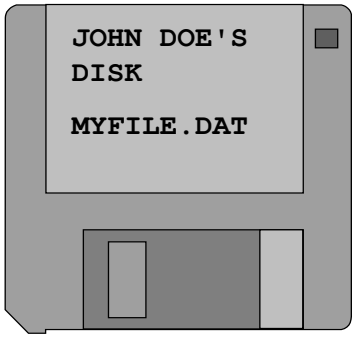
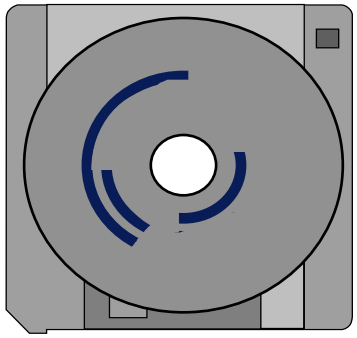
Files, Streams, Filters

Introduction

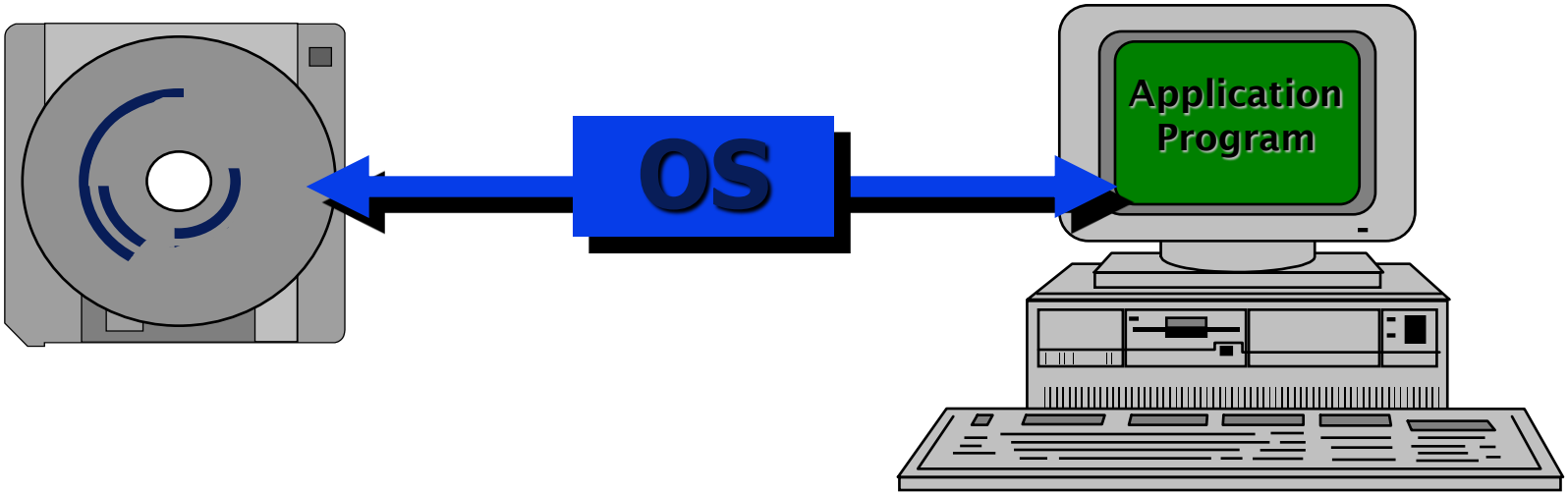
- Files are often thought of as permanent data storage (e.g. floppy diskettes)
- When a file is stored on a floppy or hard disk, the file's data may not be in contiguous sectors
  - This is called the physical file





Introduction

- An application program should not attempt to access the data directly from the disk
- It's the role of the operating system to maintain the physical file and provide an application program access to it as a logical file



Java Support for I/O

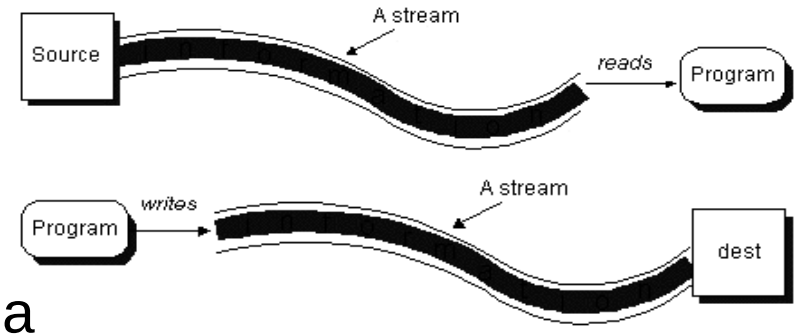
- Lets say we need to make a program that reads bytes
  - For example, we'd like to replace "\r\n" with "\n" in a file
- We don't exactly care where the bytes come from. A program that can read bytes should be equally able to read bytes from:
  - Screen (Keyboard input)
  - Files
  - Network sockets
  - Buffers in memory
  - Pipes
    - Bytes that another program (thread) is concurrently producing
- So, a major goal is to allow programs do I/O without them knowing the exact I/O targets they are using
    - This makes I/O programs independent from specific devices, and reusable in different contexts

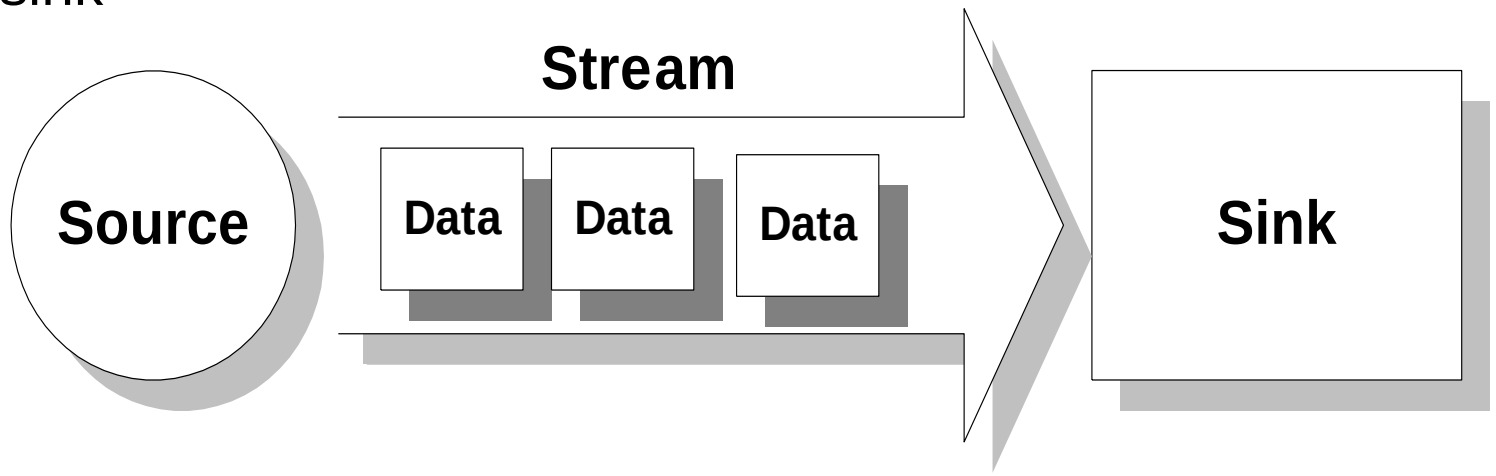
Java Support for I/O

- Operating systems push this unifying idea too:
  - Keyboard input in terminal and files are treated equally
    - One program that can read files, can read keyboard input too, without change
- The solution of Java is the abstraction of "*streams*".
  - An input (output) stream is something that is able to read (write) bytes
  - "Something" can be anything, screen, files, sockets etc.
- A program built upon the notion of streams is shielded from changes of I/O requirements

What are Streams?

- Streams are abstract information flows
- Streams are ordered sequences of data that have a source or destination
  - Data flows into your program from a source
  - Your program processes the data
  - Information flows out of your program to a sink





Using Streams

- No matter where the information is coming from or going to and no matter what type of data is being read or written, the algorithms for reading and writing data is pretty much always the same

Reading	**Writing**
open a stream	open a stream
while more information	while more information
{	{
read information	write information
}	}
close the stream	close the stream

Streams unveiled: InputStream

```
public abstract class InputStream {
  public int read();
}
```

- Why return int and not byte?
  - So as to reserve the value -1 to mark the end of the stream
- Additional methods:

```
public int read(byte[] buffer);
public int read(byte[] buffer, int offset, int length);
```

- How do I know how many bytes were put in the buffer?
  - If the method doesn't return -1, it returns the answer

Streams unveiled: OutputStream

```
public abstract class OutputStream {
  public void write(int b);
  public void write(byte[] b);
  public void write(byte[] b, int offset, int length);
}
```

- Streams also have a close() method
- Ok, how do I write bytes to a file?
  - Easy, create an appropriate OutputStream instance:

```
File outputFile = new File("data.txt");
OutputStream out = new FileOutputStream(outputFile);
out.write(5);
out.close();
```

- FileOutputStream is a concrete subclass of OutputStream

What about character data?

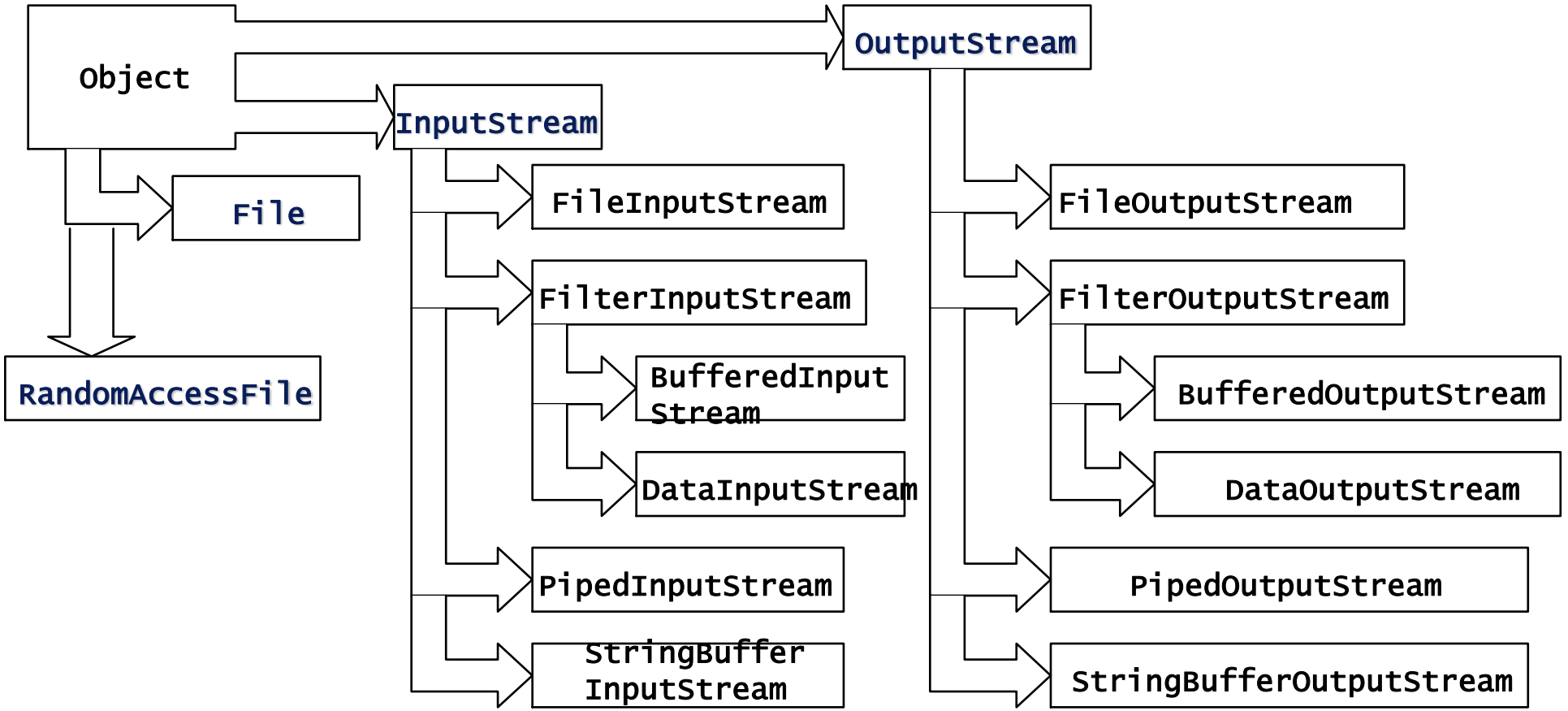
- Java defines the notions of Readers and Writers, that encode/decode characters upon lower-level streams.
- Readers and Writers are identical to InputStreams and OutputStreams, with the difference that, instead of byte, char is used.
  - Remember that Java char represents UTF characters
  - The exact encoding/decoding to/from bytes depend on the encoding scheme.
    - UTF-8, UTF-16, ISO-8859-1, US-ASCII
  - Depending on the scheme, a character can be encoded to one, two, or even four bytes!

How to write characters to a file?

- Use InputStreamReader or OutputStreamWriter classes, that take a stream and wrap it in a reader or writer
  - These act as a bridge from streams to readers/writers
- Obtain an appropriate Writer:

```
File file = new File("output.txt");
Writer out = new OutputStreamWriter(new FileOutputStream(file));
char[] data = { 'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd' };
out.write(data);
out.close();
```

A summary of java.io



I don't want to work with bytes!

- "I want to read/write other data types as ints, floats, doubles, Strings…"
  - Easy, wrap an existing I/O stream into a Data I/O stream:

```
DataOutputStream out = new DataOutputStream(outputStream);
out.writeDouble(Math.PI);
out.writeUTF("Hello, world!");
out.close();
```

- I/O is supposed to be expensive, I don't want to access the physical stream one time per byte access
  - Wrap a buffer around the stream (that reads chunks of bytes and return them one by one):

```
BufferedOutputStream out = new BufferedOutputStream(realStream);
```

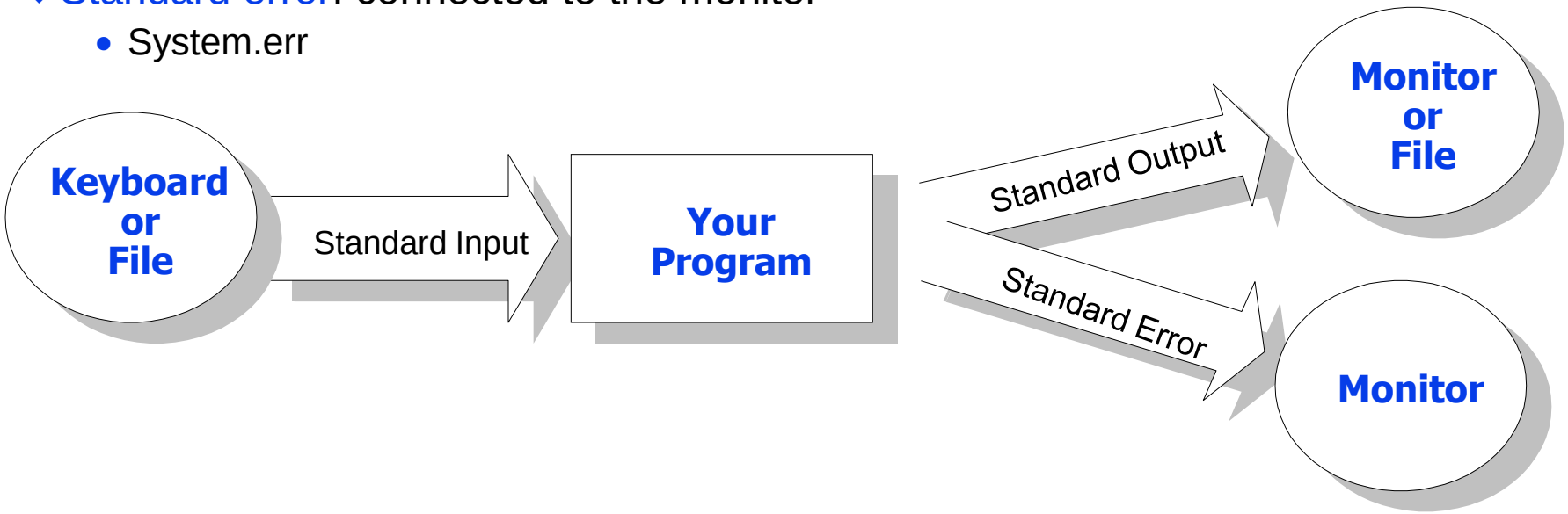
- "realStream" could be a FileOutputStream for example
  - Always wrap low level streams in buffers
    - Unless they already are, like System.out, System.in, System.err
      - *(more on these later)*

The `DataOutputStream` Class

- It also provides the following methods
  - **`writeBoolean(boolean v)`**
  - **`writeByte(int v)`** - same as OutputStream.write(int b)
  - **`writeBytes(String b)`** - 1 byte ascii, similar to:
    - **`OutputStream.write(byte[] b)`**
  - **`writeShort(int v)`**
  - **`writeChar(int v)`**
  - **`writeInt(int v)`**
  - **`writeLong(long v)`**
  - **`writeFloat(float v)`**
  - **`writeDouble(double v)`**
  - **`writeChars(String v)`** - 2 byte unicode
  - **`writeUTF(String str)`** - writes String out in UTF-8 encoding
    - first 2 bytes indicate the length of the String

Built-in Streams

- Java programs automatically open three streams, that can be accessed from the class java.lang.System:
  - Standard input: connected to the keyboard
    - System.in
  - Standard output: connected to the monitor
    - System.out
  - Standard error: connected to the monitor
    - System.err



List of Concrete InputStreams and Readers

Sink Type	Character Streams	Byte Streams
Memory	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
	StringReader StringWriter	
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream
File	FileReader FileWriter	FileInputStream FileOutputStream

List of Concrete OutputStreams and Writers

Process	Character Streams	Byte Streams
Buffering	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtering	FilterReader, FilterWriter	FilterInputStream, FilterOutputStream
Converting between bytes and characters	InputStreamReader OutputStreamReader	
Concatenation		SequenceInputStream
Object Serialization		ObjectInputStream, ObjectOutputStream
Data Conversion		DataInputStream, DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking Ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

Random Access Files

- The problem with sequential-access files is that the file must be closed and reopened to reposition the file position pointer
  - Useful for "database" types of application
- The logical file appears to the program as a sequence of bytes
  - A pointer determines the location of the next read or write
  - The methods **`seek()`** and **`getFilePointer()`** can be used to position the file

File Input and Output

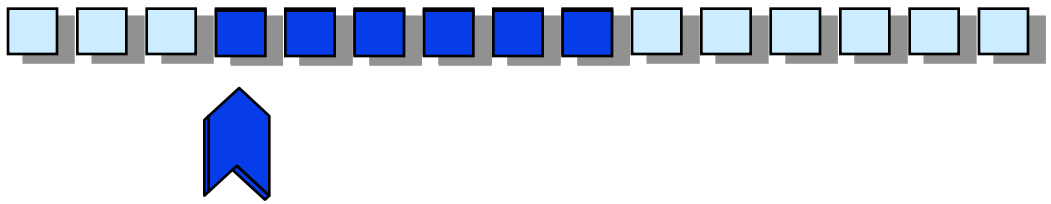
- The member function **`write`** copies data from memory into the file starting at the pointer

```
file.write(byte_buf, 0, 6);
```

An array of bytes

Starting byte

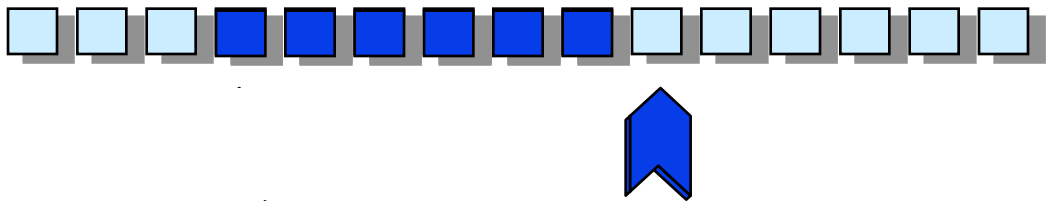
The number of bytes to write



File Input and Output

- The member function **`read`** copies data from memory into the file starting at the pointer

```
file.read(byte_buf, 0, 6);
```

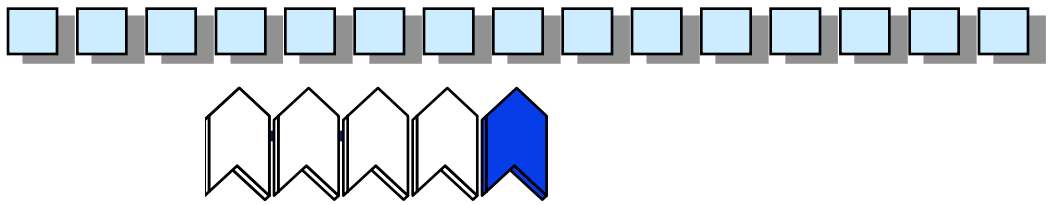


File Input and Output

- The member function **seek** positions the pointer

```
file.seek(8);
```

Move to byte position 8



The RandomAccessFile Class

- The class **RandomAccessFile** can be used to process Random Access files
  - **RandomAccessFile** is used to both read and write to files so is not derived from **InputStream** or **OutputStream**
  - Therefore **RandomAccessFile** can not be chained to other **FilterInputStream** or **FilterOutputStreams** (such as **DataInputStream**)
  - For convenience **RandomAccessFile** provides the same methods that **InputStream**, **OutputStream**, **DataInputStream** and **DataOutputStream** provide
  - However because **RandomAccessFile** is neither an **InputStream** or **OutputStream** it can not be chained to a **BufferedInputStream** or **BufferedOutputStream** and hence can not be buffered

Opening Random Access Files

- Opening a random-access file is similar to the ANSI C command **`fopen()`**
- The constructor takes a file name and a mode

```
RandomAccessFile(filename, mode);
```

```
RandomAccessFile inputFile = new
  RandomAccessFile("client.dat", "r");
RandomAccessFile ioFile = new
  RandomAccessFile("client.dat", "rw");
```

- The mode can either be
  - **r** - read only
  - **rw** - read and write
- The mode can not be anything else
  - If the mode is neither "r" or "rw" then an **`IllegalArgumentException`** is thrown
  - **`IllegalArgumentException`** is a **`RuntimeException`** so does not have to be specified or caught

Using Random Access Files

- The RandomAccessFile class also provides other methods for reading and writing data
  - **`readBoolean, writeBoolean`**
  - **`readChar, writeChar`**
  - **`readDouble, writeDouble`**
  - **`readInt, writeInt`**
  - **`readLong, writeLong`**
  - **`readUTF, writeUTF`**

Things to come next

- Command line arguments
- More on buffering
- More on character I/O
- Printing to a stream
- Formatted I/O
- Measuring time
- Serialization
- Text parsing and regular expressions

Command Line Arguments

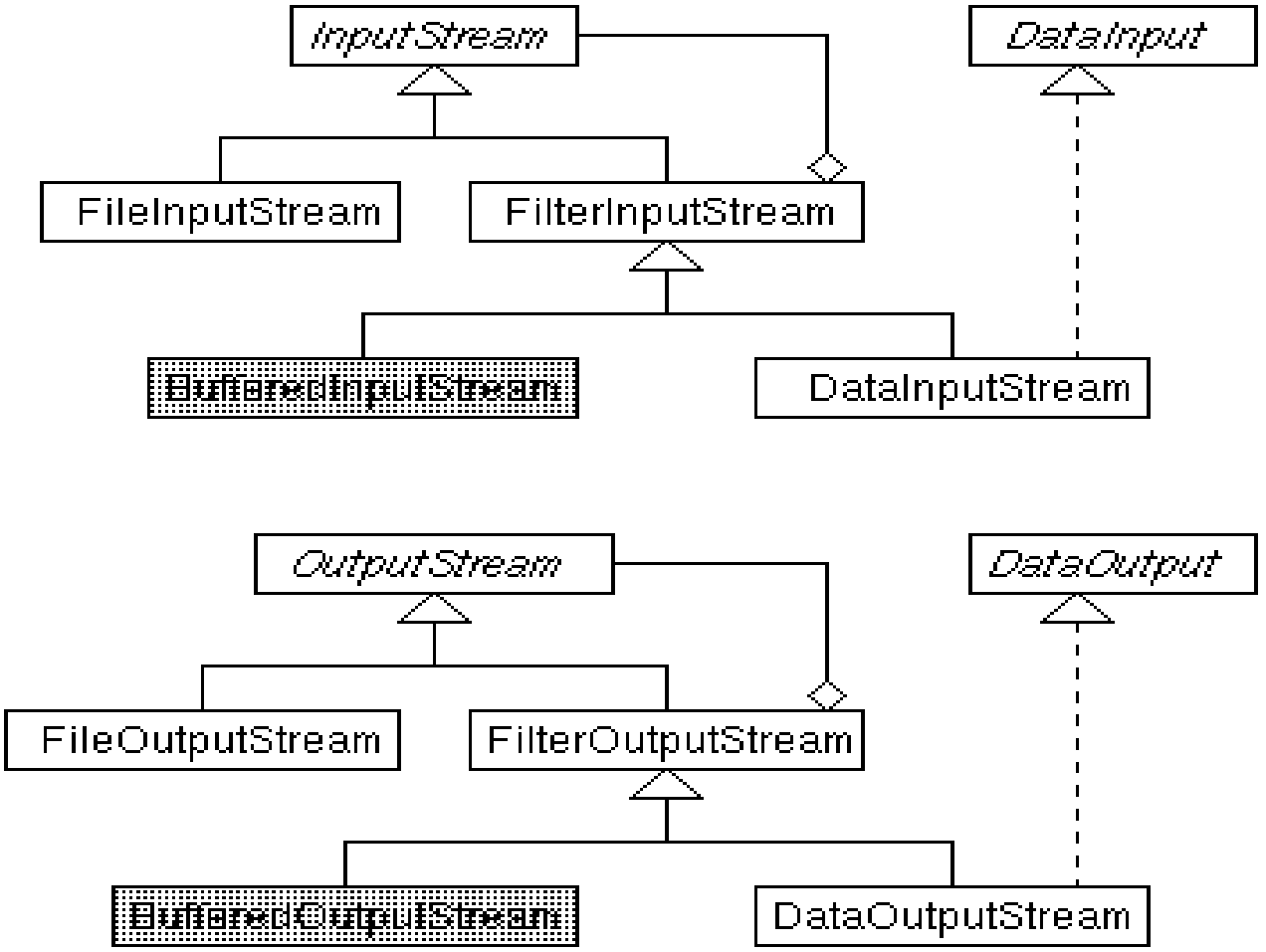
- Parameters passed to the **main()** method of your program:
- `C:>java Program arg1 arg2 “a r g 3”`
- `String[] args = {“arg1”, “arg2”, “a r g 3”};`
- How many are there?
  - **`numArgs = args.length;`**
- How to use them?

```
public static void main(String[] args) {
  System.out.println('#Args:',numArgs);
  for (i = 0; i < numArgs; i++) {
    System.out.println(args[i]);
  }
}
```

Buffering and Wrappers

- Buffering means that Java reads a block even when you request a byte
  - You get the processing advantages of byte-by-byte access but the speed of using byte arrays
- The classes **`BufferedOutputStream`** and **`BufferedInputStream`** can be used to buffer streams
  - When data is written to a **`BufferedOutputStream`** the data may not actually get written to the output device
  - Data written to the **`BufferedOutputStream`** is stored in a buffer
- When the buffer is full it is written to the output device
  - Writing the buffer to the output device is called a physical output operation
  - Whereas writing to the buffer is called a logical output operation
- Because typical physical input and output operations are extremely slow (eg. writing to disk) compared to typical processor speeds, buffering outputs normally yields significant performance improvements over unbuffered outputs

Byte Streams

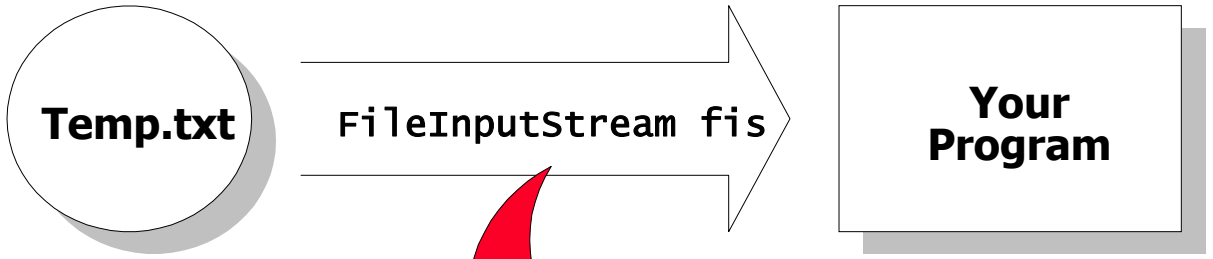


BufferedInputStream

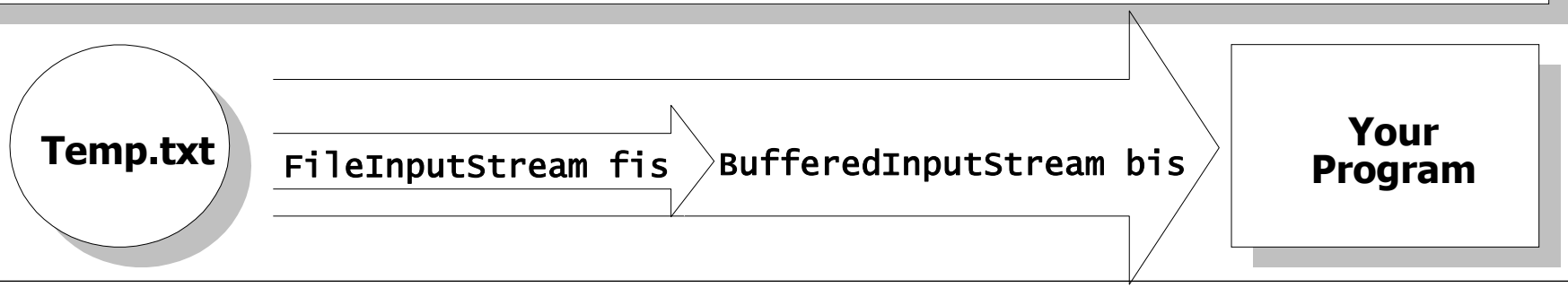
- **BufferedInputStream** provides buffering
  - Doesn't take a filename as an argument
  - How do you buffer a **FileInputStream**?
  - Answer: "wrap" the **FileInputStream** inside :

```
DataInputStream inputFile =
    new DataInputStream(
          new BufferedInputStream(
                new FileInputStream("client.dat")
          )
    );
```

```
FileInputStream fis = new FileInputStream("Temp.txt");
```




```
BufferedInputStream bis = new BufferedInputStream(fis);
```



BufferedOutputStream

- **BufferedOutputStream** is derived from **FilterOutputStream** (like **DataOutputStream**) and can be connected to an **OutputStream**

```
BufferedOutputStream output = new
    BufferedOutputStream(new
        FileOutputStream("client.data"));
```

- The **BufferedOutputStream** can also be combined with the **DataOutputStream**

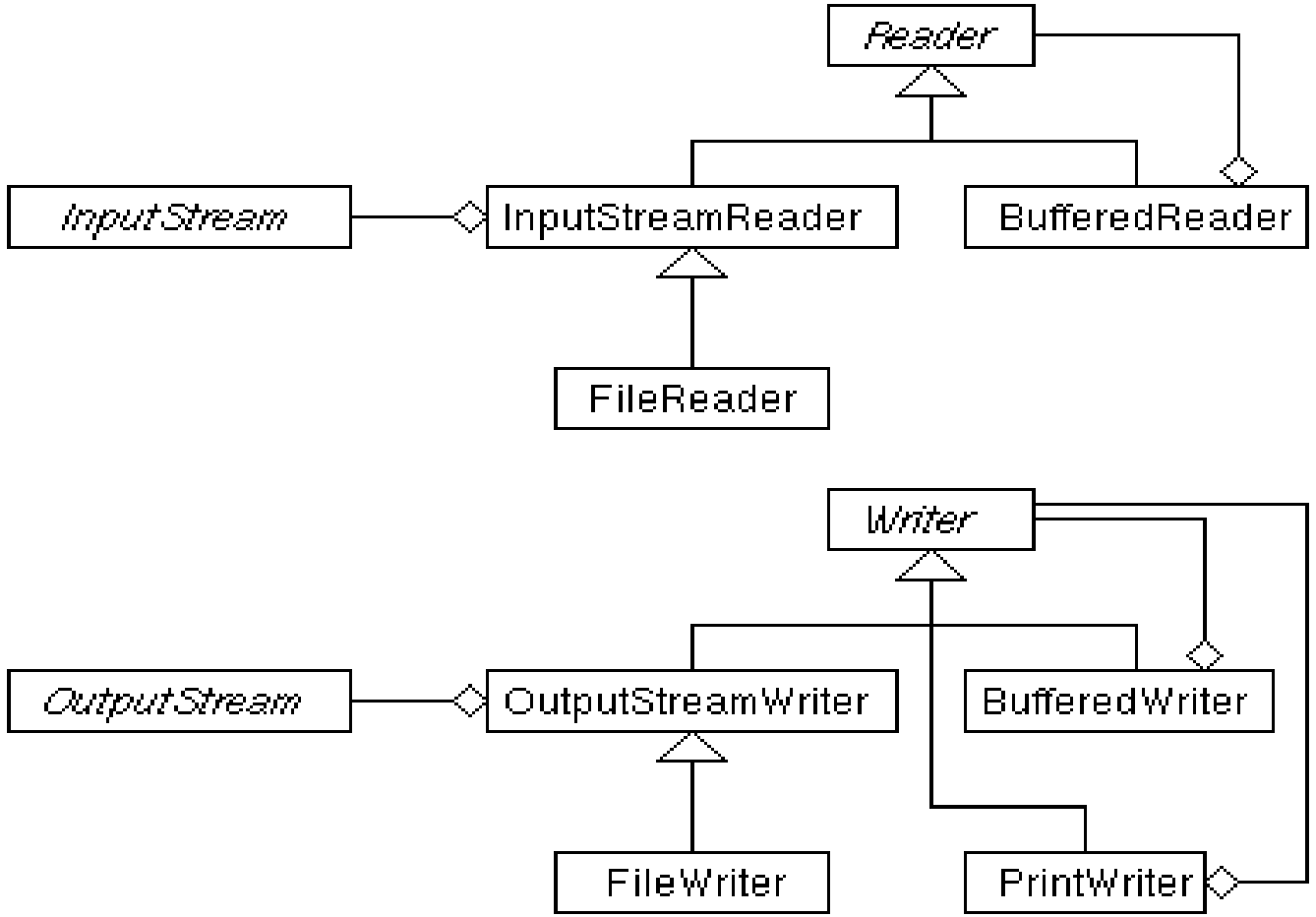
```
DataOutputStream outputFile = new
    DataOutputStream(new BufferedOutputStream(new
        FileOutputStream("client.dat")));
```

- A partially filled buffer can be forced out to the buffer using **flush**
  - **testBufferedOutputStream.flush()**

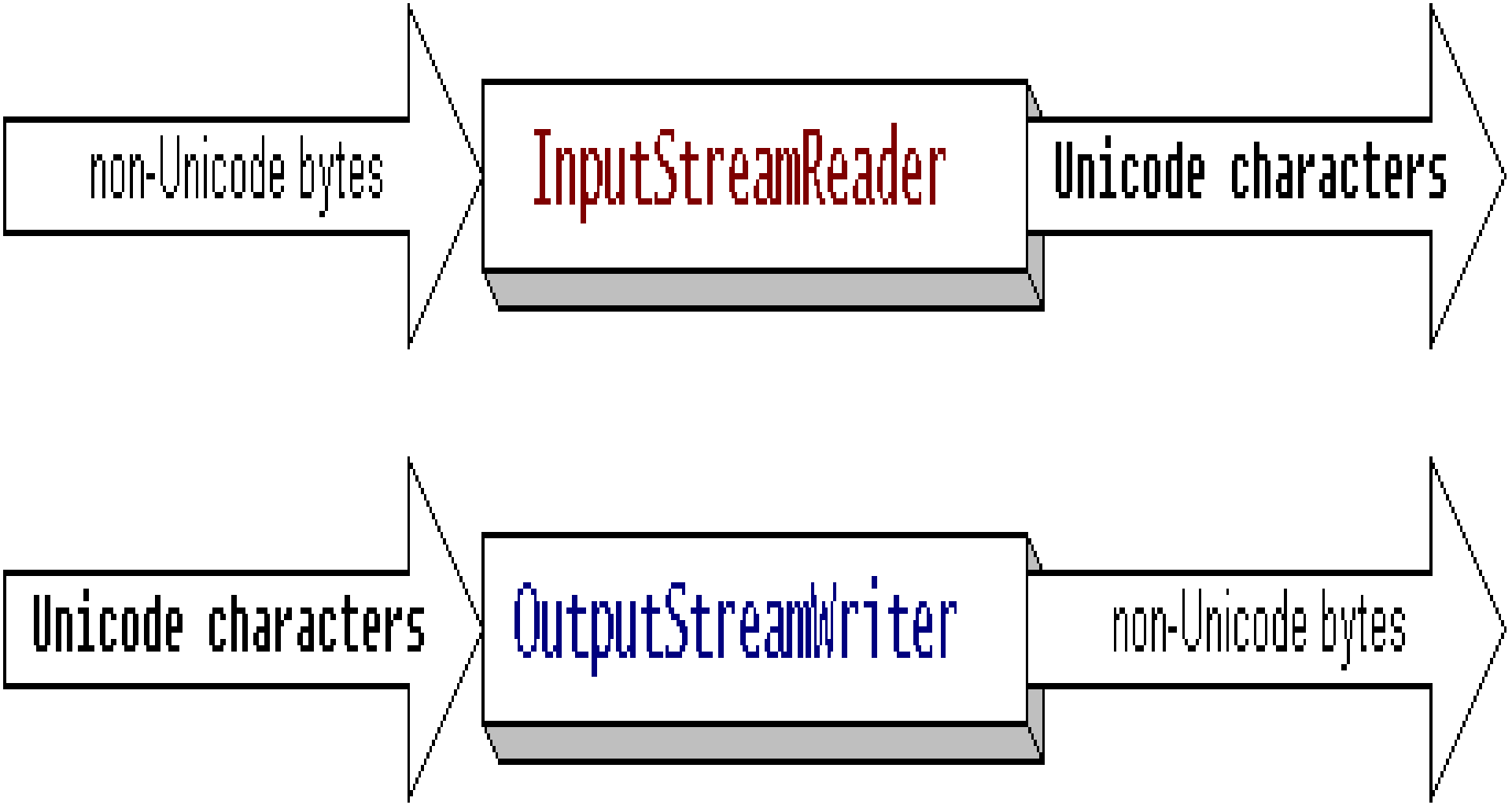
Character-Based I/O

- In Java, characters are 16 bit Unicode, not bytes
- In JDK 1.1, Sun introduced the **`Reader`** and **`Writer`**
  - Work like Input/Output stream hierarchy
  - Preferred method of dealing with text in Java world
- Reading lines using **`BufferedReader`**
  - 1. Construct a **`FileReader`** object using a filename
  - 2. Wrap the **`FileReader`** in a **`BufferedReader`**
  - 3. Use the **`readLine`**`()` method

Character Streams



Unicode Reader/Writer



Character Encoding

- By default, the character encoding is specified by the system property;

```
file.encoding=8859_1
```

- You can use other encoding by doing the following

```
BufferedReader in =
  new BufferedReader(
    new InputStreamReader(
      new FileInputStream("foo.in"), “8859_7"));

PrintWriter out =
  new PrintWriter(
    new BufferedWriter(
      new OutputStreamWriter(
        new FileOutputStream("foo.out", " 8859_7 "))));
```

Using Reader and Writer

```
BufferedReader in
   = new BufferedReader(
       new FileReader("foo.in"));

BufferedReader in
   = new BufferedReader(
       new InputStreamReader(System.in));

PrintWriter out
   = new PrintWriter(
       new BufferedWriter(
         new FileWriter("foo.out")));

Writer out
   = new BufferedWriter(
       new OutputStreamWriter(System.out));
```

Reading Strings

```
import java.io.*;
  …
public … main( String[] args ) throws IOException
{
  String s;
  InputStreamReader ir =
                  new InputStreamReader(System.in);
  BufferedReader in = new BufferedReader( ir );

  while( (s = in.readLine()) != null ) {
      System.out.println( s );
  }
  …
}
```

Reading and Writing Files

```
FileInputStream infile =
   new FileInpputStream( “myfile.dat” );

FileOutputStream outfile =
   new FileInpputStream( “results.dat” );

infile.read(…);
outfile.write(…);
```

Working with Lines

```
BufferedReader in = null;
  BufferedWriter out = null;
  try {
    FileReader fr = new FileReader(fi);
    in = new BufferedReader(fr);
    FileWriter fw = new FileWriter(fo);
    out = new BufferedWriter(fw);
  } catch (IOException ioe) {
    System.err.println("Couldn't open");
  }
```

Reading and Writing Lines

- Use the **`readLine()`** method
  - Returns a **`String`** if successful, **`null`** on failure
- Use the **`write()`** method for output
  - Three argument version: **`String`**, **`start`**, **`size`**
  - Add a newline using the **`newLine()`** method
    - Platform independent way to do output
    - File may change when copied on different platforms

PrintWriter

- A **PrintWriter** prints formatted representations of objects to a text-output stream
- Flushing does not occur until the **flush**() method is invoked
  - It is possible to enable automatic flushing, which causes a flush to take place after any **println**() method is invoked. The output of a newline character does not cause a flush
- Methods in this class never throw I/O exceptions

Display Formatting

- The **PrintWriter** class converts binary data to human-readable form for display
  - **PrintWriter**(Writer out, boolean autoflush);
  - **PrintWriter**(OutputStream out, boolean autoflush);
    - Use to convert System.out to a **PrintWriter**
    - PrintWriter pw=new PrintWriter(System.out,true);
  - Does not throw an exception
    - You must call the method **checkError**()
  - Most used methods are **print**() and **println**()
  - Also format/printf methods (synonyms)
    - pw.printf("%d %s %d = %d", 5, "+", 3, 8);
      - prints "5 + 3 = 8"

Measuring Time

- static long `System.currentTimeMillis()` : Returns the current time in milliseconds
- Example:

```
long time = -System.currentTimeMillis();
doSomeLengthyWork();
time += System.currentTimeMillis();
System.out.printf("Time elapsed: %d ms", time);
```

- If finer accuracy is needed, `System.nanoTime()` can be used. It returns the current time in nanoseconds (10^-9 sec):

The Class `File`

- So far we had a look at classes for processing files
- Java also provides the **`File`** class for finding out information about files
  - The **`File`** class does not actually open the file or do any IO operations
  - It just represents a single file
- Example operations would include determining whether a **`File`** is a file or a directory or whether it exists at all
- For compatibility between operating systems Java allows the path separator character to be either '/' or '\'
- These can even be combined
  - **`c:\java/README`**
- There are three ways to create a file object

Creating a `File` object

- **`public File(String name)`**
  - This constructor takes as an argument the path information as well as a file or directory name
  - An absolute path contains all the directories starting with the *root directory* that lead to a specific file or directory
  - A relative path contains a subset of the directories leading to a specific file or directory
  - Relative paths start from the directory in which the application was started
- **`public File(String pathToName, String name)`**
  - Uses argument **`pathToName`** (an absolute or relative path) to locate the file or directory by **`name`**
- **`public File(File directory, String name)`**
  - Uses the previously created **`File`** object **`directory`** (an absolute or relative path) to locate the file or directory specified by **`name`**

Using a File Object

boolean canRead()
boolean canWrite()
boolean exists()
boolean isFile()
boolean isDirectory()
boolean isAbsolute() - true if the arguments passed to the constructor indicate an absolute path
String getAbsolutePath()
String getName() - the name of the file or directory (excluding path)
String getPath() - the path to the file or directory
String getParent() - parent directory of the file directory
long length() - length of the file in bytes
long lastModified - system dependent representation, only useful for comparing with other values
String[] list() - returns an array of Strings representing the contents of a directory

Serialization: Reading / Writing Objects

- The **`ObjectInputStream`** and **`ObjectOutputStream`** allow you to read and write objects
- The process of reading and writing objects is called Object Serialization
- Writing

```
FileOutputStream fileOut = new FileOutputStream("tab");
ObjectOutputStream out = new ObjectOutputStream(fileOut);
Hashtable hash = getHashtable();
Out.writeObject( hash );
```

- Reading

```
FileInputStream fileIn = new FileInputStream("tab");
ObjectInputStream in = new ObjectInputStream(fileIn);
Hashtable newHash = (Hashtable)in.readObject();
```

- Easiest way to send (or save) complex objects, or whole graph of objects

Network Streams

```
// Network related stuff.
Socket sock = new Socket( … );
sock.connect( … );

// Obtain the streams
InputStream is = sock.getInpuStream();
OutputStream os = sock.getOutputStream();


// Network ? You are now working with streams
is.read();
os.write(…);
```

Working with Text: Parsing

- `java.util.Scanner` : provides functionality similar to C's "scanf".

```
Scanner in = new Scanner(System.in);
System.out.println("Type one integer: ");
while (!in.hasNextInt())
    System.out.println("Not an integer: " +
                       in.next() + ", retry: ");
int i = in.nextInt();
```

- It can read any basic type, and use any delimiter (to separate different tokens)
- Check the javadoc of Scanner to see the whole list of the available methods

Working with Text: Modifying

- It is reminded that Strings in Java are *immutable*
  - **`"Hello".toLowerCase()`** will produce a new String with value "hello"
  - This eliminates a most common source of subtle errors in C code, where it is not safe to pass around a char[], as it can be changed everywhere, anytime
    - In Java, it will just never change
- How to split a string?
  - **`public String[] split(String pattern);`**

  `"One_Two_Three".split("_") yields {"One","Two","Three"}`
- How to replace contents in a string?
  - **`public String replaceAll(String pattern, String replacement)`**

`"One_Two_Three".replaceAll("_T", "-blah-") yields:`

`"One-blah-wo-blah-hree"`

Working with Text: Modifying

- If you need to create a string from smaller parts, use a **`StringBuilder`**:

```
StringBuilder sb = new StringBuilder();
sb.append("hello");
sb.append("oops");
sb.append("world");
String s1 = sb.toString(); // s1 == "hellooopsworld"
sb.replace(5, 9, " ");
String s2 = sb.toString(); // s2 == "hello world"
sb.reverse();
String s3 = sb.toString(); // s3 = "dlrow olleh"
```

- When you type "text" + 5 + "other text", javac compiles to:
- new StringBuilder().append("text").append(Integer.toString(5)).append("other text").toString()
  - Don't worry about the chained method calls: each method returns the same string builder, for less typing

Working with Text: Regular Expressions

- Regular expressions define text patterns
- Allows to match these patterns to concrete text instances
- Example:

```
//a fraction is of pattern "number/number"
boolean isFraction(String text) {
  // "+" means "one or more"
  return text.matches("[0-9]+/[0-9]+");
}
➔ isFraction("15/777") returns true
boolean isCsdCourse(String text) {
  return text.matches("hy[0-9]{3}");
}
➔ returns true only when "hy" is succeeded by 3
 digits
```

Working with Text: Regular Expressions

- **`Reg.Ex`**. full support resides in classes Pattern and Matcher in `java.util.regex`
- Pattern defines a textual pattern. It is compiled once, and creates Matcher objects that match the pattern against concrete instances of text
- A Matcher object is used to find the pattern, once or repeatedly, against some text. It also allows to capture parts of the text that matches.
  - This text must be enclosed in brackets, in the pattern definition, to be captured. It is then placed in ***groups*** that can be accessed, when a successful match is performed.

Working with Text: Regular Expressions

- Example: Parsing dates of form “Sat DD-MM-YYYY”

```
Pattern pattern = Pattern.compile(
  “\\w{3}” // three letters (\\w is equal to [a-zA-Z])
  + “\\s+” // \\s is whitespace (space, tab, newline etc)
  + “(\\d{1,2})” // \\d is digit, or [0-9]. We need one or two digits
  + “-(\\d{1,2})” // note the brackets around the digits: this means we want to capture this
    text
  + “-(\\d{4})” );
Matcher m = pattern.matcher(“Fri 20-8-1982”);
boolean found = m.find();
if (found) {
  String dayField = m.group(1); // “20”
  String monthField = m.group(2); //“8”
  String yearField = m.group(3); //“1982”
}
```

- Consult java.util.regex.Pattern’s javadoc for a complete reference of Java support for reg.ex.
- Consider the amount of code you would write to manually extract the same data

Summary

- Java's file access is stream based and offers a number of classes based on **InputStream** and **OutputStream** to work with streams
- Using streams allows the programmer to use a common model whether working with files, networks or command lines
- Some stream classes can be connected or chained to other stream classes to provide additional functionality
- Examples are **DataInputStream** for reading in primitive data types and **BufferedInputStream** for buffering the input to improve performance
- A separate class **RandomAccessFile** is used for accessing random-access files but includes most of the methods provided by **DataInputStream** and **DataOutputStream**
- A **File** class is provided to found out information about a file

Summary

- **Reader**s allow streams to be interpreted as character streams
- **InputStreamReader** converts bytes from an input stream into characters
- **BufferedReader** buffers a **Reader** like a **BufferedInputStream** buffers an **InputStream** and provides the **readLine()** method
- **FileReader** is used to read in from a file
- **Regular expressions** is a powerful way to analyze text

FileEcho

```
import java.io.*;
public class FileEcho {
  public static void main( String args[] ) {
    if ( args.length>0 ) {
      BufferedReader in = null;
      try {
        in = new BufferedReader( new FileReader(args[0]));

        while ( in.ready() ) {
          char ch = (char)in.read();
          System.out.print( ch );
        }
      }
      catch ( FileNotFoundException e ) {
        System.out.println( "File not found" );
      }
      catch( IOException e ) {
        System.out.println( "Read error" );
        System.exit(1);
}}}}
```

InputEcho

```
import java.io.*;
public class InputEcho {
  public static void main( String args[] ) {
    // Set things up to read from the keyboard
    BufferedReader
      keyboard = new BufferedReader( new InputStreamReader(
                                                System.in ) );
    // Read stuff form input and dump to output
    try {
      String inString;
      while ( ( inString = keyboard.readLine() ) != null ) {
        System.out.println( inString );
      }
    } catch ( IOException e ) {
        System.err.println( "InputEcho:  I/O error" );
        System.exit( 1 );
}}}
```

ReadNums (1/2)

```
import java.util.*; import java.io.*;
public class ReadNums {
  public static void main( String args[] ) {
    // Make sure the number of arguments is correct
    if ( args.length != 1 ) {
      System.err.println( "Usage:  ReadNums sourceFile" );
      System.exit(1); }
    // Initialize src since the assignment is done inside
    // a try block
    BufferedReader src = null;
    // Attempt to open the file for reading
    try {
      src = new BufferedReader( new FileReader( args[0] ) );
    }
    catch ( FileNotFoundException e ) {
      System.err.println( "ReadNums:
                                  Unable to open source file" );
      System.exit(1); }
```

ReadNums (2/2)

```
// Read the numbers a line at a time from the source file
    Vector data = new Vector();
    try {
      String line;
      while ( ( line = src.readLine() ) != null ) {
        try {
          int num = Integer.parseInt( line );
          data.addElement( new Integer( num ) );
        }
        catch ( NumberFormatException e ) {} }
      src.close();
    }
    catch ( IOException e ) {
      System.err.println( "ReadNums: " + e.getMessage() );
      System.exit(1); }
    // Print out the results
    for ( int i=0; i<data.size(); i++ )
      System.out.println( data.elementAt( i ) ); }}
```