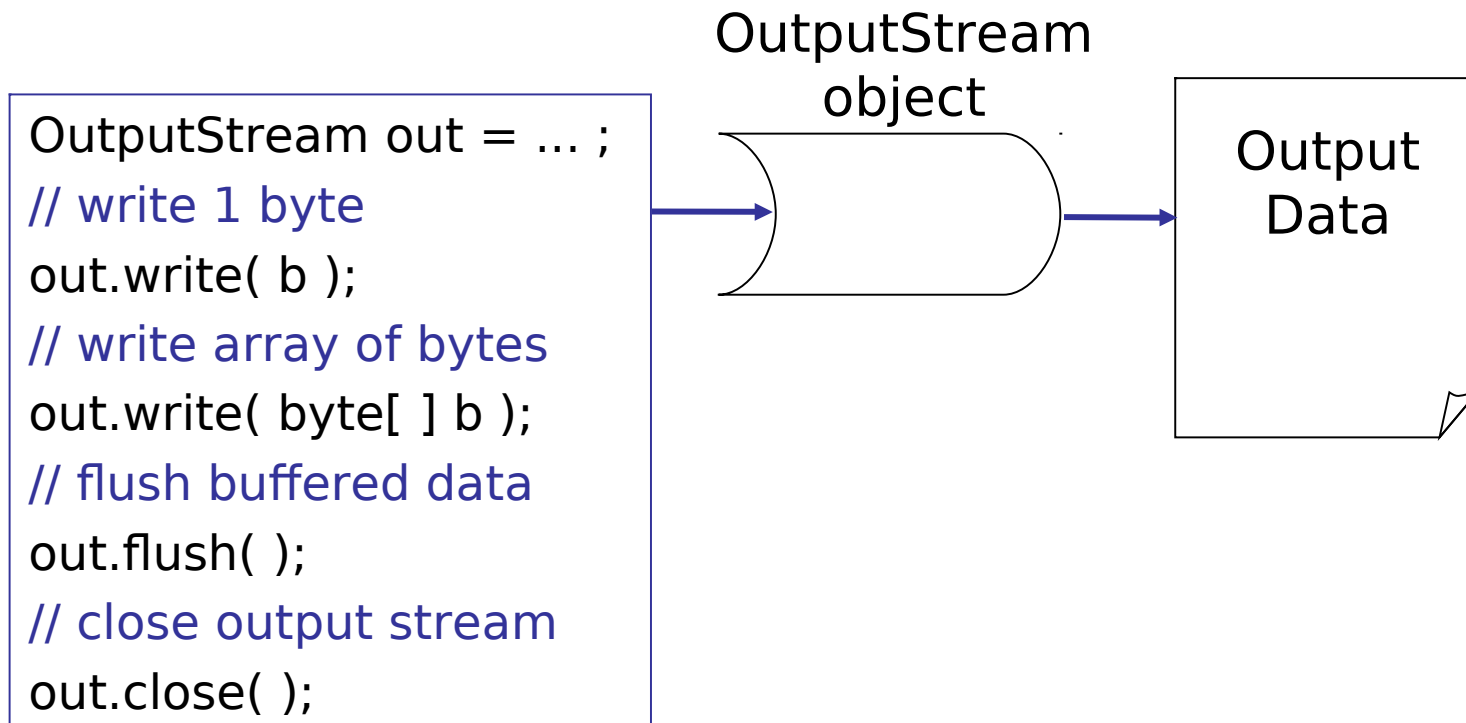
Classes for Output

| | |
|--|---|
| <code>OutputStream</code> | write output as bytes |
| <code>Writer</code> , <code>OutputStreamWriter</code> | write characters and strings to an OutputStream |
| <code>BufferedWriter</code> | writes text (Strings) to an OutputStream |
| <code>PrintStream</code> | like <code>Writer</code> but adds methods to convert other data types to string, and create formatted output. |
| <code>DataOutputStream</code> | write primitive values (byte, int, float, double) in binary format, in a portable way |

The hierarchy of output classes is similar to Java's input classes

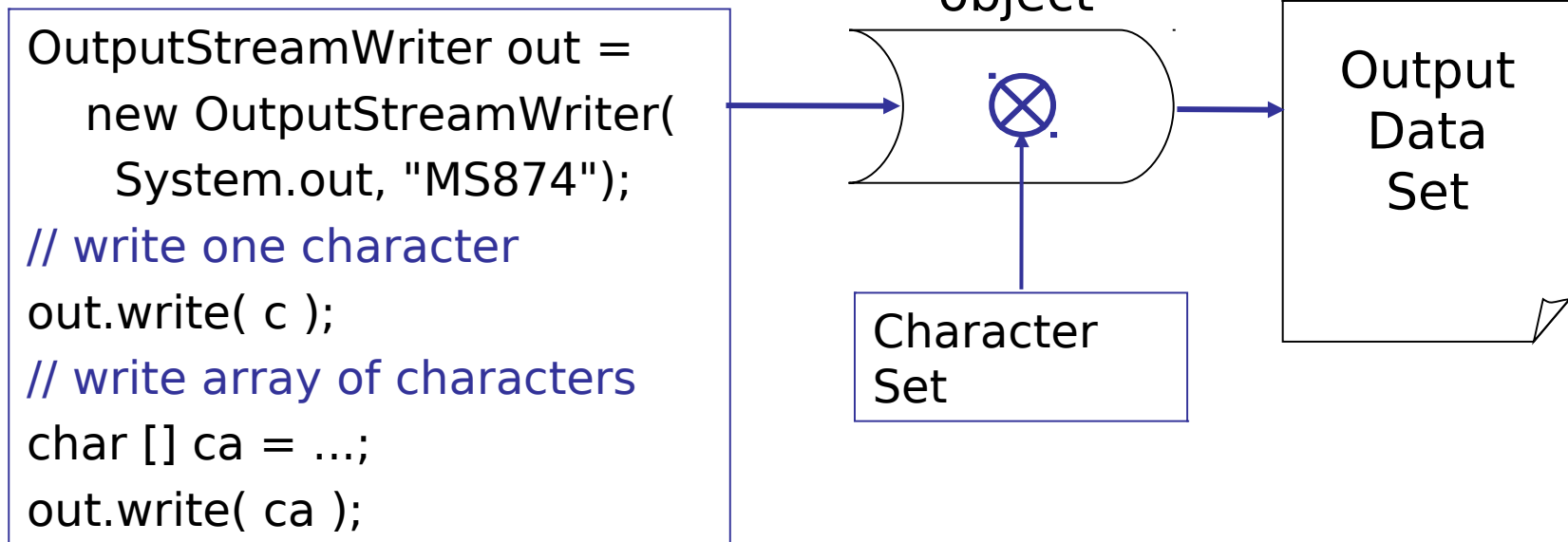
OutputStream

- OutputStream writes **bytes** to some output destination
- No interpretation of characters.
- Has several subclasses, such as FileOutputStream.



Writer

- Writer converts **characters and strings** to bytes.
- Interprets chars according to character set encoding.
- Uses an `OutputStream` to "output" the bytes.



PrintStream

PrintStream is a convenience class for writing to text files.

It is a subclass of `OutputStream`, but methods do not throw `IOException` (so you don't need try - catch).

System.out is a **PrintStream** object.

`print(int n)` - print primitives in text form, e.g. 10, -5

`print(float f)`

etc.

`println(n)` - same as `print()` but appends newline

`printf("format", arg1, arg2, ...)` - formatted output

`format("format", arg1, arg2, ...)` - same as `printf`

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`

`FileWriter`

`PipedWriter`

`StringWriter`

Writing Binary Data

`DataOutputStream`

Handling Exceptions

Two common exceptions thrown by I/O methods:

IOException - cannot perform a read or write

FileNotFoundException - file not found, or you don't have permission to access the file.

Less common, but may occur...

SecurityException - Java's security manager denies access to the file.

Handling Exceptions

Your code must deal with these exceptions in one of two ways:

1. Declare that method may throw exception:

```
public void myMethod(params) throws IOException{  
    // body of method  
}
```

2. Catch the exception and take some action. See next slide.

Catching an Exception

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

Using Files

Many I/O 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 a file
```

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

Testing Files

File has methods to:

- test file existence and permissions (can read?)
- 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.

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

if ( file.isFile() ) {
    /* this is an ordinary file (not a directory or link) */
    long length = file.length( );
    long date = file.lastModified( );
}

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

File Copy Example

Example of how to use a File, but not good code.

You also must catch IOException.

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

FileReader in = new FileReader( infile );
FileWriter out = new FileWriter( outfile );
// reading 1 char at a time is inefficient
int c;
while ( (c = in.read()) >= 0 ) out.write(c);
in.close();
out.flush(); // flush any data from buffer
out.close();
```

Extra Topics (Optional)

1. Reading & Writing Binary Format Data
2. Read without blocking (waiting for input)
3. Random Access versus Sequential Access
4. Using a Pipe - you write data to one end and read it from the other end
5. Unicode for Thai

Reading Binary Data

Examples: MP3 file, image file

Advantages:

- smaller size, faster, less loss of precision in numbers

Has methods for reading primitive values.

```
InputStream in = new FileInputStream( "mydata" );
DataInputStream data = new DataInputStream( in );
try {
    int n = data.readInt( );           // 4 byte integer
    double x = data.readDouble( );    // 8 byte double
    char c = data.readChar( );        // 2 byte unicode
}
catch ( IOException e ) { ... }
```


End-of-File for DataInputStream

- Throws EOFException if end of input encountered while reading.

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

// read doubles until end of file
double sum = 0;
while( true ) {
    try {
        double x = data.readDouble( ); // read 8 bytes
        sum += x;
    } catch ( IOException e ) { ... }
    catch ( EOFException e ) { break; } // End of File
}
data.close( );
```

Writing Binary Data

DataOutputStream methods similar to DataInputStream

```
OutputStream out = new FileOutputStream( "mydata" );
DataInputStream data = new DataInputStream( out );
long studentId = 6010541234L;
double score = 90.3;
char grade = 'A';
try {
    data.writeLong(studentId); // 8 byte long
    data.writeDouble(score); // 8 byte double
    data.writeChar(grade); // 2 byte char
}
catch ( IOException e ) { ... }
```

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 (waiting for input).

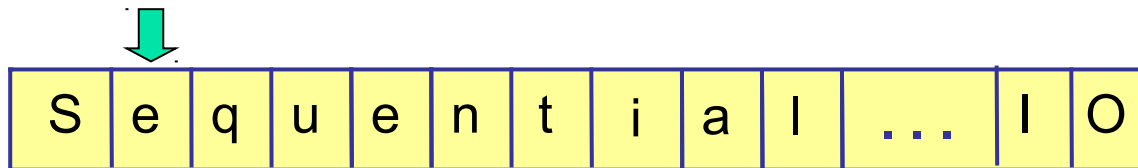
Reader classes have a **`ready()`** method.

```
InputStream in = System.in; // any InputStream

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

Sequential Access

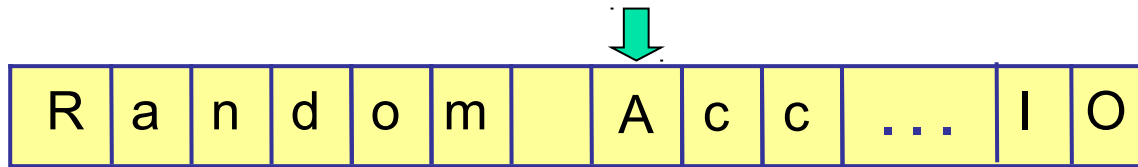
- Read/write sequentially starting from the beginning.
 - Cannot "back up" and reread or rewrite something.
 - Cannot "jump" to arbitrary location in stream.
- InputStream and OutputStream use sequential I/O.
- InputStream has a **skip(n)**, but it is still sequential.



```
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( 7L );           // goto byte 7 ('A')
int b = rand.read();      // read one byte ('A')
```

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 );
```

Reading and Writing Pipes

One object **writes** data into the pipe, another object **reads** data from the pipe.

Useful for multi-threaded applications.

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

Unicode For สวัสดิ์

The Unicode for the Thai chars used in the example are
(with vowels list after the consonant):

ส = 0E2A

ว = 0E27 0E31

ส = 0E2A

ด = 0E14 0E35

Print Unicode in Java:

```
String s = "\u0e2a\u0e27\u0e31\u0e2a\u0e14\u0e35";  
System.out.println( s );
```


More Information

Oracle **Java Tutorials** (online)

Basic I/O

<https://docs.oracle.com/javase/tutorial/essential/io/>

Handling Exceptions

<https://docs.oracle.com/javase/tutorial/essential/exceptions/>