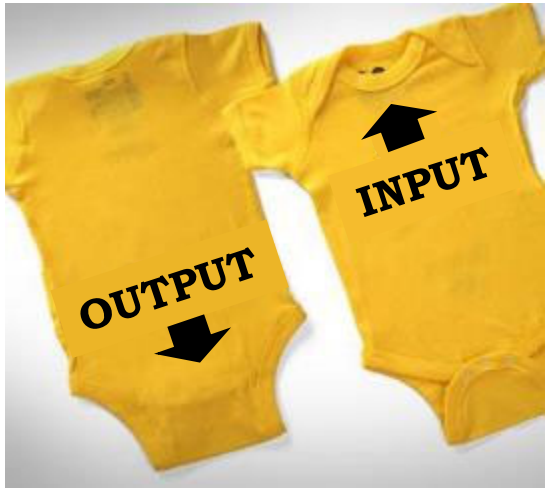
UNIVERSITÉ Concordia UNIVERSITY

COMP 249:
Object Oriented Programming II

Chapter 10

Java I/O



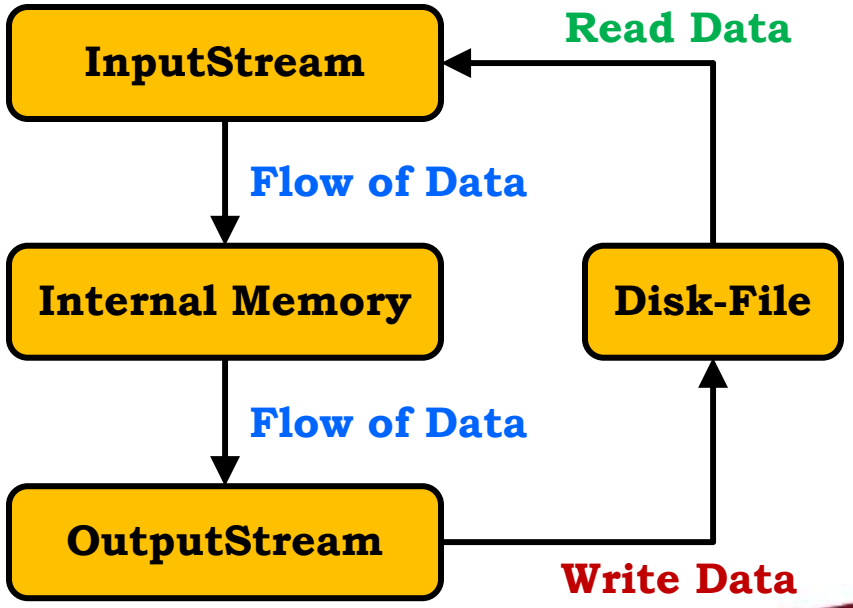
http://odditymall.com/input-output-baby-onsie

Streams

A ***stream*** is a **sequence of bytes** that flow from a source to a destination

- ***ex:*** input file, output file, keyboard, screen, …

The **java.io** package contains many classes that allow us to define various streams



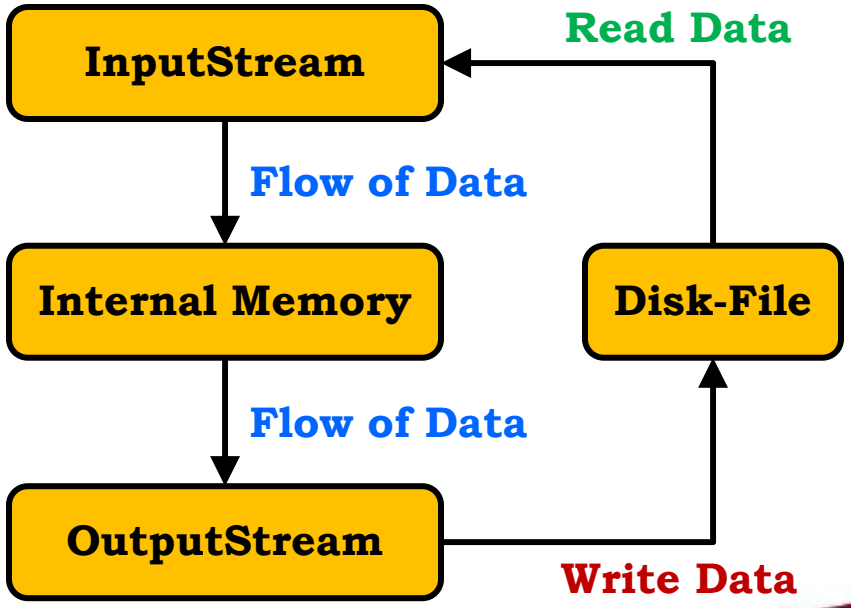
Streams

Input streams flow into your program

- ***ex:*** from a keyboard or from a file
  - **System.in** is an input stream

Output streams flow out of your program

- ***ex:*** to a screen or to a file
  - **System.out** is an output stream



Standard I/O

There are 3 standard I/O streams defined as 3 variables in the **System** class

1. **static InputStream** **in**
   - called ***standard input***
   - defined by **System.in**
   - usually a keyboard
2. **static PrintStream** **err**
   - called standard error
   - defined by **System.err**
   - usually, a monitor and in red
3. **static PrintStream** **out**
   - called standard output
   - defined by **System.out**
   - usually a monitor

RedirectionDemo.java

Can **change** the standard I/O streams

- System.setIn, System.setOut, System.setErr

ex:

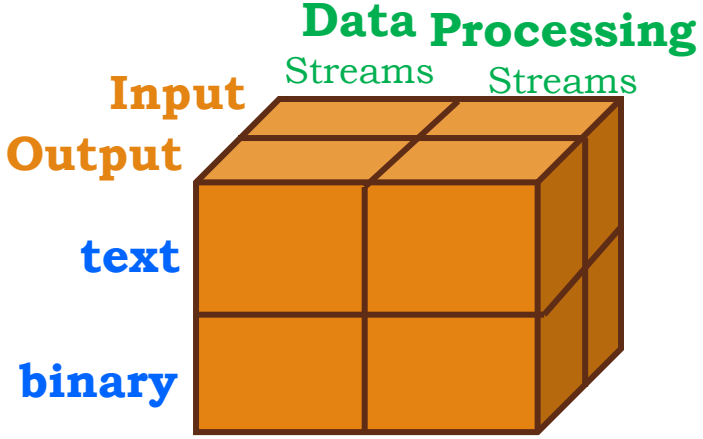
```
PrintStream out = new PrintStream(
                      new BufferedOutputStream(
                      new FileOutputStream("test.txt")));

System.setOut(out);
System.out.println("This is a test.");
out.close();
```

I/O Streams

Input vs **Output** Streams

- input: read information
- output: write information



Text vs **Binary** Files

- text format (characters)
- byte format (binary information)

Data vs **Processing** Streams

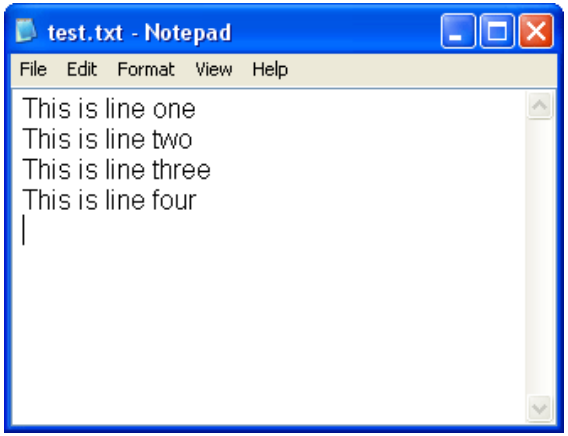
- data stream: acts as either a **source** or **destination**
- processing stream: **alters** or **manipulates** the basic data in the stream

Text File

Data is represented as a sequence of **characters** (also called ASCII file)

- integer 12345 stored as 5 characters (5 bytes)

  '1' '2' '3' '4' '5'



Advantages:

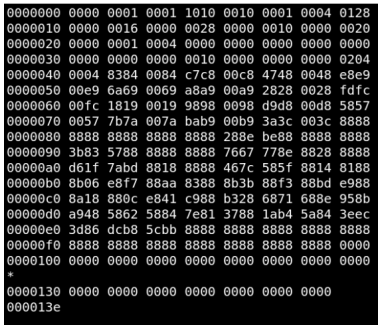
- Human-readable form
- byte format (binary information)

Disadvantages:

- less compact
- less efficient

Use classes **Reader** and **Writer** and their subclasses (will look at details in a few slides)

Binary File



Data is represented as a sequence of **bytes**

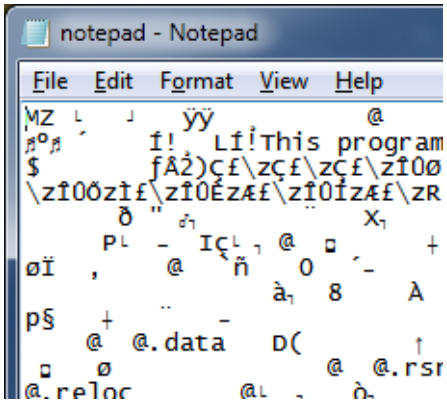
- ***ex:*** integer 12345 stored as four bytes (0 0 48 57)
  (12345 = 48 x 2^8 + 57)

Advantages:

- more compact and more efficient
- native representation

Disadvantages:

- **cannot be read** by humans



- data can be stored **differently** from machine to machine

But: in java, **more portable**

- can be read by **Java** on any machine

Typically used to read and write **sounds** and **images**

Use **InputStream** and **OutputStream** classes and their subclasses (will look at details in a few slides)

Data vs Processing

A **data** ***stream*** represents a particular **source** or **destination**

- ***ex:*** a string in memory or a file on disk

A **processing** ***stream*** (aka a **filtering** ***stream***) ***manipulates*** the data in the stream

- ***ex:*** it may **convert** the data from one format to another
- ***ex:*** it may **buffer** the stream

Just Checking

The output stream connected to the computer screen is:

A. **System.screen**

B. **System.keyboard**

C. **System.in**

D. **System.out**

E. **All of the above**

Just Checking

The stream, automatically available to your Java code, is:

A. **System.out**

B. **System.in**

C. **System.err**

D. **All of the above**

E. **None of the above**

Class Hierarchy for Java.io

- java.lang.Object
  - java.io.File

binary input streams

- java.io.InputStream
  - java.io.FileInputStream
  - java.io.ObjectInputStream
  - java.io.FilterInputStream
  - …

binary output streams

- java.io.OutputStream
  - java.io.FileOutputStream
  - java.io.ObjectOutputStream
  - java.io.FilterOutputStream
  - …

= abstract class

text input streams

- java.io.Reader
  - java.io.BufferedReader
  - java.io.InputStreamReader
    - java.io.FileReader
  - …

text output streams

- java.io.Writer
  - java.io.BufferedWriter
  - java.io.OutputStreamWriter
    - java.io.FileWriter
  - java.io.PrintWriter ←
  - …

Random Access Files

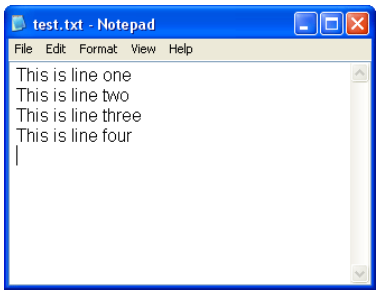
- java.io.RandomAccessFile

Writing Text Files

PrintWriter Class

- Can write strings, int, floats, Objects, *etc.* directly to an output file
  - Using **print()**, **println()**, **printf()** methods
  - Same style as ***System.out.println()***
- **Simple** and **flexible**

Writing Text Files



To write to a file:

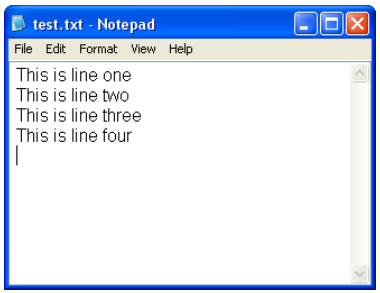
1. Construct a **PrintWriter** object

```
PrintWriter out = new PrintWriter(
        new FileOutputStream("output.txt"));
```

```
PrintWriter out = new PrintWriter("output.txt");
```

- If **file doesn't exist**, an empty file is **created**
- If **file already exists**, it is **emptied** before the new data is written into it

Writing Text Files

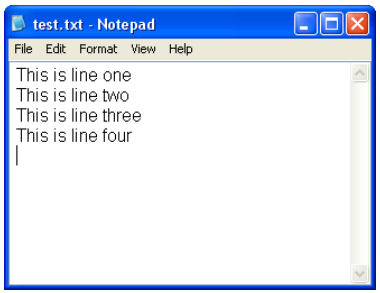


To write to a file:

1. Construct a **PrintWriter** object
   - If you want to **append** to an existing file:

```
PrintWriter out = new PrintWriter(
                      new FileOutputStream(
                          "output.txt", true));
```

Writing Text Files



To write to a file:

2. **Use print(), println() or printf() to write into a PrintWriter:**

```
out.println(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.println("Hello, World!");
```

3. **Close the file** when you are done

```
out.close();
```

Otherwise, **not all the output may be written** to the disk file

Error in the book

Since Java 5, the **PrintWriter** class has a constructor that takes a *file name* as argument.

- So instead of (as the book says)

```
PrintWriter out =
        new PrintWriter(
            new FileOutputStream("output.txt"));
```

- You can do (as the slides say)

```
PrintWriter out = new PrintWriter("output.txt");
```

Output Buffering

Many text output streams (Writer class) are **not buffered**

- each write operation causes characters to be **written immediately** to the stream
- very **inefficient**
- so wrap a **BufferedWriter** around a Writer object that does not buffer (**more efficient**)

```
PrintWriter out = new PrintWriter( new
BufferedWriter(new FileWriter("myFile.out")));
```

Exceptions

Exceptions when writing to a Text File

- When a text file is opened, a ***FileNotFoundException*** can be ***thrown***
  - In this context it means that the **file could not be created**
  - This type of ***exception*** can also be ***thrown*** when a program attempts to open a file for reading and there is **no such file**
- All exceptions from IO are **checked**
  - So, the file should be opened inside a **try** block
  - A **catch** block should catch and handle the possible exception

Example:

TextFileOutputDemo.java with Try/Catch Block

TextFileOutputDemo2.java with no Try/Catch Block but Throws

TextFileOutputDemo.java

```
PrintWriter outputStream = null; //needs to be declared outside of try
try {
        outputStream = new PrintWriter (new BufferedWriter(
                                new PrintWriter("stuff.txt")));
        System.out.println("Writing to file.");
        outputStream.println("The quick brown fox");
        outputStream.println("jumped over the lazy dog.");
}
catch(FileNotFoundException e) {
        System.out.println("Error opening the file stuff.txt.");
        System.exit(0);
}
//finally, must be right after catch
finally {
        if (outputStream != null)
                outputStream.close();
//the close() in PrintWriter does not throw any exception...
}
System.out.println("End of program.");
```

TextFileOutputDemo2.java

```
public static void main(String[] args) throws
                                    FileNotFoundException
{
    PrintWriter outputStream = null;
    outputStream = new PrintWriter (new
      BufferedWriter(new PrintWriter("stuff.txt")));
    System.out.println("Writing to file.");
    outputStream.println("The quick brown fox");
    outputStream.println("jumped over the lazy dog.");
    outputStream.close();
    System.out.println("End of program.");
}
```

Writing to a Text File

- When a program is finished writing to a file, it should **always close the stream** connected to that file

```
outputStreamName.close();
```

- This allows the system to release any resources used to connect the stream to the file
- If the program does not close the file before the program ends, **Java** will close it automatically, but it is safest to close it **explicitly**

Writing to a Text File

- Output streams connected to files are **usually buffered**
  - Rather than physically writing to the file as soon as possible, the data is saved in a **temporary location** (**buffer**)
  - When **enough data accumulates**, or when the method **flush** is invoked, the buffered data is written to the file **all at once**
  - This is **more efficient**, since **physical writes** to a file can be **slow**

Writing to a Text File

- The method **close** invokes the method **flush**, thus ensuring that all the data is written to the file
  - If a program relies on **Java** to close the file, and the program **terminates abnormally**, then any output that was **buffered** **may not** get written to the file
  - Also, if a program writes to a file and later **reopens** it to read from the same file, it **will have to be closed** first anyway
  - The **sooner** a file is closed after writing to it, the **less likely** it is that there will be a problem

throw: What happens?

Use Case	Best Way
Console Output	`new PrintWriter(System.out, true);` Pass *true* to enable auto-flushing when calling println(), printf(), or format().
Writing to a file	`new PrintWriter(new File("file.txt"));` Use try-with-resources (try(...) {}) to ensure the writer is closed automatically.
Large file writing	`new PrintWriter(new BufferedWriter(new FileWriter("file.txt")));` Wrapping PrintWriter with BufferedWriter improves performance for large files.
In memory writing	`new PrintWriter(new StringWriter());` Use *StringWriter* if you want to capture output in-memory instead of writing to a file.

Reading a file with Scanner

- **Scanner** is not in java.io (where is it?)
- **Scanner** can be used for reading from the keyboard as well as reading from a text file
- **Same methods** for reading from the keyboard
  - `nextBoolean(), nextByte(), nextInt(), hasNextLine(), nextFloat(), nextDouble(), next(), nextLine()…`

```
Scanner inFile = new Scanner (
                     new FileReader("input.txt"));
int age = inFile.nextInt();
```

Reading a file with Scanner

Display 10.3 Reading Input from a Text File Using Scanner

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4
5  public class TextFileScannerDemo
6  {
7      public static void main(String[] args)
8      {
9          System.out.println("I will read three numbers and a line");
10         System.out.println("of text from the file morestuff.txt.");
11
12         Scanner inputStream = null;
13
14         try
15         {
16             inputStream =
17                 new Scanner(new FileInputStream("morestuff.txt"));
18         }
```

(continued)

Reading a file with Scanner

Display 10.3 Reading Input from a Text File Using Scanner

```
19          catch(FileNotFoundException e)
20          {
21              System.out.println("File morestuff.txt was not found");
22              System.out.println("or could not be opened.");
23              System.exit(0);
24          }
25              int n1 = inputStream.nextInt( );
26              int n2 = inputStream.nextInt( );
27              int n3 = inputStream.nextInt( );
28
29              inputStream.nextLine(); //To go to the next line
30
31              String line = inputStream.nextLine( );
32
```

(continued)

Reading a file with Scanner

Display 10.3 Reading Input from a Text File Using Scanner

```
33              System.out.println("The three numbers read from the file are:");
34              System.out.println(n1 + ", " + n2 + ", and " +  n3);
35
36              System.out.println("The line read from the file is:");
37              System.out.println(line);
38
39              inputStream.close( );
40          }
41      }
```

File morestuff.txt

```
1 2
3 4
Eat my shorts.
```

This file could have been made with a text editor or by another Java program.

Display 10.3 Reading Input from a Text File Using Scanner

SCREEN OUTPUT

```
I will read three numbers and a line
of text from the file morestuff.txt.
The three numbers read from the file are:
1, 2, and 3
The line read from the file is:
Eat my shorts.
```

Reading a file with Scanner

Display 10.3 **Reading Input from a Text File Using** Scanner

SCREEN OUTPUT

```
I will read three numbers and a line
of text from the file morestuff.txt.
The three numbers read from the file are:
1, 2, and 3
The line read from the file is:
Eat my shorts.
```

Reading a file with Scanner

```
Scanner inFile = new Scanner (
                     new FileReader("input.txt"));
int age = inFile.nextInt();
```

- if you try to read beyond the end of a file, a ***NoSuchElementException*** will be thrown
- can use ***hasNext***... methods to avoid going beyond **EOF**
  - *hasNextLine*(), *hasNextInt*(), …

EOF with hasNextLine

Display 10.4 Checking for the End of a Text File with hasNextLine

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.PrintWriter;
5  import java.io.FileOutputStream;
6
7  public class HasNextLineDemo
8  {
9      public static void main(String[] args)
10     {
11         Scanner inputStream = null;
12         PrintWriter outputStream = null;
```

(continued)

EOF with hasNextLine

Display 10.4 Checking for the End of a Text File with hasNextLine

```
13          try
14          {
15             inputStream =
16                new Scanner(new FileInputStream("original.txt"));
17             outputStream = new PrintWriter(
18                              new FileOutputStream("numbered.txt"));
19          }
20          catch(FileNotFoundException e)
21          {
22             System.out.println("Problem opening files.");
23             System.exit(0);
24          }

25          String line = null;
26          int count = 0;
```

(continued)

EOF with hasNextLine

Display 10.4 Checking for the End of a Text File with hasNextLine

```
27          while (inputStream.hasNextLine( ))
28          {
29              line = inputStream.nextLine( );
30              count++;
31              outputStream.println(count + " " + line);
32          }

33          inputStream.close( );
34          outputStream.close( );
35      }

36  }
```

(continued)

EOF with hasNextLine

Display 10.4 **Checking for the End of a Text File with hasNextLine**

File original.txt

```
Little Miss Muffet
sat on a tuffet
eating her curves away.
Along came a spider
who sat down beside her
and said "Will you marry me?"
```

File numbered.txt (after the program is run)

```
1 Little Miss Muffet
2 sat on a tuffet
3 eating her curves away.
4 Along came a spider
5 who sat down beside her
6 and said "Will you marry me?"
```

Line Numberer

Read all lines of a file and send them to an output file, preceded by line numbers

input file:

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

output file:

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

LineNumberer.java

EOF with hasNextInt

Display 10.5 Checking for the End of a Text File with hasNextInt

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;

4  public class HasNextIntDemo
5  {
6      public static void main(String[] args)
7      {
8          Scanner inputStream = null;

9          try
10         {
11            inputStream =
12               new Scanner(new FileInputStream("data.txt"));
13         }
14         catch(FileNotFoundException e)
15         {
16            System.out.println("File data.txt was not found");
17            System.out.println("or could not be opened.");
18            System.exit(0);
19         }
```

(continued)

EOF with hasNextInt

Display 10.5 Checking for the End of a Text File with hasNextInt

```
20          int next, sum = 0;
21          while (inputStream.hasNextInt( ))
22          {
23              next = inputStream.nextInt( );
24              sum = sum + next;
25          }

26          inputStream.close( );

27          System.out.println("The sum of the numbers is " + sum);
28      }
29  }
```

File data.txt

```
1  2
3  4 hi 5
```

Reading ends when either the end of the file is reach or a token that is not an `int` is reached. So, the 5 is never read.

SCREEN OUTPUT

```
The sum of the numbers is 10
```

Just Checking

In Java, when you open a text file you should account for a possible:

A. **FileNotFoundException**

B. **FileFullException**

C. **FileNotReadyException**

D. **All of the above**

E. **None of the above**

Just Checking

The **scanner** class has a series of methods that checks to see if there is any more **well-formed input** of the appropriate type. These methods are called _____ methods:

A. **nextToken**

B. **hasNext**

C. **getNext**

D. **testNext**

E. **None of the above**

Class Hierarchy for Java.io

- java.lang.Object
  - java.io.File ←

binary input streams

- java.io.InputStream
  - java.io.FileInputStream
  - java.io.ObjectInputStream
  - java.io.FilterInputStream
  - …

binary output streams

- java.io.OutputStream
  - java.io.FileOutputStream
  - java.io.ObjectOutputStream
  - java.io.FilterOutputStream
  - …

text input streams

- java.io.Reader
  - java.io.BufferedReader
  - java.io.InputStreamReader
    - java.io.FileReader
  - …

text output streams

- java.io.Writer
  - java.io.BufferedWriter
  - java.io.OutputStreamWriter
    - java.io.FileWriter
  - java.io.PrintWriter
  - …

Random Access Files

- java.io.RandomAccessFile

The File Class

- **not** really an **I/O stream**
- contains methods to check the properties of a file
  - ***ex:*** check if a file with a specific name exists, if a file can be written into, …
- constructor takes a filename (or directory name or URL) as argument
- **useful methods:**

boolean canRead() **boolean canWrite()**
boolean createNewFile() **boolean delete()**
boolean exists() **boolean isDirectory()**
File[] listFiles() **boolean mkdir()**
boolean renameTo(File dest) …

Example: FileInfo.java

The NIO File Class

- **Modern** file system **interface**
- Part of Java **NIO** (New I/O) **package**, introduced in Java 7
- Provides a **more flexible and efficient** way to handle file and directory operations
- Designed to be **more powerful** than java.io.File, with better performance, especially for larger files or directories
- **useful classes:**

  Path: Represents a file or directory **path**.

  Files: Provides static methods for file and directory operations.

  FileSystems: Provides access to file system and its properties.

  Paths: Utility class to convert between String and Path objects.

Example: FileInfoNIO.java

The NIO File Class

Key Advantages of java.nio.file (NIO) Over java.io.File:

- **Path-based API:** Path objects provide a more flexible, **modern way** to handle file paths, without dealing with strings or file separators manually.
- **Better performance:** NIO supports **non-blocking** and **asynchronous** I/O, which provides better performance for large files or directories.
- **More efficient:** It supports **memory-mapped files** and **lazy reading**, so you can handle large files more efficiently without reading them all into memory.
- **Advanced file operations:** NIO includes built-in support for symbolic links, file attributes, and better directory traversal methods (e.g., Files.walk()).
- **Cross-platform:** NIO **abstracts** away **OS-specific details** like path separators, making it easier to write cross-platform code.

Class Hierarchy for Java.io

- java.lang.Object
  - java.io.File

binary input streams

- java.io.InputStream
  - java.io.FileInputStream
  - java.io.ObjectInputStream
  - java.io.FilterInputStream
  - …

binary output streams

- java.io.OutputStream
  - java.io.FileOutputStream
  - java.io.ObjectOutputStream
  - java.io.FilterOutputStream
  - …

text input streams

- java.io.Reader
  - java.io.BufferedReader ←
  - java.io.InputStreamReader
    - java.io.FileReader ←
  - …

text output streams

- java.io.Writer
  - java.io.BufferedWriter
  - java.io.OutputStreamWriter
    - java.io.FileWriter
  - java.io.PrintWriter
  - …

Random Access Files

- java.io.RandomAccessFile

Reading a Text File

Before Java 5, we used the class **BufferedReader** to read a text file

- only 2 methods to read input: **read**() and **readLine**()
- **read**():
  - ✓ reads a single character,
  - ✓ returns the next character as an **int** or **-1 at EOF**

```
BufferedReader in = new BufferedReader(
                            new FileReader("in.txt"));
char c; int next = in.read();
while (next != -1) {
      c = (char) next;
      next = in.read(); }
```

BufferedReadTest.java

Reading a Text File

- **readLine**():
  - ✓ reads a **line** of character and returns a String
  - ✓ returns **null** when it tries to **read beyond the EOF**
- Unlike **Scanner**, **BufferedReader** has no methods to read a number from a text file
  - ✓ So … you read a string, then converted it to a number

```
BufferedReader in = new BufferedReader(
                        new FileReader("input.txt"));
…
String line = in.readLine();
int age = Integer.parseInt(line);
```

Reading a Text File

BufferedReader (Still used, but with modern alternatives)

- **BufferedReader** is still widely used for large files, where **performance is a priority**.
- However, in modern code, **BufferedReader** is often used in combination with other modern classes or APIs (like **Files.lines**() or **Paths**).

```
public static void main(String[] args) throws IOException {
    Path path = Paths.get("example.txt");
    try (BufferedReader reader = Files.newBufferedReader(path))
    { // BufferedReader using Files
        String line;
        while ((line = reader.readLine()) != null) {
                System.out.println(line);
        }
    }
}
```

Class Hierarchy for Java.io

- java.lang.Object
  - java.io.File

binary input streams

- java.io.InputStream
  - java.io.FileInputStream
  - java.io.ObjectInputStream
  - java.io.FilterInputStream
  - …

binary output streams

- java.io.OutputStream
  - java.io.FileOutputStream
  - java.io.ObjectOutputStream
  - java.io.FilterOutputStream
  - …

text input streams

- java.io.Reader
  - java.io.BufferedReader
  - java.io.InputStreamReader
    - java.io.FileReader
  - …

text output streams

- java.io.Writer
  - java.io.BufferedWriter
  - java.io.OutputStreamWriter
    - java.io.FileWriter
  - java.io.PrintWriter
  - …

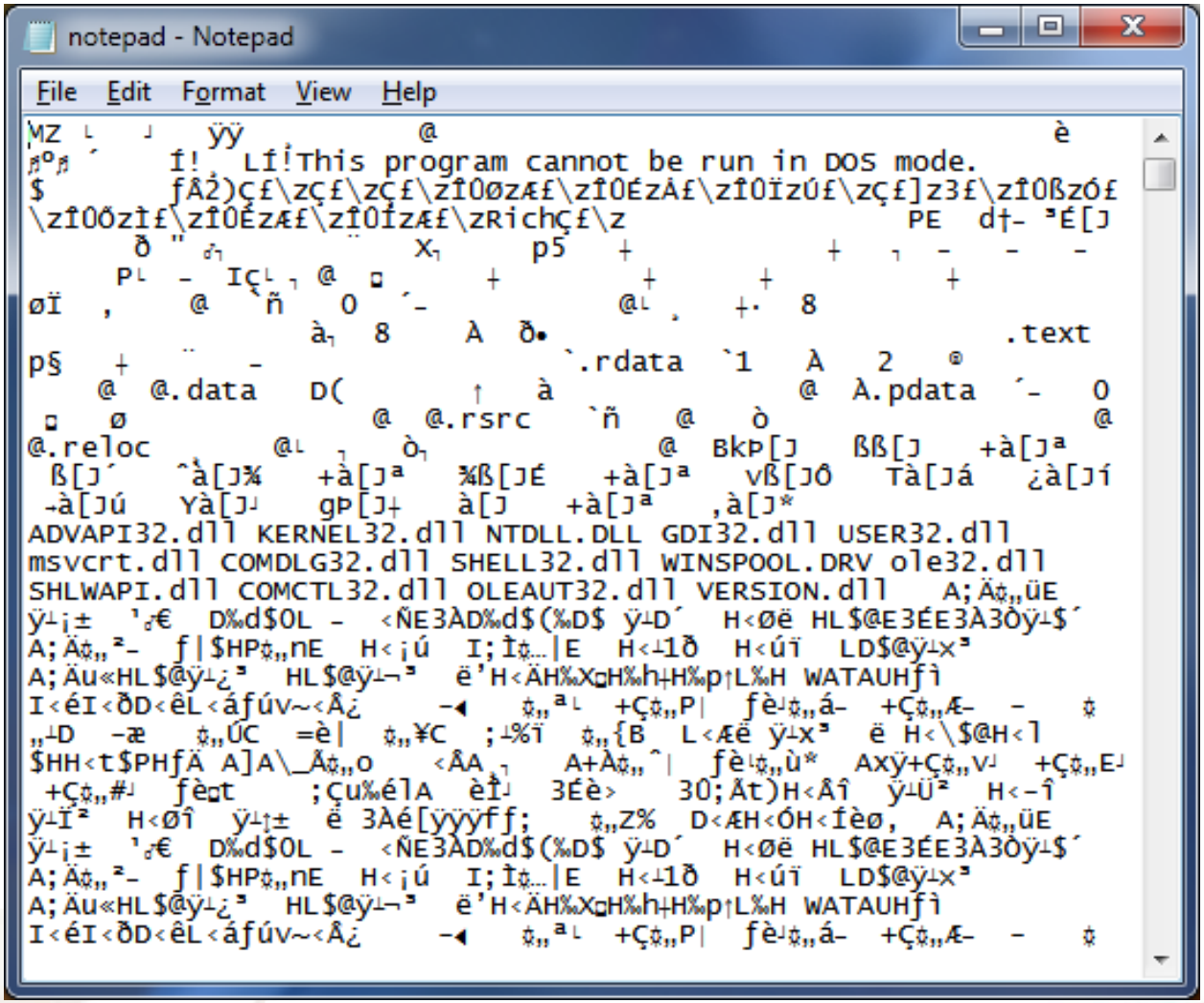
Random Access Files

- java.io.RandomAccessFile

Binary Files

- **Binary** files store data in the same format used by computer memory to store the values of variables
  - **No conversion** needs to be performed when a value is stored or retrieved from a binary file
- **Java binary** files, unlike other binary language files, are **portable**
  - A binary file created by a Java program can be moved from one computer to another
  - These files can then be read by a Java program, but **only by a Java program**

Binary Files



Read/Write Binary Files

- Use the classes **ObjectInputStream** and **ObjectOutputStream**
- Can read/write strings, int, floats, Objects, *etc.* directly to binary files
  - to read: **readInt**(), **readDouble**(), **readChar**(), **readBoolean**() and **readUTF**() (to read strings)
  - to write: **writeInt**(), **writeDouble**(), **writeChar**(), **writeBoolean**() and **writeUTF**() (to write strings)

Examples

```
ObjectOutputStream out = new ObjectOutputStream(
                         new FileOutputStream("myFile.dat"));
out.writeInt(…);
out.writeDouble(…);
out.writeChar(…);
…
```

```
ObjectInputStream in = new ObjectInputStream(
                           new FileInputStream("myFile.dat"));
int x = in.readInt();
double y = in.readDouble();
String s = in.readUTF();
…
```

Examples

```
ObjectOutputStream outputStream = null;
try {
   outputStream = new ObjectOutputStream(new FileOutputStream("numbers2.dat"));
   int n;
   …
   do {
         n = keyboard.nextInt();
         outputStream.writeInt(n);
   } while (n >= 0);
} // end of try

catch(IOException e) {
   System.out.println("Problem with output to file numbers2.dat.");
}
finally {
   try { if (outputStream != null)
                  outputStream.close( ); }
   catch(IOException e) {
         System.out.println("Can't seem to close the file...");}
} // end of finally
```

BinaryOutputDemo.java

Checking Binary EOF

- All **ObjectInputStream** methods that read from a binary file throw an ***EOFException*** when trying to read beyond the **end of a file**
  - This can be used to **end a loop** that reads all the data in a file
- Note that **different** file-reading methods check for the **end of a file** in **different ways**
  - Testing for the end of a file in the wrong way can cause a program to go into an **infinite loop** or **terminate abnormally**

Object Serialization

"Like the **Transporter** on **Star Trek**, it's all about taking **something** **complicated** and turning it into a **flat sequence** of 1s and 0s, then taking that sequence of 1s and 0s (possibly at another **place**, possibly at another **time**) and **reconstructing** the original complicated "**something**."

Object Serialization

- Java allows writing the object's **current state** to disk
- Just values of instance variables, **not methods**
- Also called ***object streams***
- Once **serialized**, the objects can be read again into another program
- **Huge advantage** over writing an object in text format by hand…
  - you **don't have to break up the object** yourself into numbers, strings, imbedded objects, … to read and write it
  - you just read/write an **entire object** (even an entire array of objects) in one shot

Object Serialization

- The idea that an object can "live" beyond execution of the program that created it is called **persistence**
- Once **serialized**, the objects can be read again into another program
- Objects are saved in **binary** format,
  - so you use the **ObjectInputStream** / **ObjectOutputStream** classes
  - not the **Reader** / **Writer** classes

Object Serialization

- It must implement the **Serializable** **interface**
  - more on interfaces **later**…
- Use the **ObjectOutputStream** and **ObjectInputStream** classes
- Use the methods:
  - **writeObject**() to serialize an object
  - **readObject**() to deserialize an object
    - ✓ you need to use a cast to convert an object to the apt type

Object Serialization

Example:

- To Serialize

```
BankAccount b = …;
ObjectOutputStream out = new ObjectOutputStream(
                         new FileOutputStream("bank.dat"));
out.writeObject(b);
```

- To Deserialize

```
ObjectInputStream in = new ObjectInputStream(
                       new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```

readObject() can throw a (**checked**) **ClassNotFoundException** so you must catch it or declare it

readObject() returns an **Object** reference, so you must cast the result

Object Serialization

- If the class has instance variables that are references to classes (**composition**)
  - these objects **will be serialized** too
  - so they **must** also **implement** the **Serializable** interface
- Many classes from the Java class library implement **Serializable** , including the ***String*** class
- Why **don't all** classes implement Serializable ????
- An **entire array** can be **serialized** (written/read in **one operation**)

`transient ?`

Object Serialization

Example: Bank Accounts

- Serialization of a Bank object (**array** of BankAccount objects)
- If a file with **serialized** data **exists** (in bank.dat), then it is **loaded**
- Otherwise, the program **starts with a new Bank object**.
- **BankAccount** objects are added to **Bank**.
- Then the **Bank object** is **saved**.

Examples

```
import java.io.*;

public class SerialTester {
     public static void main(String[] args) throws IOException,
                                                    ClassNotFoundException{
          Bank myBank;
          File f = new File("bank.dat");
          if (f.exists()) {
                    ObjectInputStream in = new ObjectInputStream(
                                                  new FileInputStream(f));
                    myBank = (Bank) in.readObject();
                    in.close();
          }
          else {
                    myBank = new Bank();
                    myBank.addAccount(new BankAccount(1001, 20000));
                    myBank.addAccount(new BankAccount(1015, 10000));
          }
```

Serialtester.java

Examples

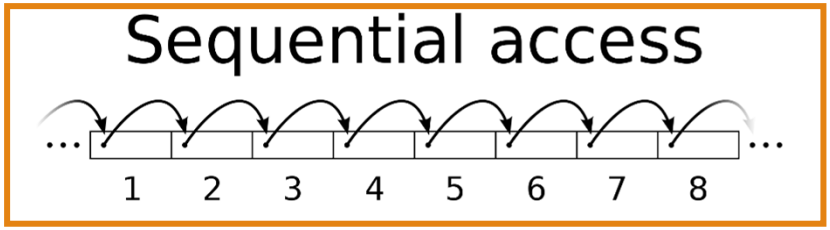
```
        // Deposit some money
        BankAccount a = myBank.find(1001);
        a.deposit(100);
        System.out.println(a.getAccountNumber() + ":" + a.getBalance());
        a = myBank.find(1015);
        System.out.println(a.getAccountNumber() + ":" + a.getBalance());
        ObjectOutputStream out = new ObjectOutputStream(
                                    new FileOutputStream(f));
        out.writeObject(myBank);
        out.close();
        }
}
```

- **See also: Bank.java**

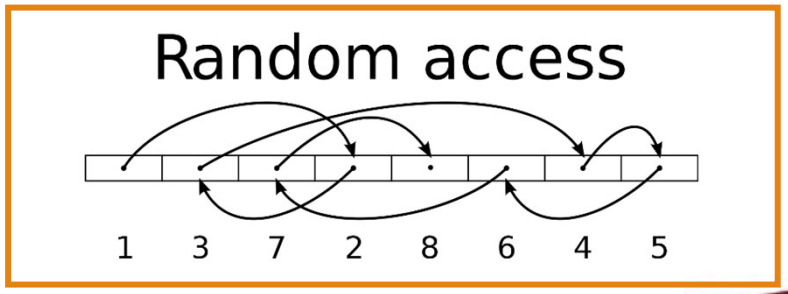
Serialtester.java

Sequential vs Random

- **Sequential** access
  - A file is processed **one byte at a time**
  - It can be inefficient

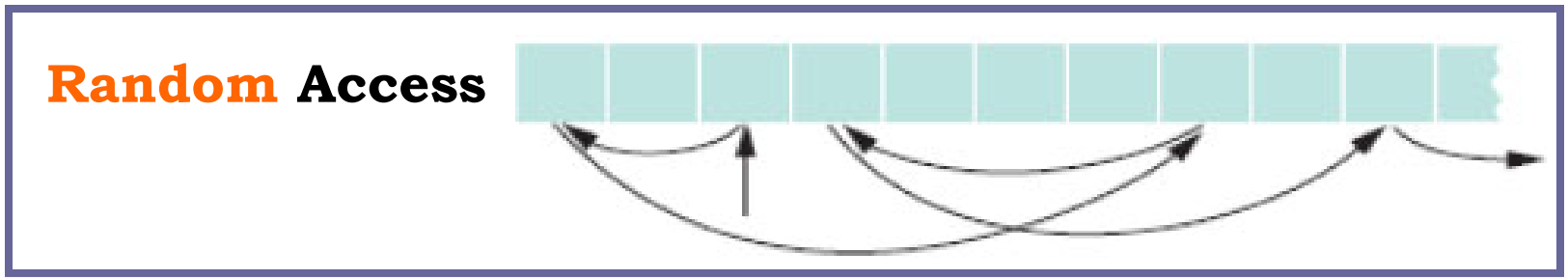


- **Random** access
  - Allows access at **arbitrary locations** in the file
  - **Only binary** disk files support random access
    - ✓ **System.in** and **System.out** do not



Random Acces

- Each file has a special **file pointer** position
  - You can read/write at the position where the pointer is



Class Hierarchy for Java.io

- java.lang.Object
  - java.io.File

binary input streams

- java.io.InputStream
  - java.io.FileInputStream
  - java.io.ObjectInputStream
  - java.io.FilterInputStream
  - …

binary output streams

- java.io.OutputStream
  - java.io.FileOutputStream
  - java.io.ObjectOutputStream
  - java.io.FilterOutputStream
  - …

text input streams

- java.io.Reader
  - java.io.BufferedReader
  - java.io.InputStreamReader
    - java.io.FileReader
  - …

text output streams

- java.io.Writer
  - java.io.BufferedWriter
  - java.io.OutputStreamWriter
    - java.io.FileWriter
  - java.io.PrintWriter
  - …

Random Access Files

- java.io.RandomAccessFile ←

RandomAccesFile Class

- You can **open a file** either for
  - Reading only ("**r**")
  - Reading and writing ("**rw**")

```
RandomAccessFile f = new RandomAcessFile("bank.dat","rw");
```

- Writes **binary** data
- Read/Write **primitives**
  - `writeDouble()`
  - `writeInt()`
  - `writeChar()`
  - …
  - `readDouble()`
  - `readInt()`
  - `readChar()`
  - …

RandomAccesFile Class

- To move the file pointer to a specific byte

```
f.seek(n);
```

- Need to know **size of primitives**
- Other languages (C) requires a **sizeof**() operator
- Java uses **standardized** sizes
  - *Byte*.SIZE = 1;
  - *Character*.SIZE = 2;
  - *Short*.SIZE = 2;
  - *Integer*.SIZE = 4;
  - *Long*.SIZE = 8;
  - *Float*.SIZE = 4;
  - *Double*.SIZE = 8;

Can use these values to calculate **offsets** into the file

Example

Save a database in a file:

- ex: age - gender – salary

```
100 43 'M' 19.55
110 83 'F' 85.60
120 25 'M' 143.45
130 37 'M' 29.99
140 11 'F' 5.50
```

access a specific record to change it

Example

```
import java.io.*;
class EmployeeRecord {
    static final int RECORD_SIZE = Integer.SIZE + Byte.SIZE +
                                   Character.SIZE + Double.SIZE;
    static final int ID_OFFSET = 0;
    static final int AGE_OFFSET = Integer.SIZE;
    static final int GENDER_OFFSET = Integer.SIZE + Byte.SIZE;
    static final int SALARY_OFFSET = Integer.SIZE + Byte.SIZE +
                                     Character.SIZE;
    static int currentID;
    final int ID;
    byte age;  char gender;    double salary;

 public EmployeeRecord(int ID, byte age, char gender, double salary){
    …
 }
 …
}
```

[RandomAccess.java]

Example

```
public class RandomAccess {
static int DBSIZE = 10;

   public static void main ( String[] aArguments ) throws IOException
   {
        String dbFile = "db.dat";
        EmployeeRecord[] dataBase = buildDataBase();

        // store the database to disk
        RandomAccessFile rf = new RandomAccessFile(dbFile, "rw");
        for(int i = 0; i < DBSIZE; i++){
                rf.writeInt(dataBase[i].getID());
                rf.writeByte(dataBase[i].getAge());
                rf.writeChar(dataBase[i].getGender());
                rf.writeDouble(dataBase[i].getSalary());
        }
        rf.close();
```

RandomAccess.java

Example

```
// update the database and store to disk
rf = new RandomAccessFile(dbFile, "rw");
System.out.println("\nChanging Salary of employee #3 to" +
                                                    " 88.88...");
rf.seek((3 * EmployeeRecord.RECORD_SIZE) +
                                        EmployeeRecord.SALARY_OFFSET);
rf.writeDouble(88.88);
rf.close();
```

RandomAccess.java

Example

```
        // read the new database into an employee array
        rf = new RandomAccessFile(dbFile, "r");
        for (int i = 0; i < DBSIZE; i++){
            EmployeeRecord emp = new EmployeeRecord(rf.readInt(),
                                                    rf.readByte(),
                                                    rf.readChar(),
                                                    rf.readDouble());
            dataBase[i] = emp;
        }
        rf.close();
}
```

RandomAccess.java

Example

```
    static EmployeeRecord[] buildDataBase(){
        EmployeeRecord[] dataBase = new EmployeeRecord[DBSIZE];

        dataBase[0] = new EmployeeRecord(100, (byte)43, 'M', 19.55);
        dataBase[1] = new EmployeeRecord(110, (byte)83, 'F', 85.60);
        …
        dataBase[9] = new EmployeeRecord(190, (byte)16, 'M', 13.56);

        return dataBase;
    }
}
```

RandomAccess.java