CSE 1325: Object-Oriented Programming

Lecture 11

File Input / Output

Mr. George F. Rice
george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

A plateau is a high form of flattery.

This work is licensed under a Creative Commons Attribution 4.0 International License.

Overview

- Java file I/O
  - Filenames and formats
  - Text vs binary
  - File classes
    - Reading and writing
    - Try-with-resources
- Encapsulated save / open
  - Simple classes and primitives
  - Enums
  - Lists and Maps
  - Library classes
  - Subclasses

Finding a Password in Memory by carlosalbertotelxelra per the Pixabay License
https://pixabay.com/photos/password-security-dump-memory-704252/

File Input / Output

- In addition to interacting with the *user*, your program must interact with the *file system*
  - Load data for analysis
  - Save and restore data created by the program or entered by the user
  - Modify existing files
- Java abstracts the file system in a way that works with major operating systems' options

All About Files

- A file is just a named sequence of bytes
- The file has a **name**
  - The name includes a storage device (optional), path, and basename
  - Storage device (such as C:) is needed for Windows but NOT Linux or Mac / Unix – the latter have a “unified file system”
- The file has a **data format**
  - The format (or schema) defines how the data is organized
  - May be formal (XML or JSON schema file) or a document
  - Not always a given, for example, many early Microsoft Office file formats are still undocumented
- Practically, you must know the name and data format to use a file!

File – A self-contained named sequence of bytes available via the operating system

Filename Examples

- The first are Windows, then Mac, then Linux
  - `C:\Program Files\WindowsApps`
    `/Applications`
    `/usr/bin`
    Applications
  - `C:\Users\George F. Rice\Documents`
    `/Users/ricegf`
    `/home/ricegf`
    User Documents
  - `D:\`
    `/Volumes/flash`
    `/media/ricegf/flash`
    Removable Media

Typical for Windows
...Mac
...Linux

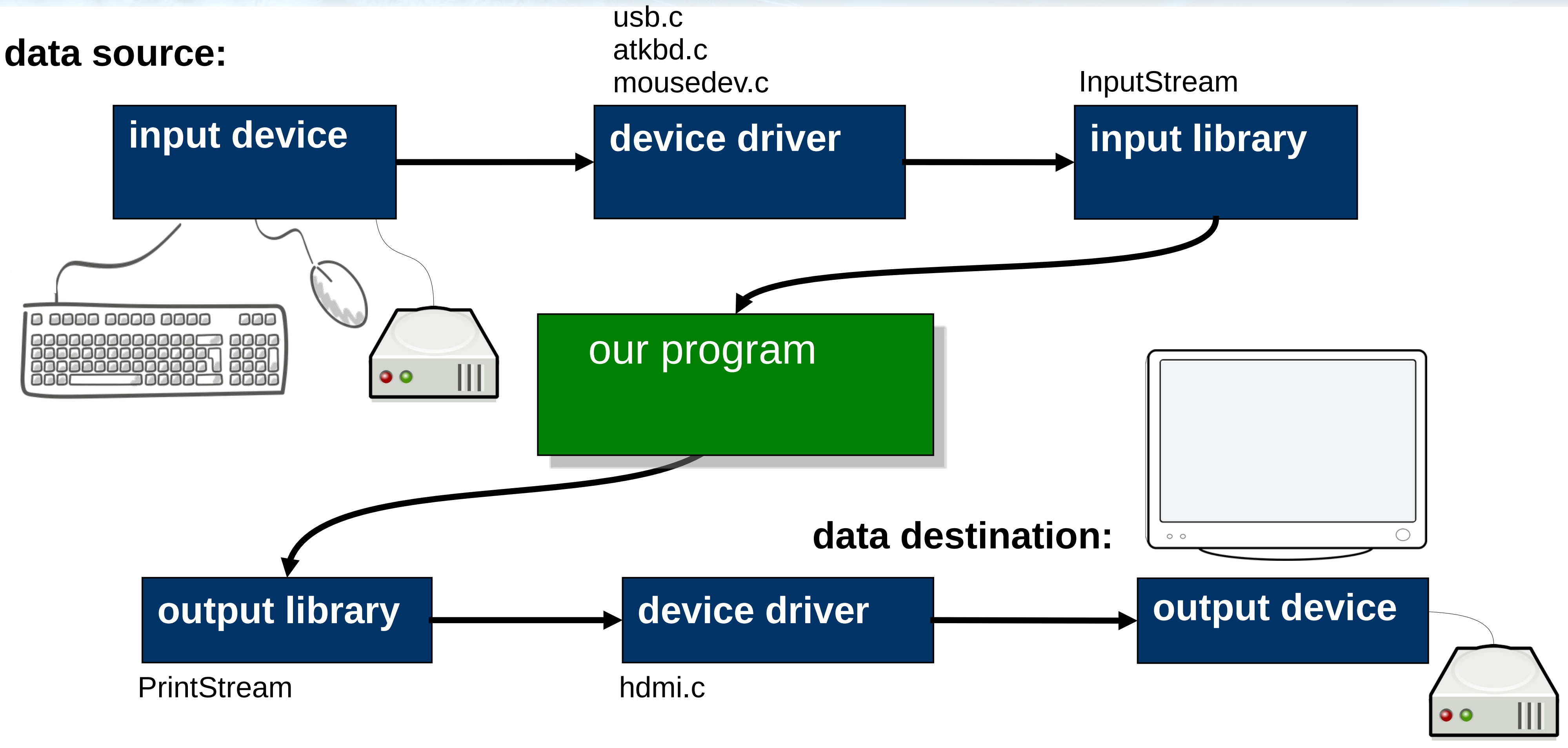
Data Format Example: XML

- How you handle streams determine the program flow
- Example: Handling an XML file
  - DOM*: The XML stream's tags are interpreted to build a complete Document Object Model structure containing all the data in memory
  - SAX*: The XML stream's tags trigger events – calls to your program's methods to handle each tag
- Both approaches are valid and will parse the XML file, but the code looks *very* different

```
<note>
<to>A. Student</to>
<from>Prof Rice</from>
<heading>Reminder</heading>
<body>Exam #3 is Apr 24.</body>
</note>
```

* Document Object Model (DOM) versus Simple API for XML (SAX)
API = Application Programming Interface, XML = eXtended Markup Language

Input and Output are Byte Streams

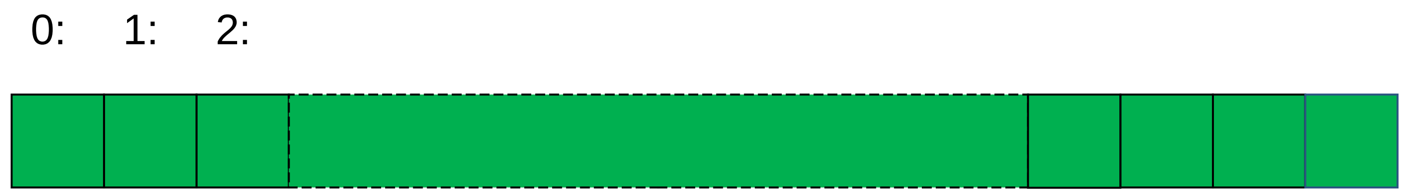


Source Buffer Process Sink

Text is *buffered* during
Input and output!

Keyboard and mouse by Schmidsi per the Pixabay License, attribution not required
https://pixabay.com/vectors/mouse-electronics-keyboard-computer-1412284/
Disk and Monitor by Clker-Free-Vector-Images per the Pixabay License
https://pixabay.com/vectors/monitor-display-computer-screen-309590/
https://pixabay.com/vectors/drive-disk-disc-external-hard-37120/
Drivers from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/

A File



- At the fundamental level, a file is a named sequence of bytes numbered from 0 upwards
- More detail is inferred by programs that interpret a “file format”, e.g., the 7 (or more) bytes "123.456" may mean
  - the floating-point number 123 point 456
  - Or maybe, 123 thousand 456 (and 123,456 is 123 point 456)
  - Or maybe, a grid coordinate where x is 123 and y is 456
  - Or 0x003132322E343536 (1.3847465e+16)

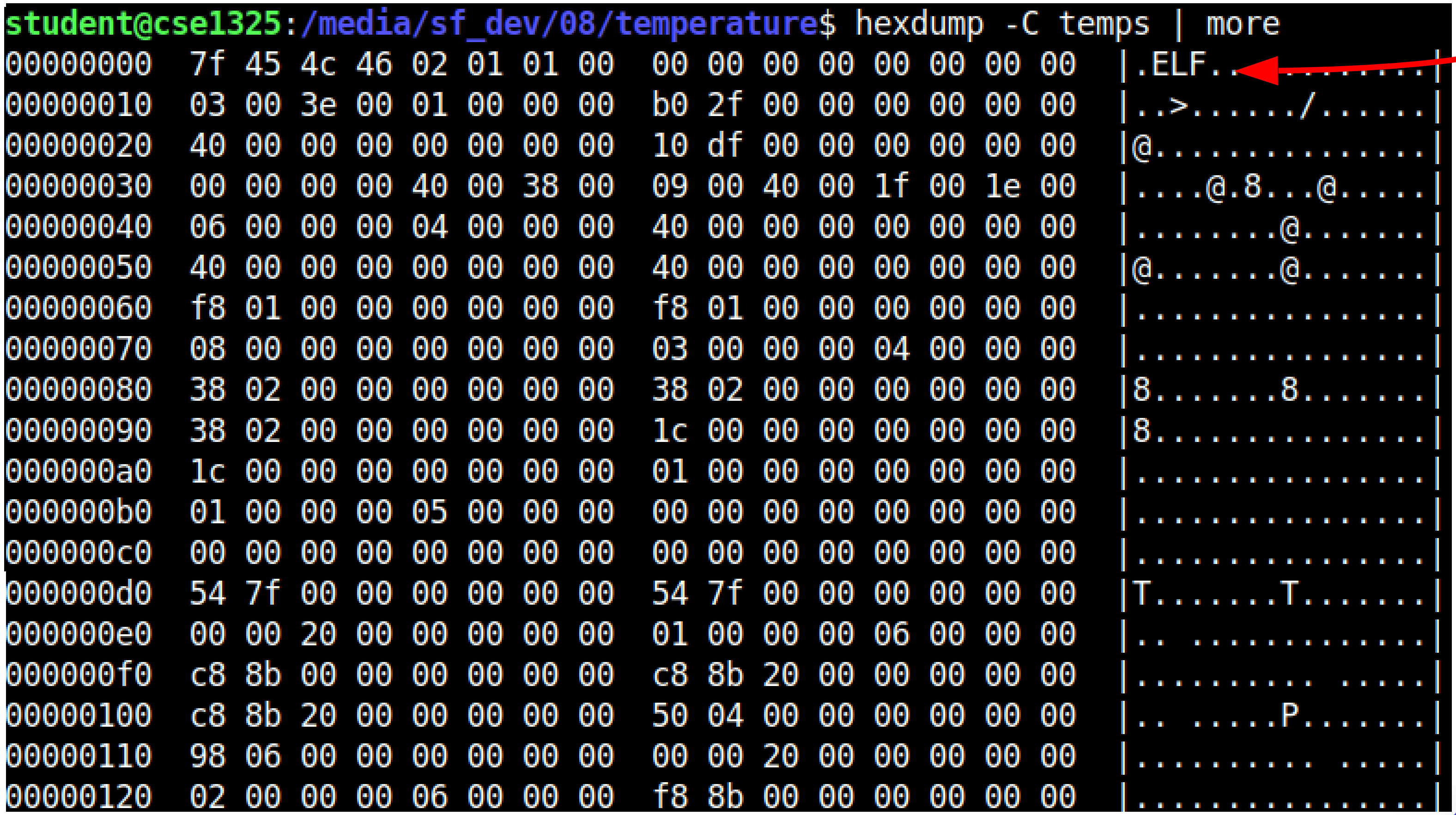
You can examine binary files

In Linux and Mac OS X, “hexdump -C [file]” will reveal the contents of a binary file
Windows requires a hex editor app or a Powershell script – GIYF

Executable Linkable Format (ELF)

Linux Binary Executable i.e., output of g++

"Magic Cookie" → 0x7f454c46, or .ELF, signals a Linux loadable binary file



```
student@cse1325:/media/sf_dev/08/temperature$ hexdump -C temps | more
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  b0 2f 00 00 00 00 00 00  |..>....../......|
00000020  40 00 00 00 00 00 00 00  10 df 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1f 00 1e 00  |....@.8...@.....|
00000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |@.......@.......|
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 00 00  |................|
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00  |................|
00000080  38 02 00 00 00 00 00 00  38 02 00 00 00 00 00 00  |8.......8.......|
00000090  38 02 00 00 00 00 00 00  1c 00 00 00 00 00 00 00  |8...............|
000000a0  1c 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
000000b0  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000d0  54 7f 00 00 00 00 00 00  54 7f 00 00 00 00 00 00  |T.......T.......|
000000e0  00 00 20 00 00 00 00 00  01 00 00 00 06 00 00 00  |.. .............|
000000f0  c8 8b 00 00 00 00 00 00  c8 8b 20 00 00 00 00 00  |.......... .....|
00000100  c8 8b 20 00 00 00 00 00  50 04 00 00 00 00 00 00  |.. .....P.......|
00000110  98 06 00 00 00 00 00 00  00 00 20 00 00 00 00 00  |.......... .....|
00000120  02 00 00 00 06 00 00 00  f8 8b 00 00 00 00 00 00  |................|
```

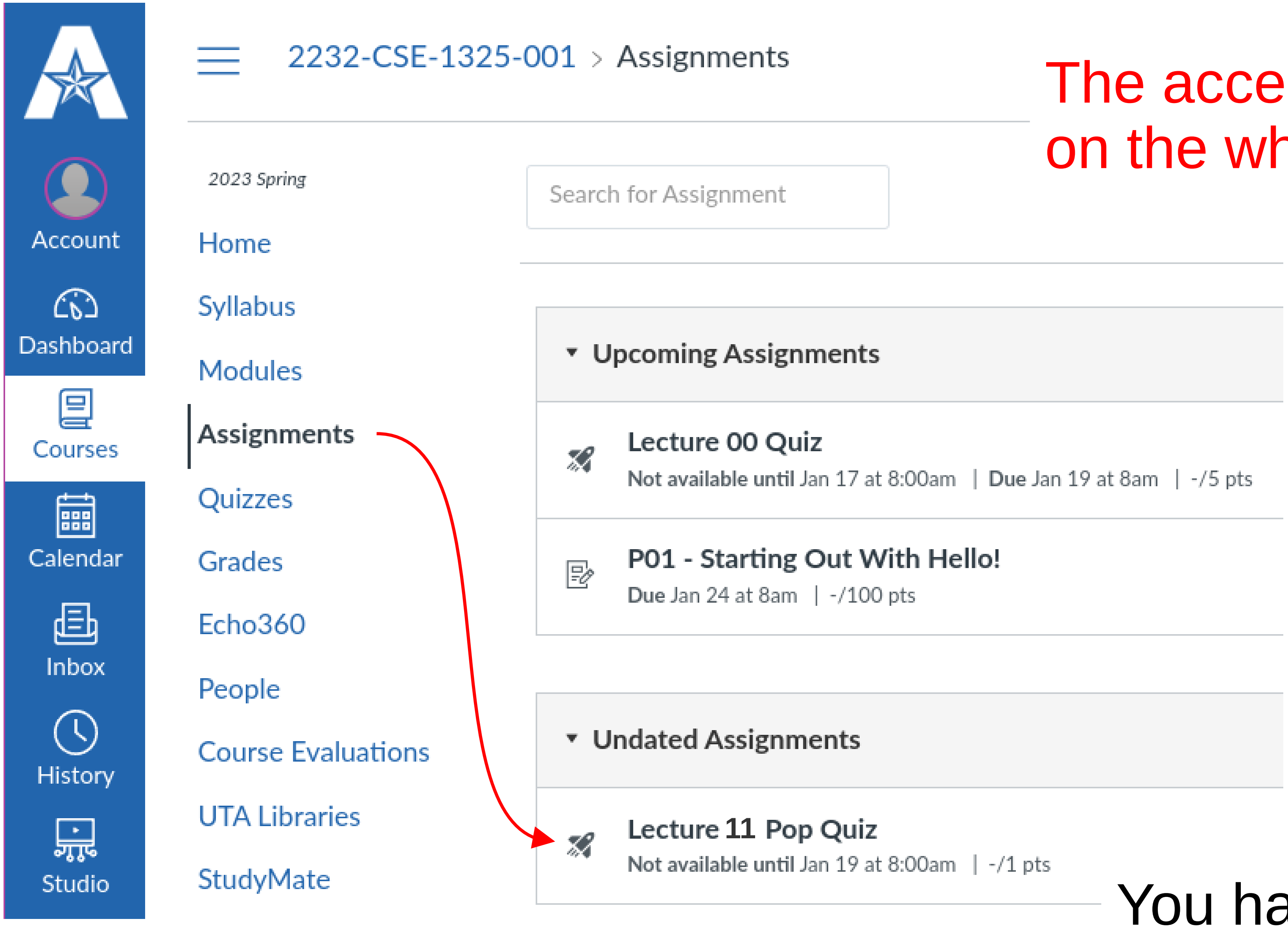
You can examine binary files

In Linux and Mac OS X, "hexdump -C [file]" will reveal the contents of a binary file
Windows requires a hex editor app or a Powershell script – GIYF

Text vs Binary Files

- Use text whenever possible
  - You can read it (without a fancy program)
  - You can debug your programs more easily
  - Text is portable across different systems
  - Size (compressed) is typically comparable
  - Most information can be represented reasonably as text
- Use binary when you must
  - For example, image, sound, and video files for faster decoding
  - Compressed and / or encrypted files
- We won’t cover binary files in class
  - Additional information on binary files follows the break
    for interested students only

Pop Quiz (in Canvas)



2232-CSE-1325-001 > Assignments

2023 Spring

▾ Upcoming Assignments

Lecture 00 Quiz
Not available until Jan 17 at 8:00am | Due Jan 19 at 8am | -/5 pts

P01 - Starting Out With Hello!
Due Jan 24 at 8am | -/100 pts

▾ Undated Assignments

Lecture 11 Pop Quiz
Not available until Jan 19 at 8:00am | -/1 pts

The access code is on the whiteboard.

You have 1 minute.

java.io vs java.nio

- Java offers both classic I/O (**java.io**, **stream**-oriented) & non-blocking I/O (**java.nio**, **buffer**-oriented) libraries
- **java.io** blocks (stops) until your data has been read or written, and then allows you to continue
- **java.nio** returns *while data is being read or written*
  - You must periodically check to see if your input data has arrived so that you can process it
  - You must periodically check to see if your output data has been written so you can reuse the buffer
- As a result, java.nio is harder – let's do java.io!

java.io means ***streams***

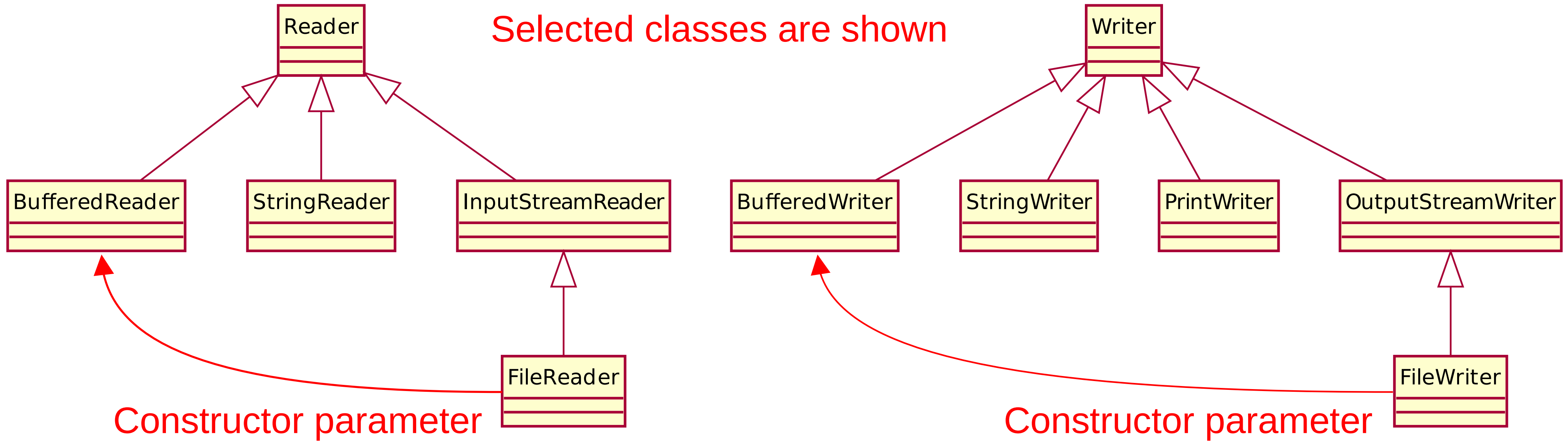
java.io Text Stream Classes

- Java has numerous stream classes, including*
  - Base Class: **Writer** and **Reader**
  - Buffered: **BufferedWriter** and **BufferedReader**
  - Strings: **StringWriter** and **StringReader**
  - Formatted: **PrintWriter**
  - File: **FileWriter** and **FileReader**
- Each pair are classes instanced to create a stream
  - We usually use BufferedReader and BufferedWriter for text
  - Buffering temporarily stores data in memory to deal effectively with speed mismatches and to improve efficiency

* You may notice a pattern…

Opening a Text File in Java

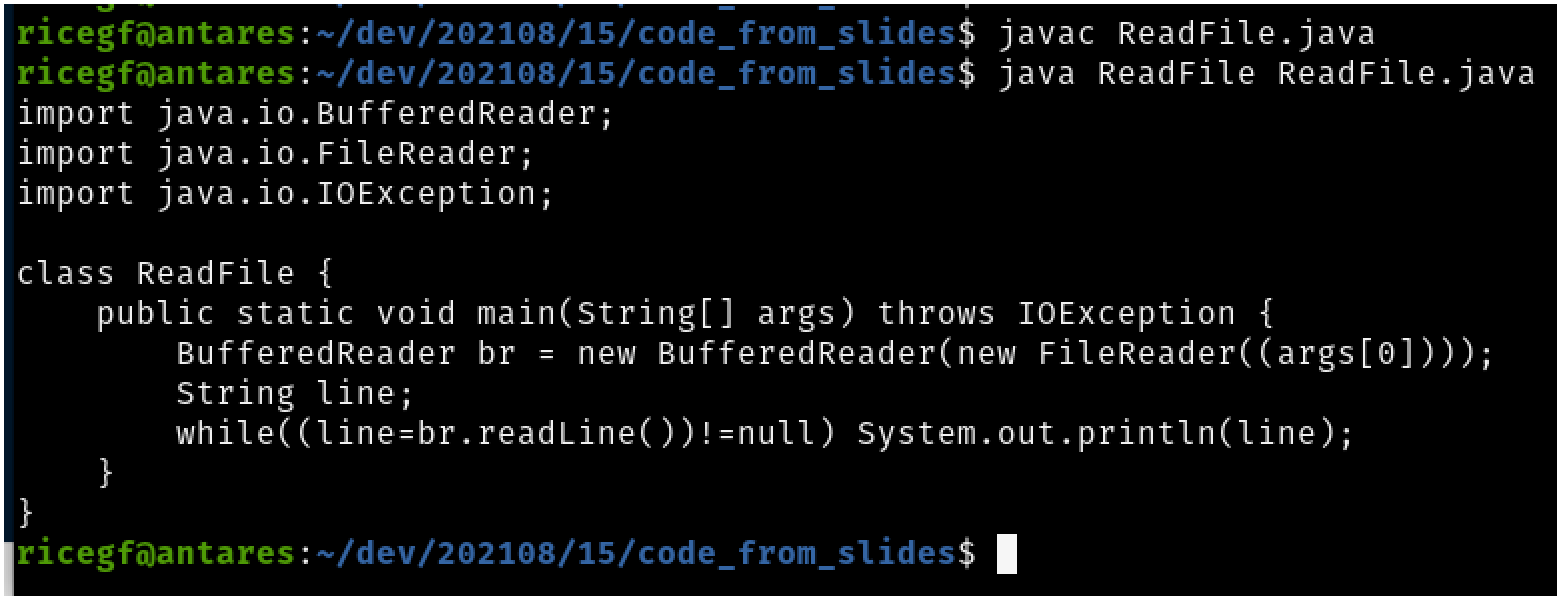
- Instance a FileReader / FileWriter to open a file
  - Instancing one is equivalent to C’s fopen
- Pass that object to a BufferedReader / BufferedWriter constructor to buffer the I/O stream (recommended!)



Opening a File for Reading

Instancing a FileReader opens the filename for reading.

BufferedReader buffers the stream.



```
ricegf@antares:~/dev/202108/15/code_from_slides$ javac ReadFile.java
ricegf@antares:~/dev/202108/15/code_from_slides$ java ReadFile ReadFile.java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class ReadFile {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader((args[0])));
        String line;
        while((line=br.readLine())!=null) System.out.println(line);
    }
}
ricegf@antares:~/dev/202108/15/code_from_slides$ 
```

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class ReadFile {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader((args[0])));
        String line;
        while((line=br.readLine())!=null) System.out.println(line);
    }
}
```

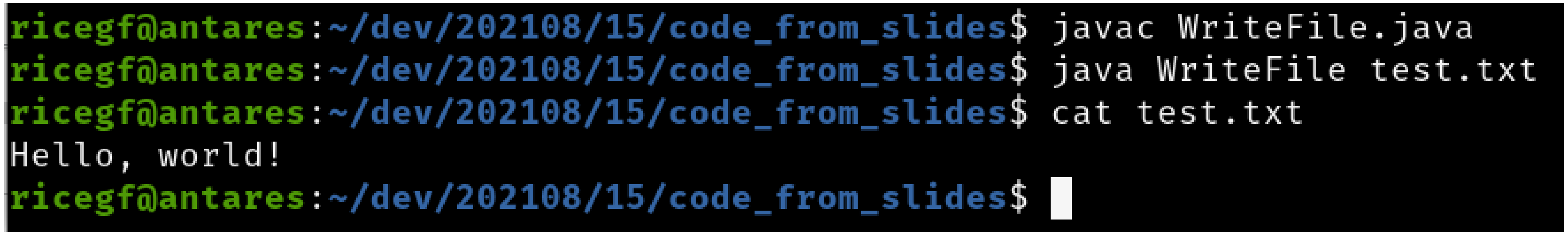
Know this idiom!

br.readLine() returns null for end of file

Opening a File for Writing

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

class WriteFile {
    public static void main(String[] args) throws IOException {
        BufferedWriter br = new BufferedWriter(new FileWriter(args[0]));
        br.write("Hello, world!\n");
        br.close();
    }
}
```

So FileReader and FileWriter create the streams, and BufferedReader and BufferedWriter manage reading and writing to them.

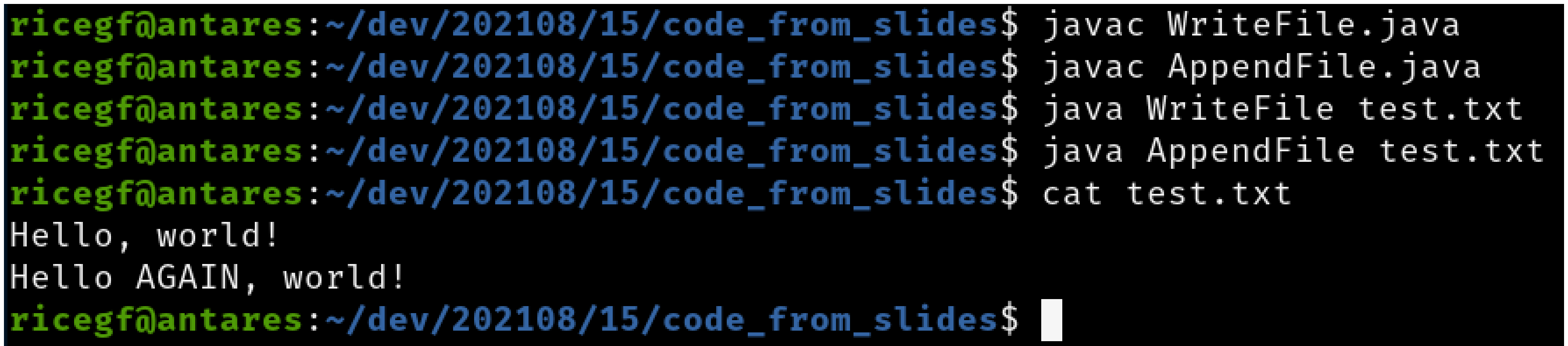


```
ricegf@antares:~/dev/202108/15/code_from_slides$ javac WriteFile.java
ricegf@antares:~/dev/202108/15/code_from_slides$ java WriteFile test.txt
ricegf@antares:~/dev/202108/15/code_from_slides$ cat test.txt
Hello, world!
ricegf@antares:~/dev/202108/15/code_from_slides$ 
```

Opening a File for Appending

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

class WriteFile {
    public static void main(String[] args) throws IOException {
        BufferedWriter br = new BufferedWriter(new FileWriter(args[0], true));
        br.write("Hello AGAIN, world!\n");
        br.close();
    }
}
```

FileWriter includes an overloaded constructor with a boolean 2nd parameter. If true, the existing file is opened at the end, so that text can be appended (added to the end of) that file.



```
ricegf@antares:~/dev/202108/15/code_from_slides$ javac WriteFile.java
ricegf@antares:~/dev/202108/15/code_from_slides$ javac AppendFile.java
ricegf@antares:~/dev/202108/15/code_from_slides$ java WriteFile test.txt
ricegf@antares:~/dev/202108/15/code_from_slides$ java AppendFile test.txt
ricegf@antares:~/dev/202108/15/code_from_slides$ cat test.txt
Hello, world!
Hello AGAIN, world!
ricegf@antares:~/dev/202108/15/code_from_slides$ █
```

An Exception(al) Problem

- If exception occurs during br.write, br.close is never called!
- One (awkward) solution is a lot of try / catch / finally (!)

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class WriteFileTry {
    public static void main(String[] args) throws IOException {
        BufferedWriter br = null;
        try {
            br = new BufferedWriter(new FileWriter(args[0]));
            br.write("Hello, world!\n");
        } catch (IOException e) {
            System.err.println("Failed to write: " + e);
        } finally {
            try {
                if(br != null) br.close(); // close if open
            } catch (IOException e) {
                System.err.println("Failed to close: " + e);
            }
        }
    }
}
```

Uuuugly!

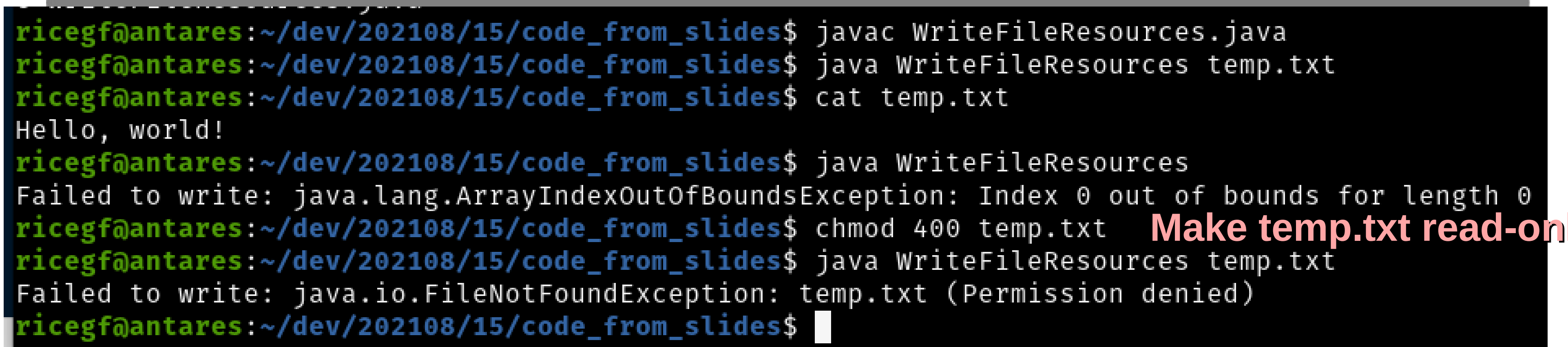
Try With Resources FTW!

- We can instead include the resource declaration in the try itself

Declare the BufferedWriter *as part of the try itself.* Catch will “auto-close” it for you - guaranteed!

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class WriteFileResources {
    public static void main(String[] args) throws IOException {
        try (BufferedWriter br = new BufferedWriter(new FileWriter(args[0]))) {
            br.write("Hello, world!\n");
        } catch (Exception e) {
            System.err.println("Failed to write: " + e);
        }
    }
}
```

MUCH better!



```
ricegf@antares:~/dev/202108/15/code_from_slides$ javac WriteFileResources.java
ricegf@antares:~/dev/202108/15/code_from_slides$ java WriteFileResources temp.txt
ricegf@antares:~/dev/202108/15/code_from_slides$ cat temp.txt
Hello, world!
ricegf@antares:~/dev/202108/15/code_from_slides$ java WriteFileResources
Failed to write: java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
ricegf@antares:~/dev/202108/15/code_from_slides$ chmod 400 temp.txt
ricegf@antares:~/dev/202108/15/code_from_slides$ java WriteFileResources temp.txt
Failed to write: java.io.FileNotFoundException: temp.txt (Permission denied)
ricegf@antares:~/dev/202108/15/code_from_slides$
```

Make temp.txt read-only

Try-with-resources and the `AutoCloseable` Interface

- When the try / catch block exits, the Java runtime calls the `void close()` method on all objects instanced in the try-with-resources
  - Each class must `implements AutoCloseable`
  - `AutoCloseable` requires the `void close()` method
  - Calling this method is *guaranteed* no matter what!
- Implementing classes include BufferedReader, StringReader, XMLReader, ZipFile...
- If your class manages resources, it can implement `AutoCloseable` and rely on try-with-resources too!

https://docs.oracle.com/javase/17/docs/api/java/lang/AutoCloseable.html

Encapsulated Save / Open

- How should we write and restore the data encapsulated in our classes
  - We could make all fields package-private and create a **`SaveOpen`** class to save and open it
  - But this compromises encapsulation
  - And we *still* need one **`SaveOpen`** class per package
- A better approach is to add 2 class members to *every* class we write containing persistent data
  - A **`save`** method to write the object's fields to a file
  - A constructor to recreate the object from a file

Save Method

- What parameters would a save method need?
  - ONLY a stream to which its fields could be written
  - A **`BufferedWriter`** would be perfect!
- The save method writes each of its fields to the **`BufferedWriter`** instance, *one line per field*
  - This greatly simplifies recreating the object
  - The newline is a natural field divider in the file
  - **`public void save(BufferedWriter bw) { }`**

Constructor

- What parameters would a constructor need?
  - ONLY a stream from which its fields could be read
  - A **`BufferedReader`** would be perfect!
- The constructor reads each field from the **`BufferedReader`** instance, *one line per field*
  - This greatly simplifies recreating the object
  - The newline is a natural field divider in the file
  - **`public Foo(BufferedReader br) { }`**

A Simple Class

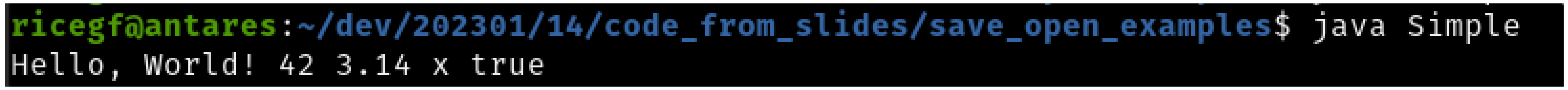
- Class Simple encapsulates a String and one each of our most common primitives

```java
public class Simple {
    public Simple(String aString, int anInt, double aDouble,
                  char aChar, boolean aBoolean) {
        this.aString  = aString;   this.anInt = anInt;
        this.aDouble  = aDouble;   this.aChar = aChar;
        this.aBoolean = aBoolean;
    }

    @Override
    public String toString() {
        return aString + " " + anInt + " "
             + aDouble + " " + aChar + " "
             + aBoolean;
    }

    private String aString;
    private int anInt;
    private double aDouble;
    private char aChar;
    private boolean aBoolean;
}
```

Testing a Simple Class

- This simple main method creates a Simple object and prints it to the console

```
    public static void main(String[] args) {
        // Create and print a simple object
        Simple simple = new Simple(
            "Hello, World!", 42, 3.14, 'x', true);
        System.out.println(simple.toString());
}
```



```
ricegf@antares:~/dev/202301/14/code_from_slides/save_open_examples$ java Simple
Hello, World! 42 3.14 x true
```

- Now we need to add save and open capability

Testing a Simple Class

- These are the imports for our data classes

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
```

- Remember, IOException is *checked*
  - Our readline() and write() methods must handle it
  - EITHER use a **`try / catch`**
  - OR add **`throws IOException`** to the constructor and save method

Adding Save and Open Capability to class Simple

- Write each field on a separate line to BufferedWriter

```
    public void save(BufferedWriter bw) throws IOException {
        bw.write(     aString  + '\n');
        bw.write("" + anInt    + '\n');
        bw.write("" + aDouble  + '\n');
        bw.write("" + aChar    + '\n');
        bw.write("" + aBoolean + '\n');
    }
```

IMPORTANT: The order of each write and corresponding readLine must match *exactly*!

- Recreate each field from a BufferedReader line

```
    public Simple(BufferedReader br) throws IOException {
        this.aString  =                      br.readLine();
        this.anInt    = Integer.parseInt    (br.readLine());
        this.aDouble  = Double.parseDouble  (br.readLine());
        this.aChar    =                      br.readLine().charAt(0);
        this.aBoolean = Boolean.parseBoolean(br.readLine());
    }
```

Here we elect to throw IOException out of the constructor / method.
Columns are exaggerated to emphasize the pattern to follow.

Comparing Simple Objects

- Override the equals method

```
    @Override
    public boolean equals(Object o) {
        if(this == o) return true;
        if(o == null || this.getClass() != o.getClass()) return false;
        Simple s = (Simple) o;
        return aString.equals(s.aString)
            && anInt    == s.anInt
            && aDouble  == s.aDouble
            && aChar    == s.aChar
            && aBoolean == s.aBoolean;
    }
```

- This allows us to ensure the saved object and opened object are the same!
- As usual, comparing those double fields brings worry of rounding errors
  - Perhaps **`Math.abs(aDouble - s.aDouble) < 0.0001`** instead?

Now for the Controller Class
(Named WithSimple in This Case)

- First we set up our controller class

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

import java.util.Scanner;

public class WithSimple {
    private String filename = "Untitled.simple";
    private Simple simple = null;
    private Simple simpleRecreated = null;

    private Scanner in = new Scanner(System.in);
```

These are the imports every Controller with open and save capability will need.

FileReader and FileWriter do the actual opening of the files for us.

In addition to the data objects, we add a **`filename`** String that ”remembers” which filename is currently open (or was last saved).

Now for the Controller Class
(Named WithSimple in This Case)

- Now we need save() and saveAs methods

```
    // save() opens filename and tells simple to write itself
    private void save() {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter(filename))) {
            simple.save(bw);
            System.out.println("Wrote simple to " + filename);
        } catch (Exception e) {
            System.err.println("Failed to save: " + e);
        }
    }

    // saveAs obtains a new filename and then calls save to write it
    private void saveAs() {
        System.out.print("Enter a Simple filename to save: ");
        String s = in.nextLine();
        if(s.isEmpty()) return;
        filename = s;
        save();
    }
```

Try-with-resources!

These are “behaviors” called from our main menu

Now for the Controller Class
(Named WithSimple in This Case)

- Next we need an open() method

```
    // Open requests a new filename, but gives the option
    //   of keeping the existing filename if desired
    private void open() {
        System.out.print("Enter a Simple filename to open (Enter for '"
                       + filename + "'): ");
        String s = in.nextLine();
        if(!s.isEmpty()) filename = s;
        try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
            simpleRecreated = new Simple(br);
            System.out.println("Opened simpleRecreated from " + filename);

        } catch (Exception e) {
            System.err.println("Failed to read: " + e);
            simpleRecreated = null;
        }
    }
```

Try-with-resources!

These are "behaviors" called from our main menu

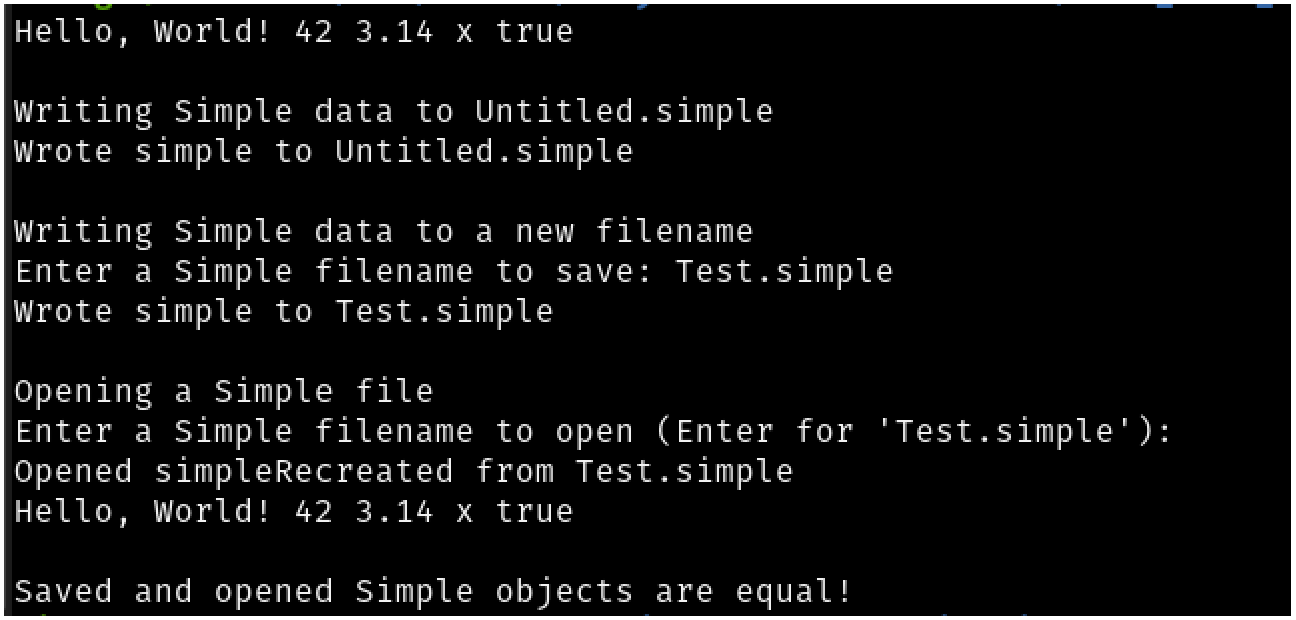
Now that we open to field **`simpleRecreated`**, so we can compare with what was saved from field **`simple`** to verify success.

Main Method and mdi()

```
public static void main(String[] args) {
    WithSimple ws = new WithSimple();
    ws.mdi();
}

public void mdi() {
   // Create and print a simple object
    simple = new Simple("Hello, World!", 42, 3.14, 'x', true);
    System.out.println(simple.toString());

    // Save the object to the default filename
    System.out.println("\nWriting Simple data to " + filename);
    save();

    // Save as a new filename
    System.out.println("\nWriting Simple data to a new filename");
    saveAs();

    // Open to a new Simple object simpleRecreated
    System.out.println("\nOpening a Simple file");
    open();
    System.out.println(simpleRecreated.toString());

    // Ensure what was saved was successfully opened
    if(simple.equals(simpleRecreated))
        System.out.println("\nSaved and opened Simple objects are equal!");
}
```

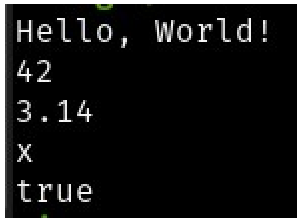
Testing Simple File I/O

- It works!



```
Hello, World! 42 3.14 x true

Writing Simple data to Untitled.simple
Wrote simple to Untitled.simple

Writing Simple data to a new filename
Enter a Simple filename to save: Test.simple
Wrote simple to Test.simple

Opening a Simple file
Enter a Simple filename to open (Enter for 'Test.simple'):
Opened simpleRecreated from Test.simple
Hello, World! 42 3.14 x true

Saved and opened Simple objects are equal!
```

- Our save file format evolves naturally



```
Hello, World!
42
3.14
x
true
```

- You can easily see the data written in simple.txt

Less Simple Classes

- Enums
  - For Enum **`E e;`**, save as **`bw.write(e.name());`**
    and restore as **`e = E.valueOf(br.readLine());`**
- Arrays, ArrayLists, and other Collections / Maps
  - For **`ArrayList<Double> ds;`**, save the size first then each element:
    `bw.write(ds.size()); for(Double d: ds) bw.write("" + d + '\n');`
  - Recreate the List or Map and then add each element in turn
    `ds = new ArrayList<>(); int size = Integer.parseInt(br.readLine());`
    `while(size-- > 0) ds.add(Double.parseDouble(br.readLine());`
- Classes with fields that are classes
  - Classes we wrote should already have save methods and constructors
  - Other classes we must address individually – see their JavaDoc pages!
- Superclasses and subclasses
  - For superclass X, given **`X x;`**, first write subclass name, then save the object:
    `bw.write(x.getClass().getName()); x.save(bw);`
  - To restore, check the subclass name to determine the subclass constructor:
    `String s = br.readLine();if(s.equals("pkg.SubX")) x = new SubX(br)`;

A Class with Enums: Flag

- We'll save and open class Flag which contains 3 color enum fields

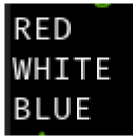
```
enum Color {BLUE, RED, WHITE}

class Flag {
    Color color1;    Color color2;    Color color3;

    public Flag(Color color1, Color color2, Color color3) {
        this.color1 = color1;
        this.color2 = color2;
        this.color3 = color3;
    }
    public void save(BufferedWriter bw) throws IOException {
        bw.write("" + color1.name() + '\n');
        bw.write("" + color2.name() + '\n');
        bw.write("" + color3.name() + '\n');
    }
    public Flag(BufferedReader br) throws IOException {
        color1 = Color.valueOf(br.readLine());
        color2 = Color.valueOf(br.readLine());
        color3 = Color.valueOf(br.readLine());
    }

    // Plus the obvious equals and toString implementation (see cse1325-prof)
}
```

Write each enum field using its name() method to avoid problems if the enum's toString() method is overloaded)

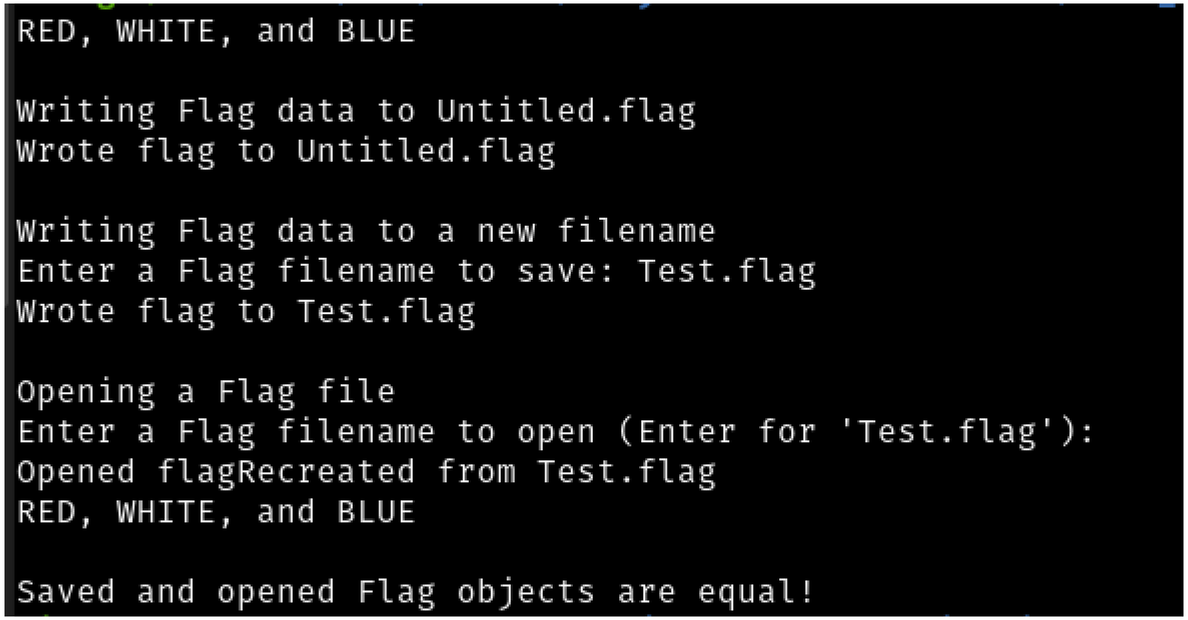
Restore each enum field using its valueOf() method

A Class with Enums: Testing Flag

- Using a similar controller, the Flag class is saved and opened successfully
- The file format is predictable



```
RED
WHITE
BLUE
```



```
RED, WHITE, and BLUE

Writing Flag data to Untitled.flag
Wrote flag to Untitled.flag

Writing Flag data to a new filename
Enter a Flag filename to save: Test.flag
Wrote flag to Test.flag

Opening a Flag file
Enter a Flag filename to open (Enter for 'Test.flag'):
Opened flagRecreated from Test.flag
RED, WHITE, and BLUE

Saved and opened Flag objects are equal!
```

Saving and Recreating Arrays

- Here we have an array and an ArrayList

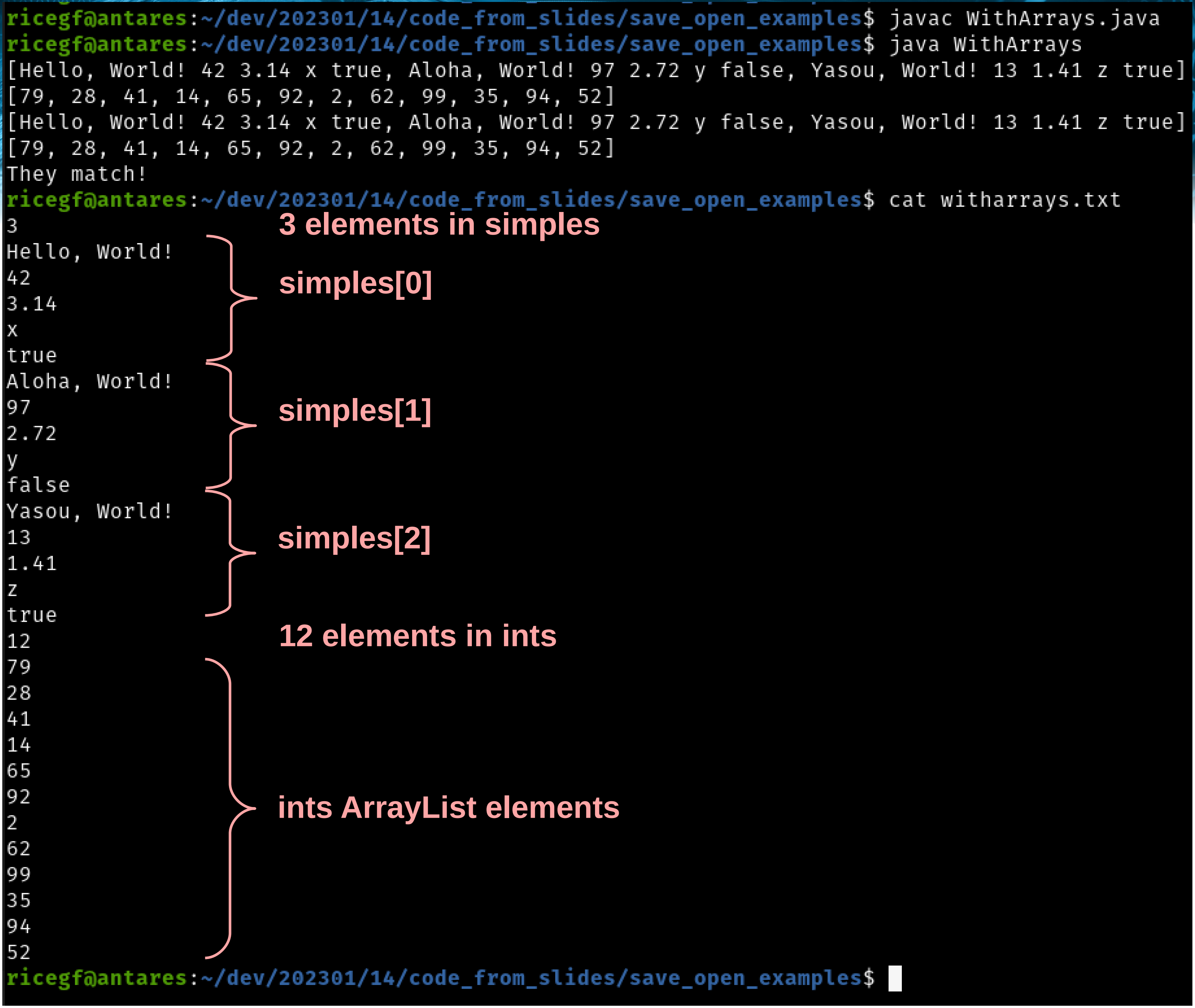
```
public class WithArrays {
    private Simple[] simples;          // Classic array
    private ArrayList<Integer> ints;  // ArrayLists, too!

    public WithArrays(Simple[] simples, int numInts) {
        this.simples = simples;
        ints = new ArrayList<>();
        while(numInts-- > 0) ints.add((int) (Math.random() * 100));
    }

    public void save(BufferedWriter bw) throws IOException {
        bw.write("" + simples.length + '\n');        // Length of array
        for(Simple s : simples) s.save(bw);          // Save the elements

        bw.write("" + ints.size() + '\n');           // Size of ArrayList
        for(int i : ints) bw.write("" + i + '\n'); // Save the elements
    }

    public WithArrays(BufferedReader br) throws IOException {
        int size = Integer.parseInt(br.readLine()); // Length of array
        simples = new Simple[size];                 // Instance the array
        for(int i=0; i<size; ++i) simples[i] = new Simple(br);

        size = Integer.parseInt(br.readLine());     // Size of ArrayList
        ints = new ArrayList<>();                   // Instance the ArrayList
        while(size-- > 0) ints.add(Integer.parseInt(br.readLine()));
    }
```

The full code is on cse1325-prof!

Here's the “normal” constructor.

To save the arrays, write the size first, then save each element.

To reconstruct, read in the size, then construct that many elements.

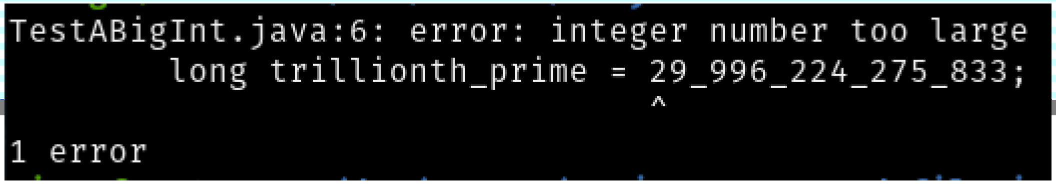


```
ricegf@antares:~/dev/202301/14/code_from_slides/save_open_examples$ javac WithArrays.java
ricegf@antares:~/dev/202301/14/code_from_slides/save_open_examples$ java WithArrays
[Hello, World! 42 3.14 x true, Aloha, World! 97 2.72 y false, Yasou, World! 13 1.41 z true]
[79, 28, 41, 14, 65, 92, 2, 62, 99, 35, 94, 52]
[Hello, World! 42 3.14 x true, Aloha, World! 97 2.72 y false, Yasou, World! 13 1.41 z true]
[79, 28, 41, 14, 65, 92, 2, 62, 99, 35, 94, 52]
They match!
ricegf@antares:~/dev/202301/14/code_from_slides/save_open_examples$ cat witharrays.txt
3
Hello, World!
42
3.14
x
true
Aloha, World!
97
2.72
y
false
Yasou, World!
13
1.41
z
true
12
79
28
41
14
65
92
2
62
99
35
94
52
ricegf@antares:~/dev/202301/14/code_from_slides/save_open_examples$
```

A Class with Fields that are Classes

- Might as well use Simple as “our” class
- Use BigInteger as “their” class – no save()!

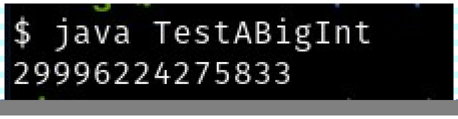
```
public class TestABigInt {
    public static void main(String[] args) {
        long trillionth_prime = 29_996_224_275_833;
        System.out.println(trillionth_prime);
    }
}
```



```
TestABigInt.java:6: error: integer number too large
        long trillionth_prime = 29_996_224_275_833;
                                ^
1 error
```

  - Class BigInteger handles *any* size integer!

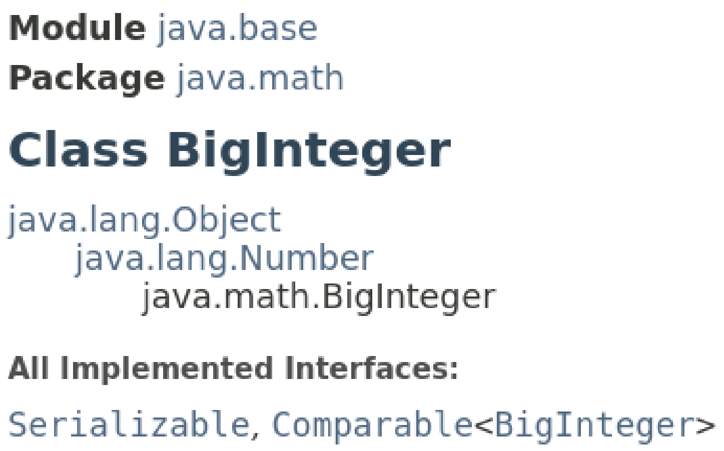
```
import java.math.BigInteger;

public class TestABigInt {
    public static void main(String[] args) {
        BigInteger bi = new BigInteger("29996224275833"); // Doesn’t permit separators *sigh*
        System.out.println(bi);
    }
}
```



```
$ java TestABigInt
29996224275833
```

How to Handle BigInteger

- Classes from other sources (like the library) may not provide explicit save and constructor equivalents
- Search the documentation for one or more *methods* that extract the data in a format compatible with a *constructor*
  - We'll ignore toString and BigInteger(String) as they won't always be available (though you should use them if they are)
  - Instead, BigInteger offers this promising non-String option:



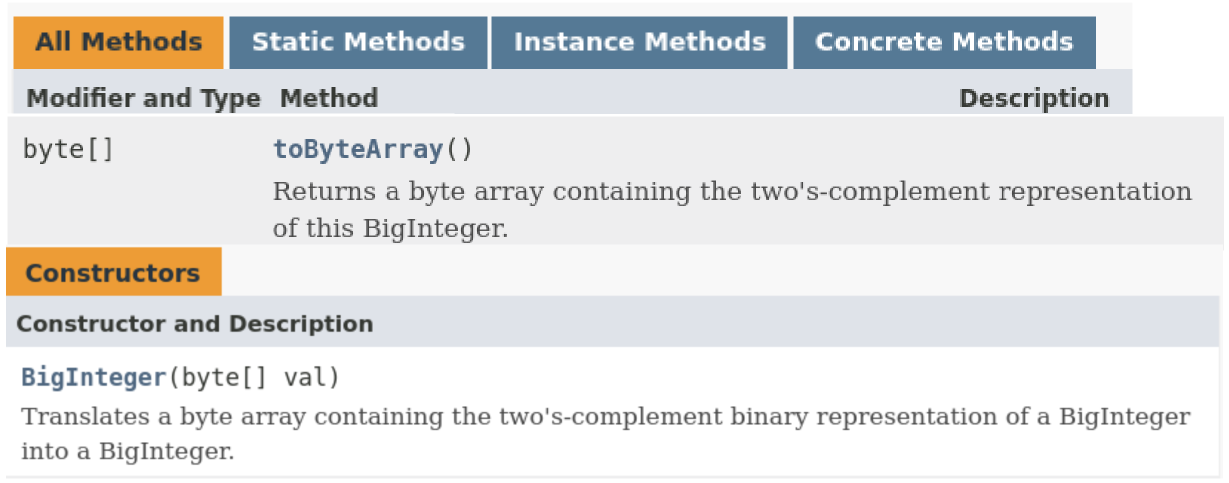
Module java.base
Package java.math

Class BigInteger

java.lang.Object
    java.lang.Number
        java.math.BigInteger

All Implemented Interfaces:

Serializable, Comparable<BigInteger>



Modifier and Type	Method	Description
byte[]	toByteArray()	Returns a byte array containing the two's-complement representation of this BigInteger.

Constructors

Constructor and Description
BigInteger(byte[] val) Translates a byte array containing the two's-complement binary representation of a BigInteger into a BigInteger.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/math/BigInteger.html

Handling Class Fields
in WithClasses

- Here's the save and constructor

See the rest of this class on cse1325-prof!

Write the best representation of the classes as you can.

Construct replacement field values using that data.

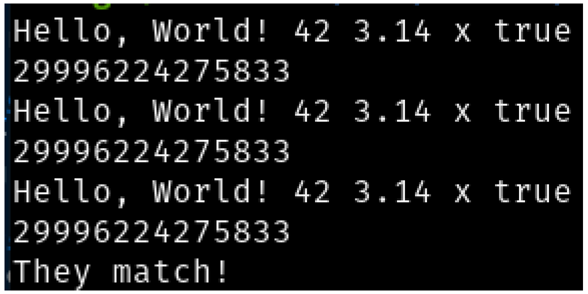
```java
public class WithClasses {
    private Simple simple;   // A class we wrote
    private BigInteger bi;   // A class we did NOT write

    public void save(BufferedWriter bw) throws IOException {
        simple.save(bw);
        byte[] bytes = bi.toByteArray();
        bw.write("" + bytes.length + "\n");           // Write number of bytes in the BigInteger
        for(Byte b : bytes) bw.write("" + b + "\n");  // Write each byte
    }

    public WithClasses(BufferedReader br) throws IOException {
        simple = new Simple(br);
        int size = Integer.parseInt(br.readLine());   // Read number of bytes in the BitInteger
        byte[] bytes = new byte[size];                // Create array of bytes to hold them
        for(int i=0; i<size; ++i) bytes[i] = Byte.parseByte(br.readLine());  // Read in the bytes
        bi = new BigInteger(bytes);                   // Recreate the BigInteger
    }
```

- Some classes may be too complex to extract enough data to recreate them
  - But any other approach would have the same issue!

Testing WithClasses

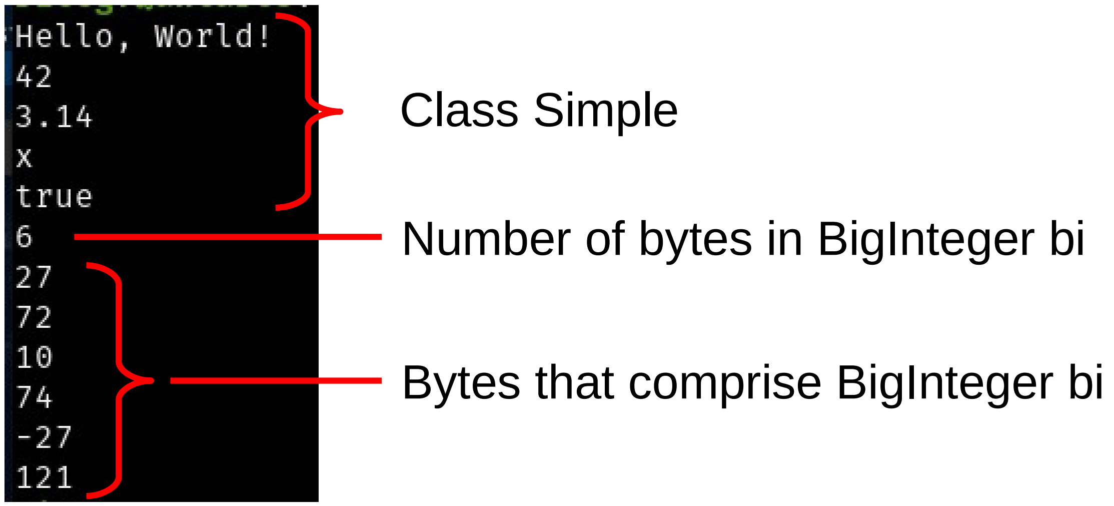
- It works!
- The trillionth prime is stored in 6 bytes

Output



```
Hello, World! 42 3.14 x true
29996224275833
Hello, World! 42 3.14 x true
29996224275833
Hello, World! 42 3.14 x true
29996224275833
They match!
```

File withclasses.txt



```
Hello, World!
42
3.14
x
true
6
27
72
10
74
-27
121
```

Class Simple
Number of bytes in BigInteger bi
Bytes that comprise BigInteger bi

A Class with Inheritance: Home

Featuring an ArrayList of Subclasses

```
class Home {
    ArrayList<Animal> pets = new ArrayList<>();

    public Home(Animal... animals) {
        for(Animal animal : animals)
            pets.add(animal);
    }
    public void save(BufferedWriter bw) throws IOException {
        bw.write("" + pets.size() + '\n');
        for(Animal pet : pets) {
            bw.write(pet.getClass().getName() + '\n');
            pet.save(bw);
        }
    }

    public Home(BufferedReader br) throws IOException {
        int size = Integer.parseInt(br.readLine());
        while(size-- > 0) {
            String type = br.readLine();
            if(type.equals("Dog")) pets.add(new Dog(br));
            else if(type.equals("Cat")) pets.add(new Cat(br));
            else throw new IOException("Bad pet type: " + type);
        }
    }
```

This ArrayList is the challenge here.
It contains both Dog and Cat objects
(both subclasses of Animal).
See the full code at cse1325-prof.

First, note how we save each pet,
which may be a Dog or Cat subclass.
pet.getClass().getName() returns
the full package.class name
of the type for each object –
in this case, just Dog or Cat.

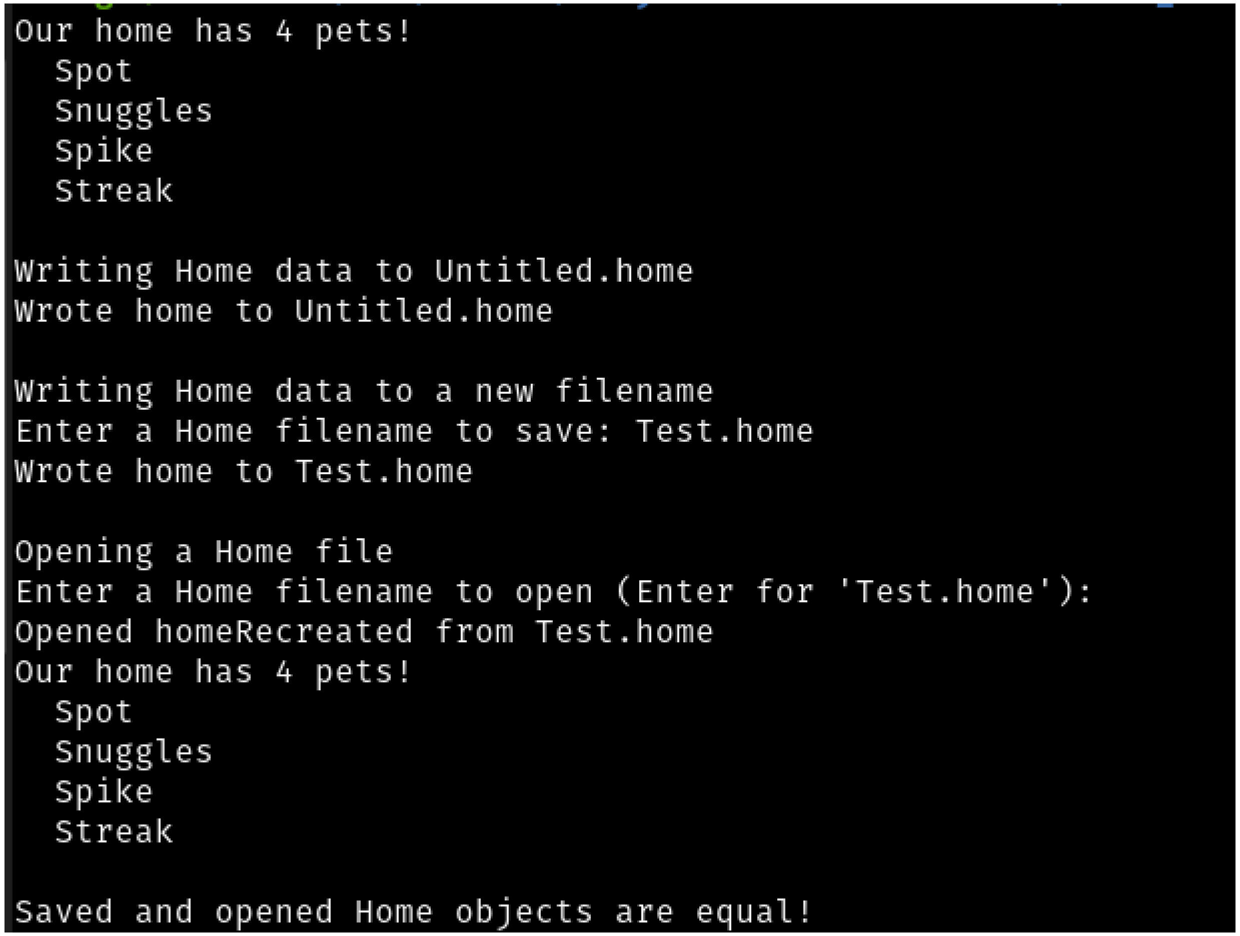
Then the pet saves itself.

To restore the array list, we read the type of each pet (a String) and compare it
to the subclass names to discover which constructor to call.

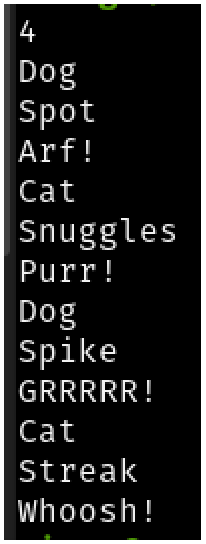
Then the subclass constructor restores itself.

Testing Home

- Using a similar controller, this works, too!
- You can see the subclass types in the file



```
Our home has 4 pets!
  Spot
  Snuggles
  Spike
  Streak

Writing Home data to Untitled.home
Wrote home to Untitled.home

Writing Home data to a new filename
Enter a Home filename to save: Test.home
Wrote home to Test.home

Opening a Home file
Enter a Home filename to open (Enter for 'Test.home'):
Opened homeRecreated from Test.home
Our home has 4 pets!
  Spot
  Snuggles
  Spike
  Streak

Saved and opened Home objects are equal!
```



```
4
Dog
Spot
Arf!
Cat
Snuggles
Purr!
Dog
Spike
GRRRRR!
Cat
Streak
Whoosh!
```

A Note on Saving with Inheritance

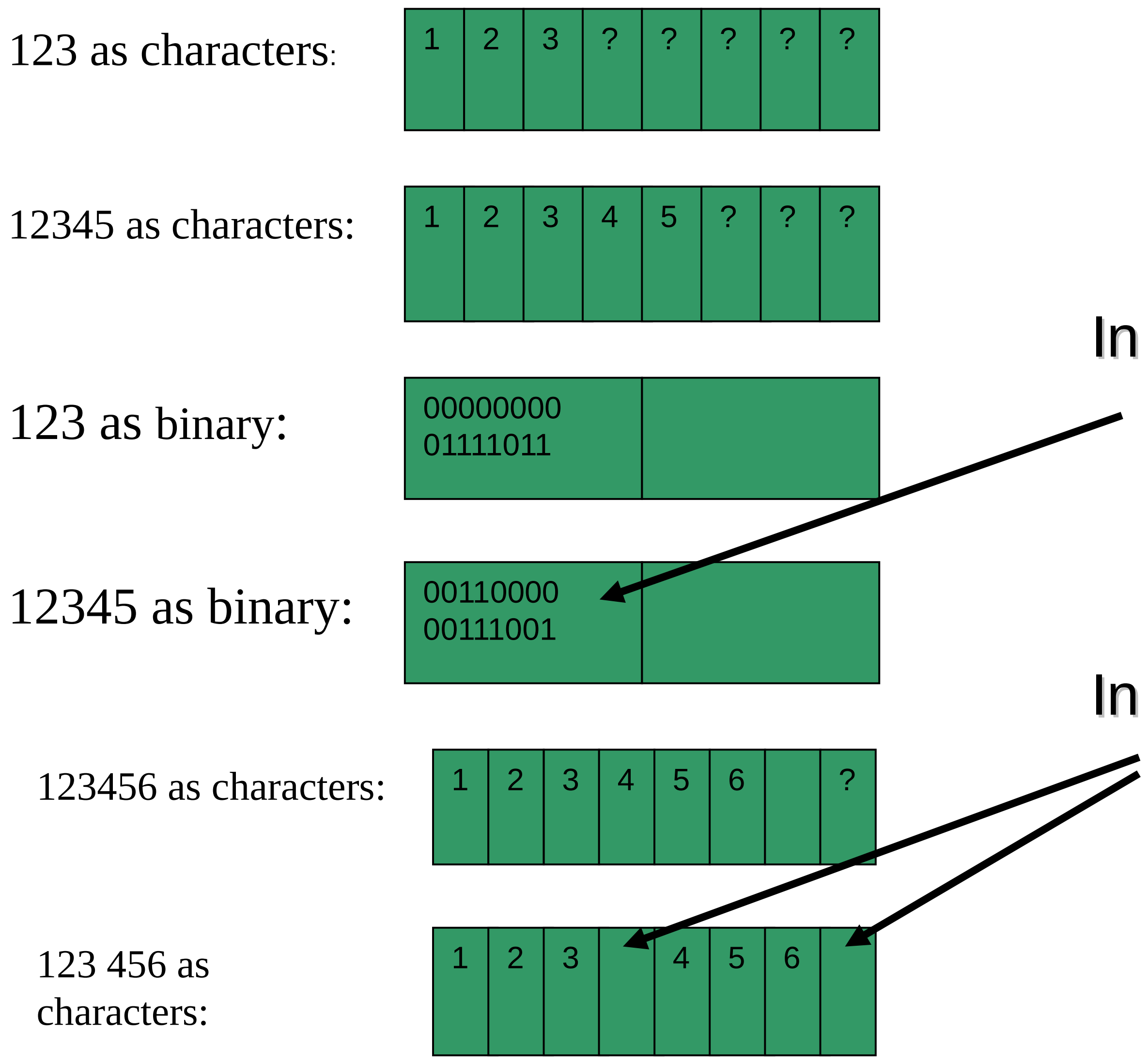
- It's very bad form for the superclass to include the names of its subclasses
  - This makes the dependency from subclass to superclass bi-directional – bad for maintainability
- Could we instance a Java class just using its name in a String? Is that *possible*?
  - Yes! This is called "introspection"
  - It's something like "meta-gaming" – Java evaluates its own runtime image to instance objects on the fly
  - But introspection is *advanced* Java, so we'll grit our teeth and accept a sub-optimal solution for now

What We Learned Today

- Using **`BufferedWriter`** and **`BufferedReader`**
  - Using try-with-resources to ensure opened files are always closed when the try / catch exits
  - Handling *checked* exceptions
    - Use a try / catch as usual
    - Add a **`throws`** clause to the method or constructor declaration, requiring the caller to handle it
      - Yes, your main method may include a **`throws`** clause, in which case the program will abort if the checked exception is thrown
- How to use them in object-oriented programs *without* breaking encapsulation!

The End

Text vs. binary files



In binary files, we use offsets and sizes to delimit values

In text files, we use character delimiters and separation / termination characters to delimit values

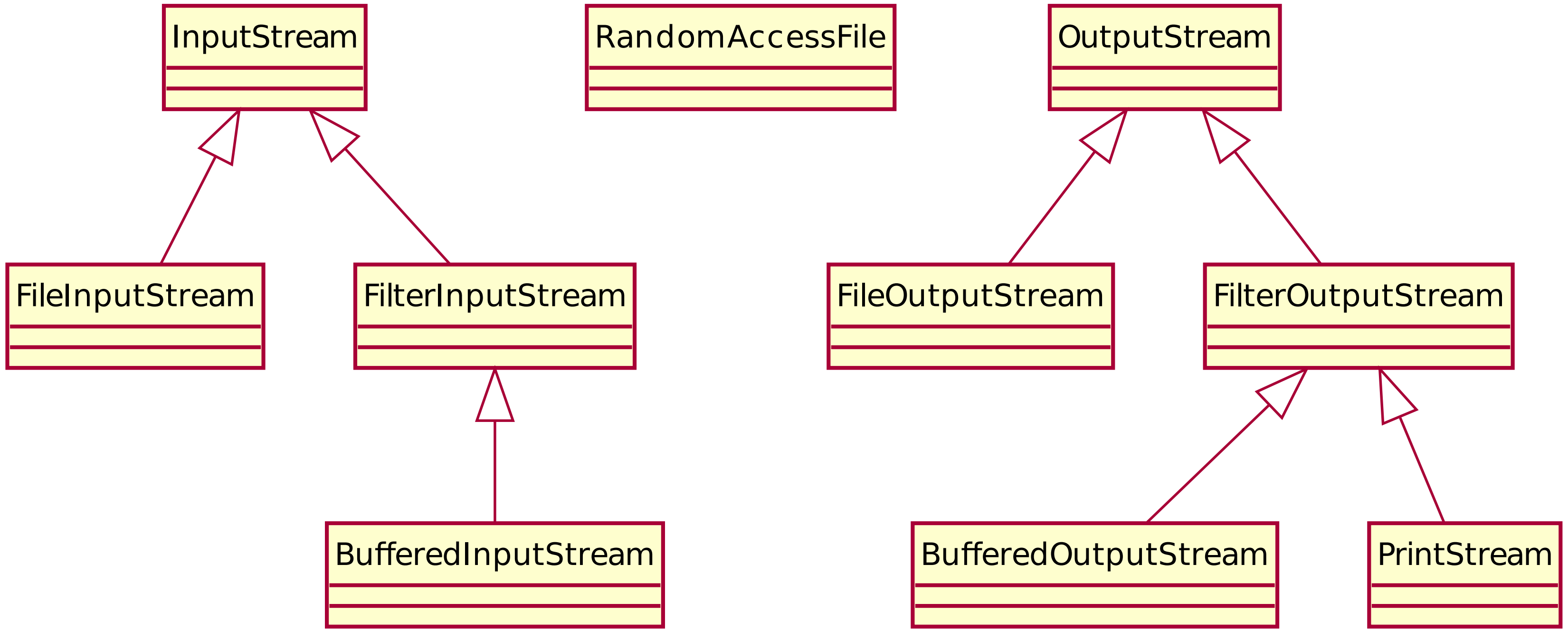
Java Binary (Byte) Streams

- Java has numerous stream classes, including*
  - Basic: **OutputStream** and **InputStream**†
  - Buffered: **BufferedOutputStream** and **BufferedInputStream**
  - Random Access: **RandomAccessFile**
  - Formatted: **PrintStream**‡
  - File: **FileOutputStream** and **FileInputStream**
- Each pair are classes instanced to create a stream
  - We usually use Buffered and Basic for bytes, too

* You may notice a pattern…
† System.in is an InputStream instance
‡ System.out and System.err are PrintStream instances

Binary Streams Class Diagram

- BufferedInputStream / BufferedOutputStream require an InputStream / OutputStream in their constructor
  - They merely add buffering to the I/O stream
  - Use a FileInputStream / FileOutputStream for file I/O

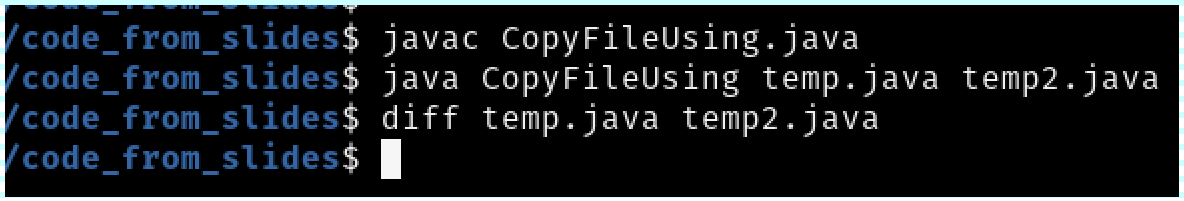


Selected classes are shown

Binary File I/O

- Java uses Stream libraries for binary I/O
- This is an unbuffered file copy with try-using-resources, copying one byte at a time

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyFileUsing {
    public static void main(String args[]) throws IOException {
        if(args.length != 2) {
            System.err.println("usage: java CopyFile [sourcefile] [destinationFile]");
            System.exit(-1);
        }
        try (
            FileInputStream in = new FileInputStream(args[0]);
            FileOutputStream out = new FileOutputStream(args[1]);
        ) {
            int c;
            while ((c = in.read()) != -1) out.write(c);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

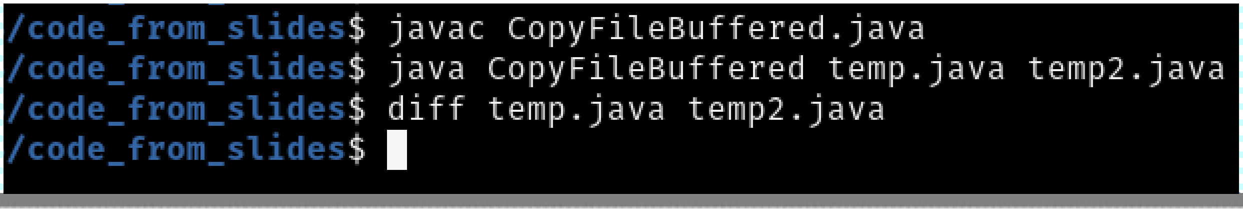


```
/code_from_slides$ javac CopyFileUsing.java
/code_from_slides$ java CopyFileUsing temp.java temp2.java
/code_from_slides$ diff temp.java temp2.java
/code_from_slides$ █
```

Buffered Binary File I/O

- This version is buffered, copying 16 kbytes at a time

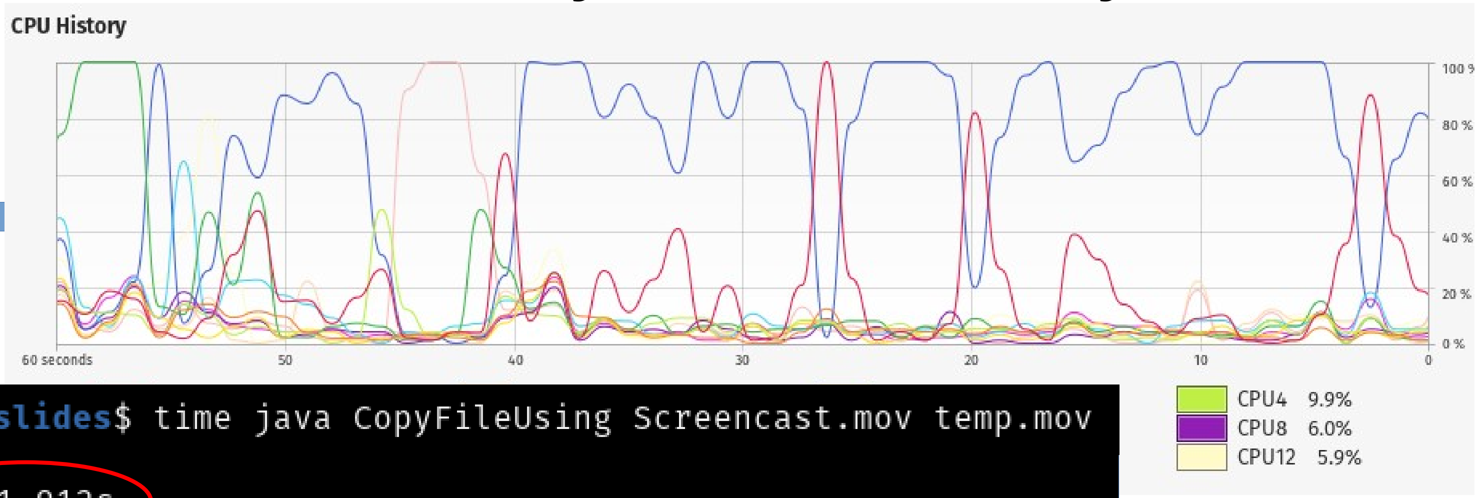
```java
import java.io.BufferedInputStream;     import java.io.BufferedOutputStream;
import java.io.FileInputStream;         import java.io.FileOutputStream;
import java.io.IOException;

public class CopyFileBuffered {
    private static final int BUFFER_SIZE = 16384; // 16 kilobytes
    public static void main(String[] args) {
        if(args.length != 2) {
            System.err.println("usage: java CopyFileBuffered [source] [destination]");
            System.exit(-1);
        }
        try (
            BufferedInputStream in = new BufferedInputStream(
                                         new FileInputStream(args[0]));
            BufferedOutputStream out = new BufferedOutputStream(
                                           new FileOutputStream(args[1]));
        ) {
            byte[] buffer = new byte[BUFFER_SIZE];  // Buffer to hold block of data
            int length;                             // Number of bytes actually read
            while ((length = in.read(buffer)) > 0) out.write(buffer, 0, length);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



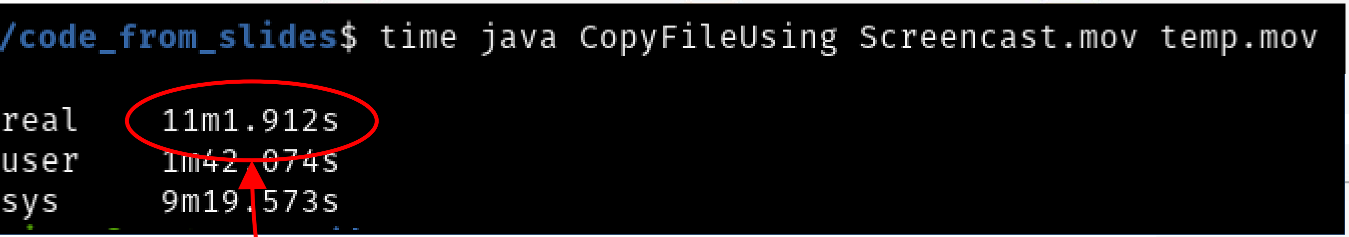
```
/code_from_slides$ javac CopyFileBuffered.java
/code_from_slides$ java CopyFileBuffered temp.java temp2.java
/code_from_slides$ diff temp.java temp2.java
/code_from_slides$ 
```

Performance of Buffered I/O

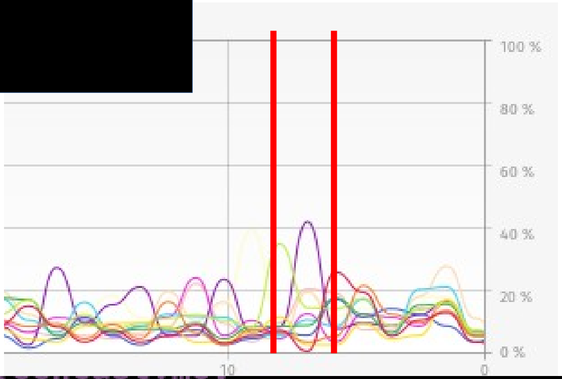
- Buffered I/O may be noticeably faster



Unbuffered

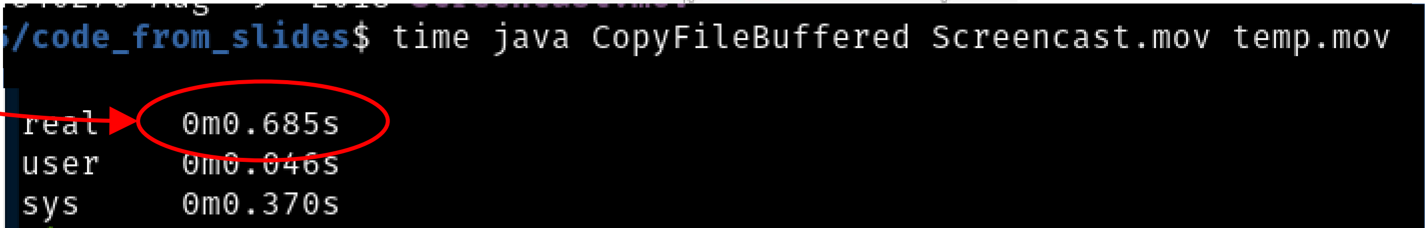


```
/code_from_slides$ time java CopyFileUsing Screencast.mov temp.mov

real    11m1.912s
user    1m42.074s
sys     9m19.573s
```



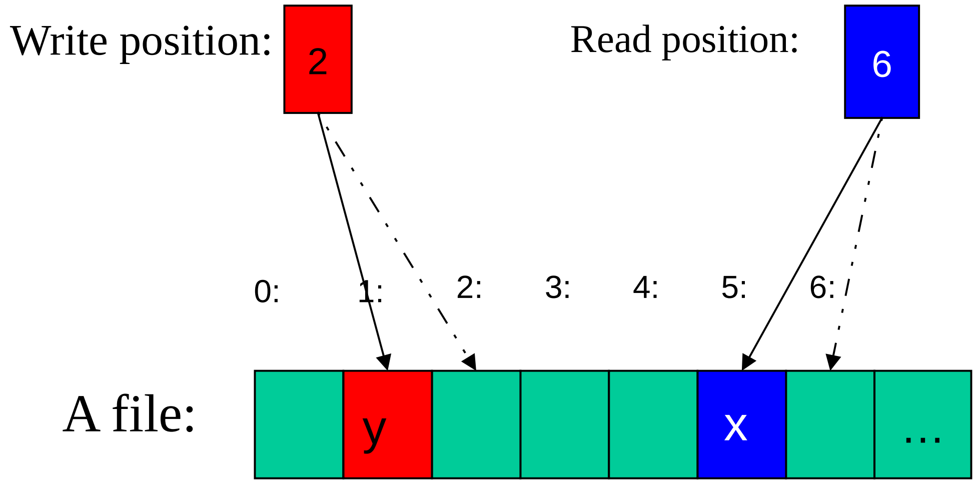
Buffered

Buffered is
966x faster!



```
/code_from_slides$ time java CopyFileBuffered Screencast.mov temp.mov

real    0m0.685s
user    0m0.046s
sys     0m0.370s
```

Positioning in a filestream



Note: Many languages (e.g., C++) maintain separate read and write positions. Java has one position for both.

```
RandomAccessFile fs = new RandomAccessFile(filePath, "rw"); // open for input & output

fs.seek(5);  // move reading position (‘g’ for ‘get’) to 5 (the 6th character)
int b =fs.readByte(); // read the x and increment the reading position to 6
System.out.println("sixth character is " + (char) b + '(' << b << ")\n");

fs.seek(1);  // move writing position to 1 (the 2nd character)
fs.writeByte((int) 'y'); // write and increment writing position to 2
```

Example: MP3 Metadata

- MP3 music files often support the ID3 standard
  - 128 bytes of metadata appended to the music
  - Binary format (so we must seek!)
  - https://en.wikipedia.org/wiki/ID3#ID3v1
- Let’s write a Java program to extract it!
- We could also add methods to *change* it
  - Not for illicit purposes, of course – “a technology demo”
  - Left as an exercise for the student who wishes to do well in CSE1325, their degree, and their career

MP3Reader Constructor, close(), and Attributes

```java
import java.io.*;

public class MP3Reader implements AutoCloseable {
    public MP3Reader(String filePath) throws FileNotFoundException, IOException {
        mp3 = new RandomAccessFile(filePath, "r");
        length = mp3.length();

        // verify MP3 format
        String tag = readString(128, 3);
        if(!tag.equals("TAG"))
            throw new IOException(filePath + " doesn't support ID3v1 (" + tag + ")");
    }
    @Override
    public void close() {
        try {
            mp3.close();  // This releases the resource attribute mp3 when done
        } catch(IOException e) {
        }
    }
    private RandomAccessFile mp3;
    private long length;
    private static final String[] genres = {
        "Blues", "Classic rock", "Country", "Dance", "Disco", "Funk", "Grunge",
        // Continues for almost 100 genres!
```

AutoCloseable supports try-with-resources!
Needed because we leave mp3 file open.

Reading MP3 Fields

```
protected String readString(int offset, int size) throws IOException {
    mp3.seek(length - offset);
    byte[] bytes = new byte[size];
    mp3.read(bytes);
    return new String(bytes);
}
protected int readByte(int offset) throws IOException {
    mp3.seek(length - offset);
    return mp3.read();
}

String title() throws IOException {return readString(125, 30);}
String artist() throws IOException {return readString(95,30);}
String album() throws IOException {return readString(65, 30);}
String year() throws IOException {return readString(35, 4);}
String comment() throws IOException {
    if(hasTrack()) return readString(31,28);
    else return readString(31, 30);
}
boolean hasTrack() throws IOException {return readByte(3) == 0;}
int track() throws IOException {return readByte(2);}
String genre() throws IOException {
    int index = readByte(1);
    if(index >= genres.length) return "Unknown";
    else return genres[index];
}
```

Utility method readString returns a field of bytes (in ASCII) as a String

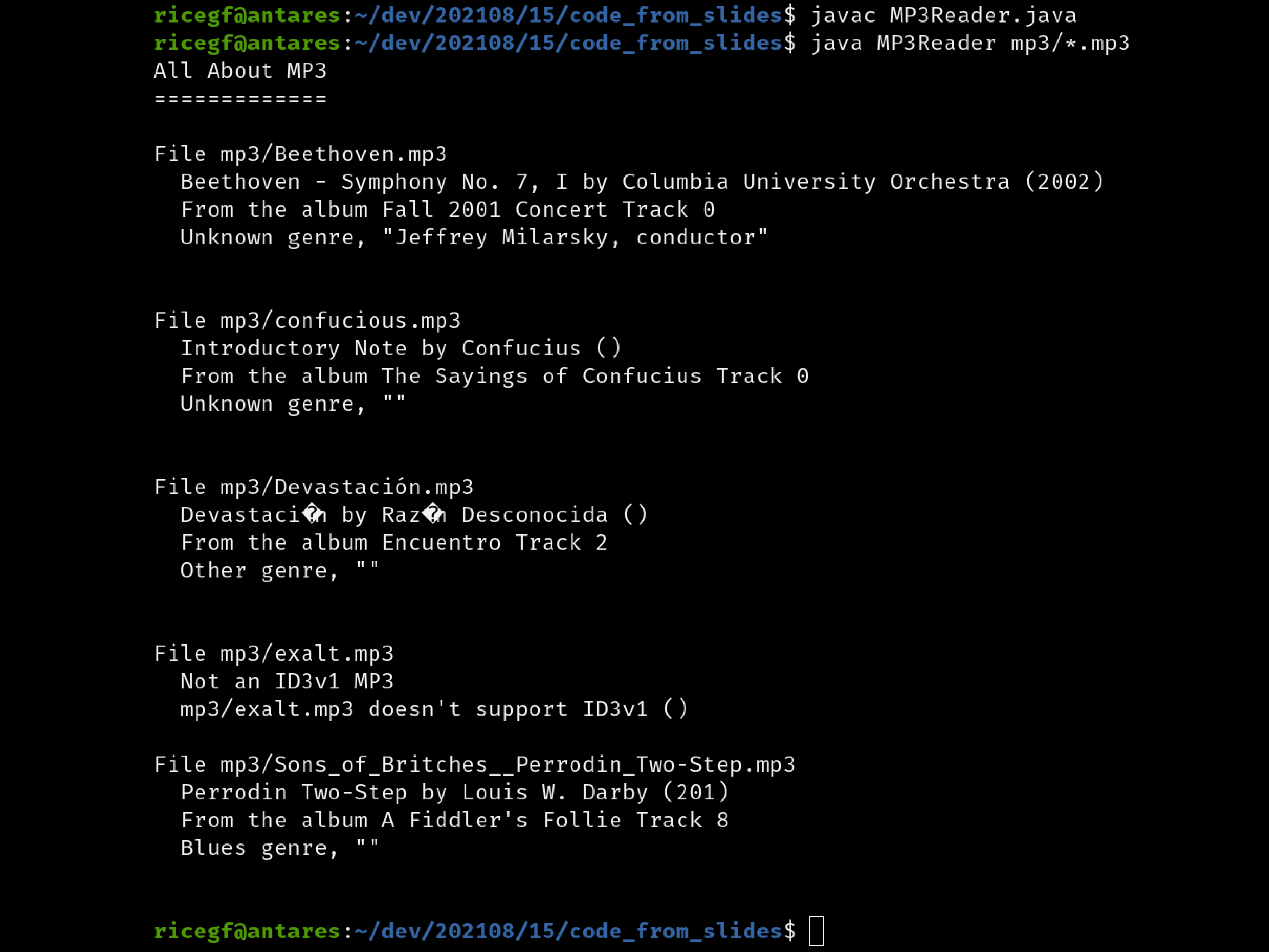
Utility method readByte returns a byte as an int

These offsets from the end of the file and field sizes come straight from the spec!

Main

- Main reports on any number of mp3 files
  - Note we are now a “resource” (because we open a file) and thus we should support try-with-resources

```
public static void main(String[] args) {
    System.out.println("All About MP3\n +
                       “=============\n");
    for(String file : args) {
        System.out.println("File " + file);
        try (MP3Reader mp3 = new MP3Reader(file)) {  Custom try-with-resources!
            System.out.println("  " + mp3.title()
                             + " by " + mp3.artist()
                             + " (" + mp3.year() + ")");
            if(mp3.hasTrack()) System.out.println("  From the album " + mp3.album()
                                               + " Track " + mp3.track());
            else System.out.println("  From the album " + mp3.album());
            System.out.println("  " + mp3.genre() + " genre, \""
                                    + mp3.comment() + "\"");
            System.out.println("\n");
        } catch (Exception e) {
            System.err.println("  Not an ID3v1 MP3\n  " + e.getMessage());
         }
    }
}
```



```
ricegf@antares:~/dev/202108/15/code_from_slides$ javac MP3Reader.java
ricegf@antares:~/dev/202108/15/code_from_slides$ java MP3Reader mp3/*.mp3
All About MP3
=============

File mp3/Beethoven.mp3
  Beethoven - Symphony No. 7, I by Columbia University Orchestra (2002)
  From the album Fall 2001 Concert Track 0
  Unknown genre, "Jeffrey Milarsky, conductor"


File mp3/confucious.mp3
  Introductory Note by Confucius ()
  From the album The Sayings of Confucius Track 0
  Unknown genre, ""


File mp3/Devastación.mp3
  Devastaci�n by Raz�n Desconocida ()
  From the album Encuentro Track 2
  Other genre, ""


File mp3/exalt.mp3
  Not an ID3v1 MP3
  mp3/exalt.mp3 doesn't support ID3v1 ()

File mp3/Sons_of_Britches__Perrodin_Two-Step.mp3
  Perrodin Two-Step by Louis W. Darby (201)
  From the album A Fiddler's Follie Track 8
  Blues genre, ""


ricegf@antares:~/dev/202108/15/code_from_slides$ 
```

All Stream Classes
(For the Over-Achieving Student)

	Byte Based		**Character Based**	
	Input	**Output**	**Input**	**Output**
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data – Unformatte	DataInputStream	DataOutputStream		
– Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

Positioning vs Streaming

- Whenever you can
  - Use simple streaming
    - Streams/streaming is a very powerful metaphor
    - Write most of your code in terms of text streams
      e.g., rename the old file with a trailing '~' or '.bak', and write the updated file to the original filename
  - Positioning is far more error-prone
    - Handling of the end of file position is system dependent and basically unchecked
    - A subtle bug can destroy the file being edited

Boom by OpenClipart-Vectors per the Pixabay License
https://pixabay.com/vectors/boom-explosion-sound-detonation-2028563/