COMP1007

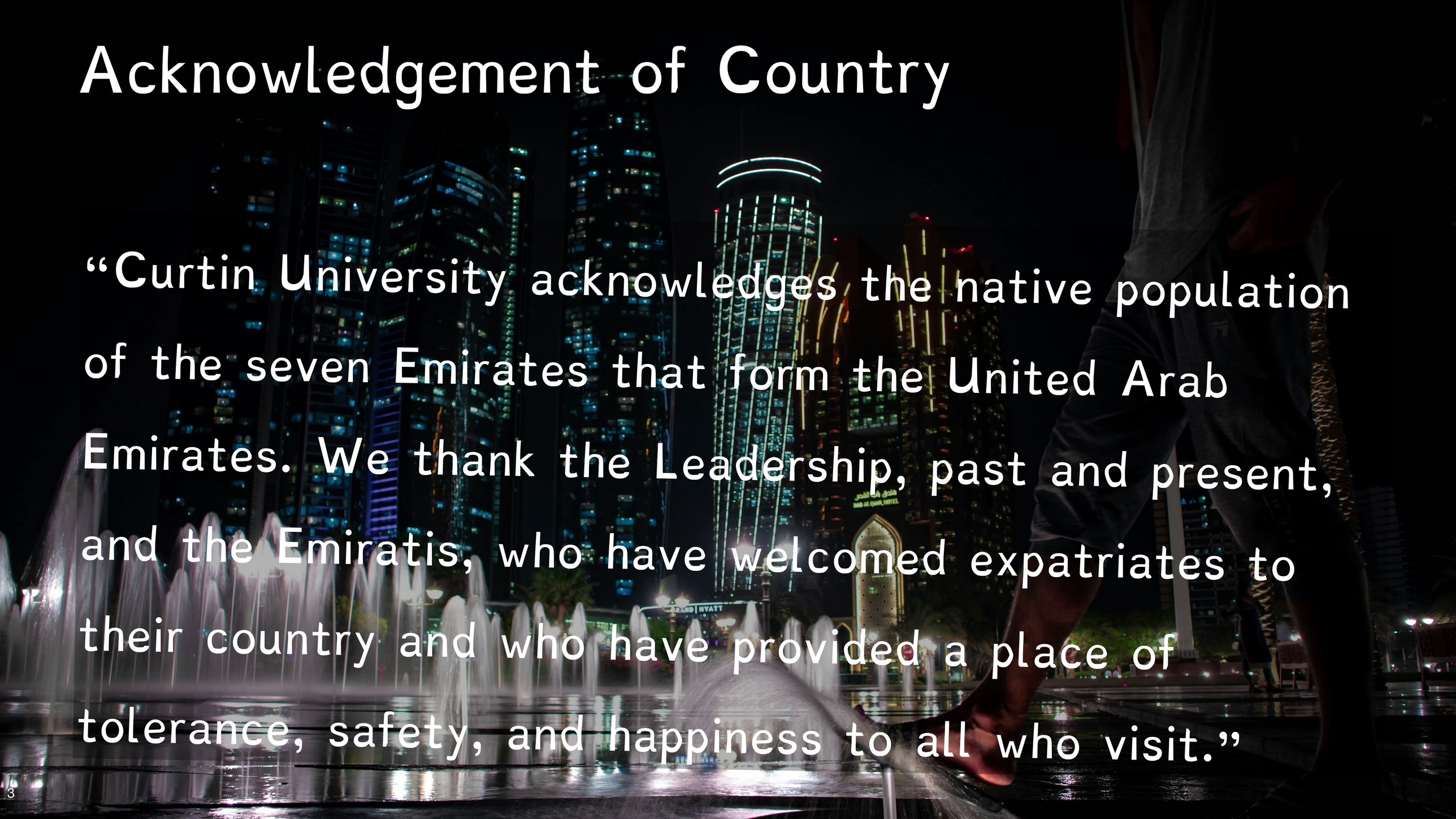
File I/O

Dr David A. McMeekin

Acknowledgement of Country

“I acknowledge the Wadjuk people of the Noongar nation on which Curtin University’s Boorloo Campus sits, and the Wangkatja people where the Karlkurla Campus sits, as the traditional custodians of the lands and waterways, and pay my respect to elders past, present and emerging.”

Acknowledgement of Country



“Curtin University acknowledges the native population of the seven Emirates that form the United Arab Emirates. We thank the Leadership, past and present, and the Emiratis, who have welcomed expatriates to their country and who have provided a place of tolerance, safety, and happiness to all who visit.”

COMP1007 - Unit Learning Outcomes

- Identify appropriate primitive data types required for the translation of pseudocode algorithms into Java;
- Design in pseudocode simple classes and implement them in Java in aLinux command-line environment;
- Design in pseudocode and implement in Java structured procedural algorithms in a Linux command-line environment;
- Apply design and programming skills to implement known algorithms in real world applications; and
- Reflect on design choices and communicate design and design decisions in a manner appropriate to the audience.

Outline

- Scope File IO
- Reading Files
- Exceptions
- Parsing
- Writing Files
- Reference Variables

Variable Scope

Variable Scope

```
import java.util.*;
public class ScopeApp03
{
    public static void main(String[] args)
    {
        //Local variables: numOne, numTwo
        //SCOPE: we are in main()
        int numOne = 12;
        int numTwo = 39;

        System.out.println("numOne is: " + numOne);
        System.out.println("numTwo is: " + numTwo);
    }
}
```

Variable Scope

```java
import java.util.*;
public class ScopeApp04
{
    public static void main(String[] args)
    {
        //Local variables: numOne, numTwo
        //SCOPE: Available within the main() method only
        int numOne = 12;
        int numTwo = 39;

        System.out.println("numOne is: " + numOne);
        System.out.println("numTwo is: " + numTwo);
        thisIsAnAmazingMethod();
    }

    public static void thisIsAnAmazingMethod()
    {
        // The variables numOne, numTwo do not exist within the scope of the
        // method thisIsAnAmazingMethod()
        // Compile error
        System.out.println("We are inside: thisIsAnAmazingMethod()");
        System.out.println("numOne is: " + numOne);
        System.out.println("numTwo is: " + numTwo);
    }
}
```

Variable Scope

```java
import java.util.*;
public class ScopeApp05
{
    public static void main(String[] args)
    {
        //Local variables: numOne, numTwo
        //SCOPE: Available within the main() method only
        int numOne = 12;
        int numTwo = 39;

        System.out.println("numOne is: " + numOne);
        System.out.println("numTwo is: " + numTwo);

        // The values in numOne, numTwo are passed to the
        // method thisIsAnAmazingMethod(numOne, numTwo)
        thisIsAnAmazingMethod(numOne, numTwo);
    }

    // pNumOne and pNumTwo are local variables within thisIsAnAmazingMethod() method
    // pNumOne and pNumTwo are intialised with the values in numOne and numTwo respectively.
    // SCOPE: pNumOne and pNumTwo are available within thisIsAnAmazingMethod() method only
    public static void thisIsAnAmazingMethod(int pNumOne, int pNumTwo)
    {
        System.out.println("We are inside: thisIsAnAmazingMethod()");
        System.out.println("pNumOne is: " + pNumOne);
        System.out.println("pNumTwo is: " + pNumTwo);
    }
}
```

Variable Scope

```java
import java.util.*;
public class ScopeApp06
{
    public static void main(String[] args)
    {
        //Local variables: numOne, numTwo
        //SCOPE: Available within the main() method only
        int numOne = 12;
        int numTwo = 39;

        System.out.println("numOne is: " + numOne);
        System.out.println("numTwo is: " + numTwo);

        // The values in numOne, numTwo are passed to the
        // method thisIsAnAmazingMethod(numOne, numTwo)
        thisIsAnAmazingMethod(numOne, numTwo);
        System.out.println(numThree);
    }

    // pNumOne and pNumTwo are local variables within thisIsAnAmazingMethod() method
    // pNumOne and pNumTwo are intialised with the values in numOne and numTwo respectively.
    // SCOPE: pNumOne and pNumTwo are available within thisIsAnAmazingMethod() method only
    public static void thisIsAnAmazingMethod(int pNumOne, int pNumTwo)
    {
        //Local variables: numThree
        //SCOPE: Available within the thisIsAnAmazingMethod() method only
        int numThree = 42;

        System.out.println("We are inside: thisIsAnAmazingMethod()");
        System.out.println("pNumOne is: " + pNumOne);
        System.out.println("pNumTwo is: " + pNumTwo);
        System.out.println("numThree is: " + numThree);
    }
}
```

Variable Scope

```java
import java.util.*;
public class ScopeApp07
{
    public static void main(String[] args)
    {
        //Local variables: numOne, numTwo
        //SCOPE: Available within the main() method only
        int numOne = 12;
        int numTwo = 39;

        System.out.println("numOne is: " + numOne);
        System.out.println("numTwo is: " + numTwo);

        // The values in numOne, numTwo are passed to the
        // method thisIsAnAmazingMethod(numOne, numTwo)
        thisIsAnAmazingMethod(numOne, numTwo);

        // numThree does not actually exist in this scope
        // Compile error
        System.out.println("numThree is: " + numThree);
    }

    // pNumOne and pNumTwo are local variables within thisIsAnAmazingMethod() method
    // pNumOne and pNumTwo are intialised with the values in numOne and numTwo respectively.
    // SCOPE: pNumOne and pNumTwo are available within thisIsAnAmazingMethod() method only
    public static void thisIsAnAmazingMethod(int pNumOne, int pNumTwo)
    {
        //Local variables: numThree
        //SCOPE: Available within the thisIsAnAmazingMethod() method only
        int numThree = 42;

        System.out.println("We are inside: thisIsAnAmazingMethod()");
        System.out.println("pNumOne is: " + pNumOne);
        System.out.println("pNumTwo is: " + pNumTwo);
        System.out.println("numThree is: " + numThree);
    }
}
```

Variable Scope

```java
import java.util.*;
public class GlobalsNotSoGood
{
    static final int numThree = 42; //Global constant.
                                    //SCOPE: available through the entire class.

    public static void main(String[] args)
    {
        //Local variables:
        //SCOPE: Available within the main() method only
        int numOne = 12;
        int numTwo = 39;

        System.out.println("numOne is: " + numOne);
        System.out.println("numTwo is: " + numTwo);
        thisIsAnAmazingMethod(numOne, numTwo);
        System.out.println("numThree in main() is: " + numThree);
    }

    // pNumOne and pNumTwo are local variables within thisIsAnAmazingMethod() method
    // pNumOne and pNumTwo are intialised with the values in numOne and numTwo respectively.
    // SCOPE: pNumOne and pNumTwo are available within thisIsAnAmazingMethod() method only

    public static void thisIsAnAmazingMethod(int pNumOne, int pNumTwo)
    {
        System.out.println("We are inside: thisIsAnAmazingMethod()");
        System.out.println("pNumOne is: " + pNumOne);
        System.out.println("pNumTwo is: " + pNumTwo);
        System.out.println("numThree in thisIsAnAmazingMethod() is: " + numThree);
    }
}
```

File I/O

Why Files?

- RAM (Random Access Memory) is volatile and private to an application;
- Not good for:
  - Storing application data long-term (between runs);
  - Sharing information between applications.
- In contrast, files stored on disk are:
  - (semi)permanent;
  - Can be shared between applications; and
  - Can be manipulated by users outside the application.

File Input/Output

- Files are effective input and outputs to/from applications;
- File I/O: File Input File Output;
- Three steps of File I/O:
  1. Open the file;
  2. Read data from a file and/or write data to a file; and
  3. Close the file.

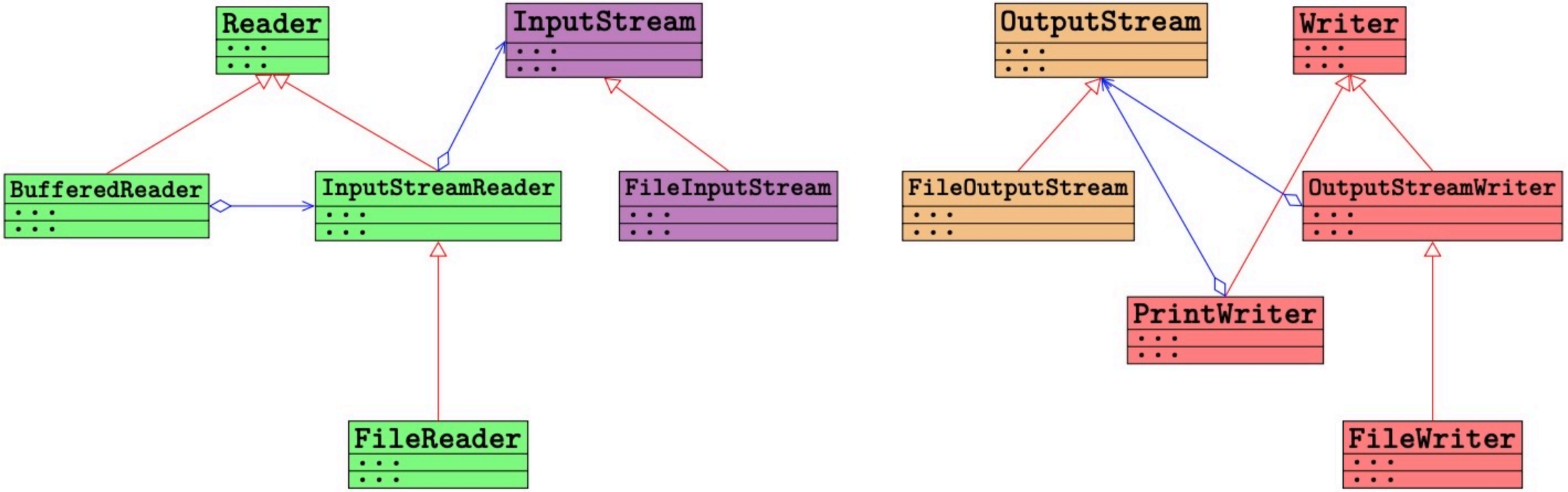
File I/O in Java

- Java encapsulates File I/O in a set of classes;
- All File I/O classes support the above three steps;
- Different classes manipulate different file types;
- Different classes perform I/O in different ways;
- Other kinds of I/O are part of the same set of classes:
  - Keyboard input, screen output, network I/O, device I/O, etc.
- The different I/O streams are unified in a single consistent programming interface.

Java I/O Classes

- The I/O classes can be split into three groups:
  1. Classes that represent an I/O stream (e.g., file, device, etc.);
  2. Classes that read from a stream; and
  3. Classes that write to a stream.
- Each I/O type, file, network, device, uses a specialised version of the basic classes;
- PDI concentrates on the File I/O classes.

Java I/O Classes Hierarchy



- **FileReader** and **FileWriter** are more for convenience and we won't be talking about them, or using them.

Steps in Java for Reading a File

1. Create a stream object for a file:

   FileInputStream

2. Create an object that can read that stream:

   InputStreamReader

3. Read and process data from a file;

4. Close the FileInputStream

NB: A FileReader combines the first two steps into one.

Steps in Java for Writing a File

1. Create a stream object for a file:

   FileOutputStream

2. Create an object that can write to that stream:

   OutputStreamWriter or PrintWriter

3. Write data to the file;

4. Close the FileOutputStream

NB: A FileWriter combines the first two steps into one.

Reading Files

Efficient File I/O

- `InputStreamReader` has two `read()` methods:
  - One reads a single `byte` at a time;
  - returns -1 if no more `byte`s to read (end-of-file reached).
  - One reads an array of `byte`'s at a time.
- Rarely ever need a single `byte` of data at a time;

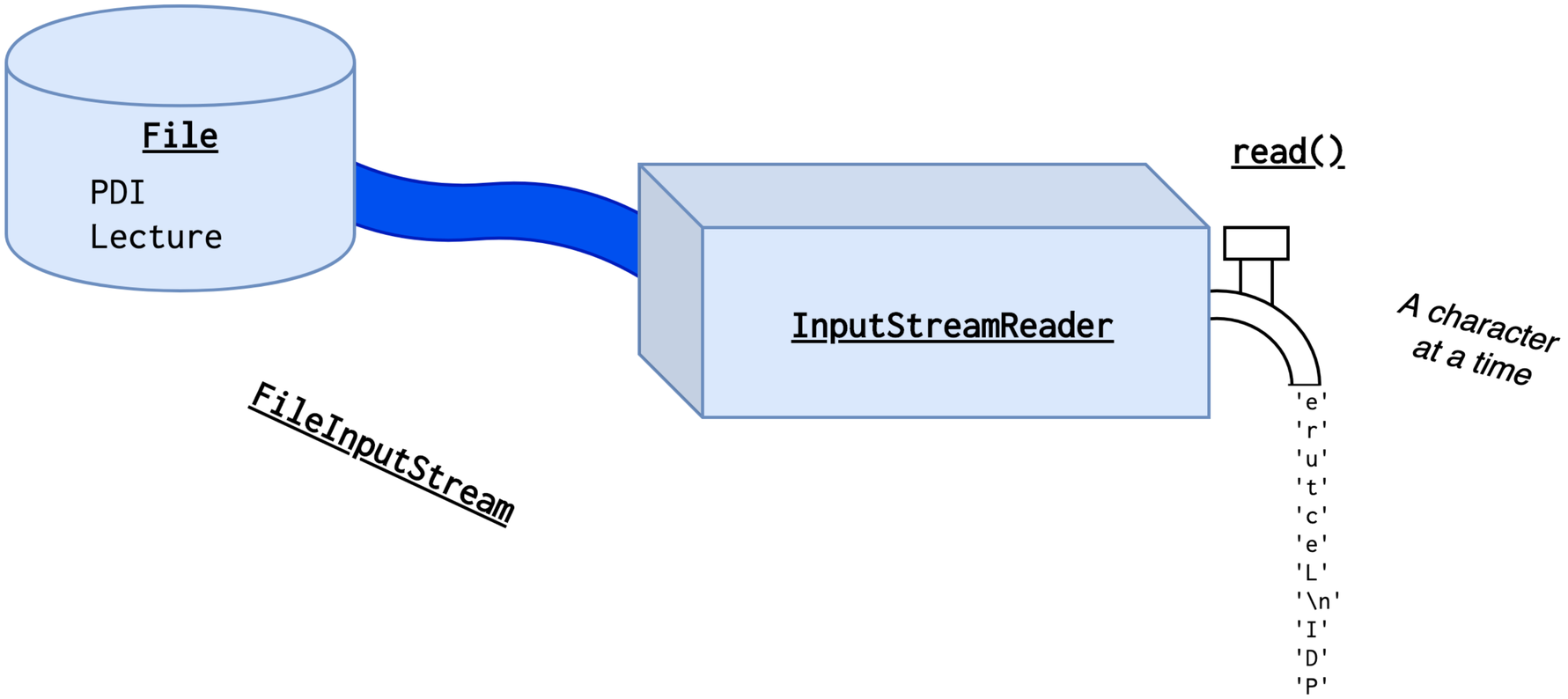
Text vs. Binary Data

- Data files fall into two broad categories:
  - Files containing text data (relatively unstructured);
  - Files containing binary data (everything else).
- Binary data is highly structured:
  - e.g., images, databases, executable files, etc.;
  - There is a lot of prior knowledge on precisely where information exists in the file and how large blocks are;
- Text data is unstructured, difficult to know beforehand how many bytes to read:
  - How many characters in an arbitrary line of text???

Efficient File I/O Processing

- Reading blocks of N byte's from a file is okay:
  - Generally will know beforehand how many bytes are needed.
- When reading text, must be constantly ready for the end of data as it can't be predicted:
  - end of word, end of line, or end of file.
- One approach: read data in a byte at a time, check each byte for end-of-X (' ', '\n' or -1)
  - A problem is this is very slow, HDD are fastest when reading blocks of data at a time;
  - Similar to filling a bucket one drop at a time vs. opening the tap and letting the water flow freely.

InputStreamReader



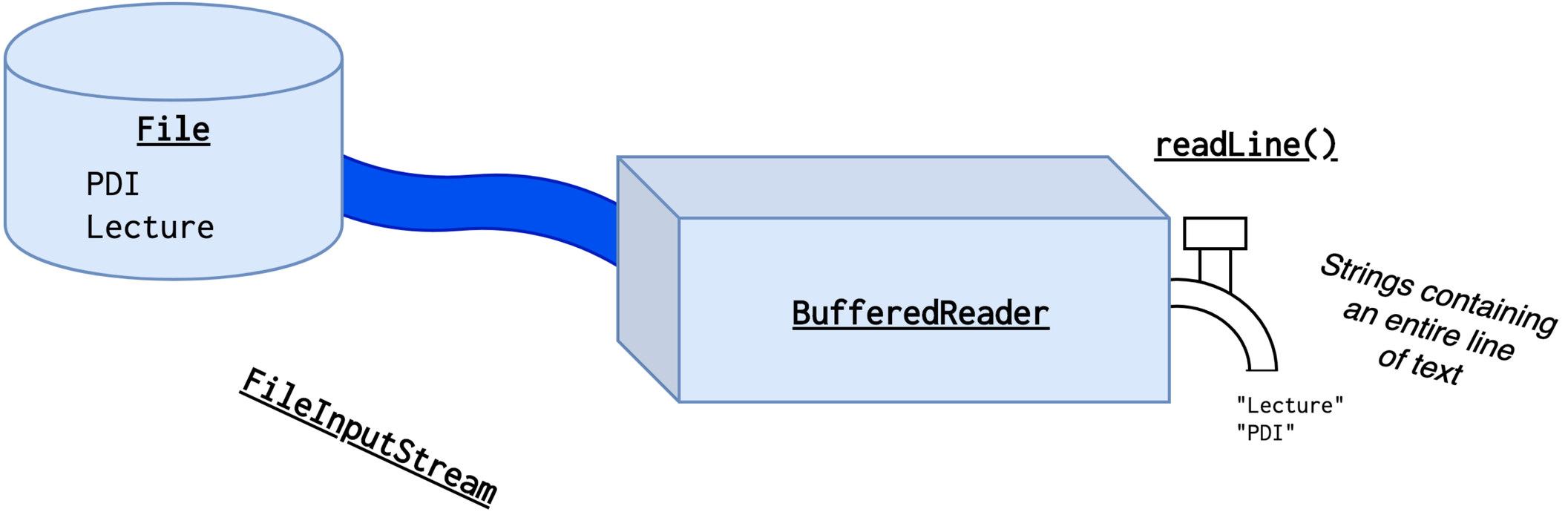
- Note: `read()` returns `-1` when end-of-file reached;
- Note the "`\n`" (newline character) is returned like any other character in the file.

BufferedReader

- BufferedReader reads in chunks of data, buffers in memory search for end-of-X
  - readLine() method looks for the end-of-line;
  - The idea, read in ('buffer') chunks of 1024 bytes, search these bytes in RAM one at a time for the '\n'
    - '\n' = newline character marks the end of the line;
    - As RAM is much faster than disk, this is a more efficient method.
- When a line is found, it is extracted and returned, the rest of the buffer is kept in memory;
- For subsequent readLine() calls, it first checks the buffer for the '\n' before reading another 1024-byte chunk from the file.

Note: a BufferedReader is created from another Reader.

BufferedReader



- `readLine()` does not include the '`\n`' in the returned lines;
- `readLine()` returns `null` when end-of-file occurs.

File Reading: One Line at a Time – Pseudocode

```
METHOD: readFile
IMPORT: filename (String)
EXPORT: none
    theFile = OPENFILE filename
    lineNum = 0
    READ line FROM theFile          // Read first line from file
    WHILENOT(theFile=EOF)           //EOF = end of file.
                                    // Detecting this is language-specific
        lineNum = lineNum + 1
        DISPLAY(line)               // Display line content
        READ line FROM theFile      // Read the next line
    ENDWHILE
    CLOSEFILE theFile               // Close the file
END readFile
```

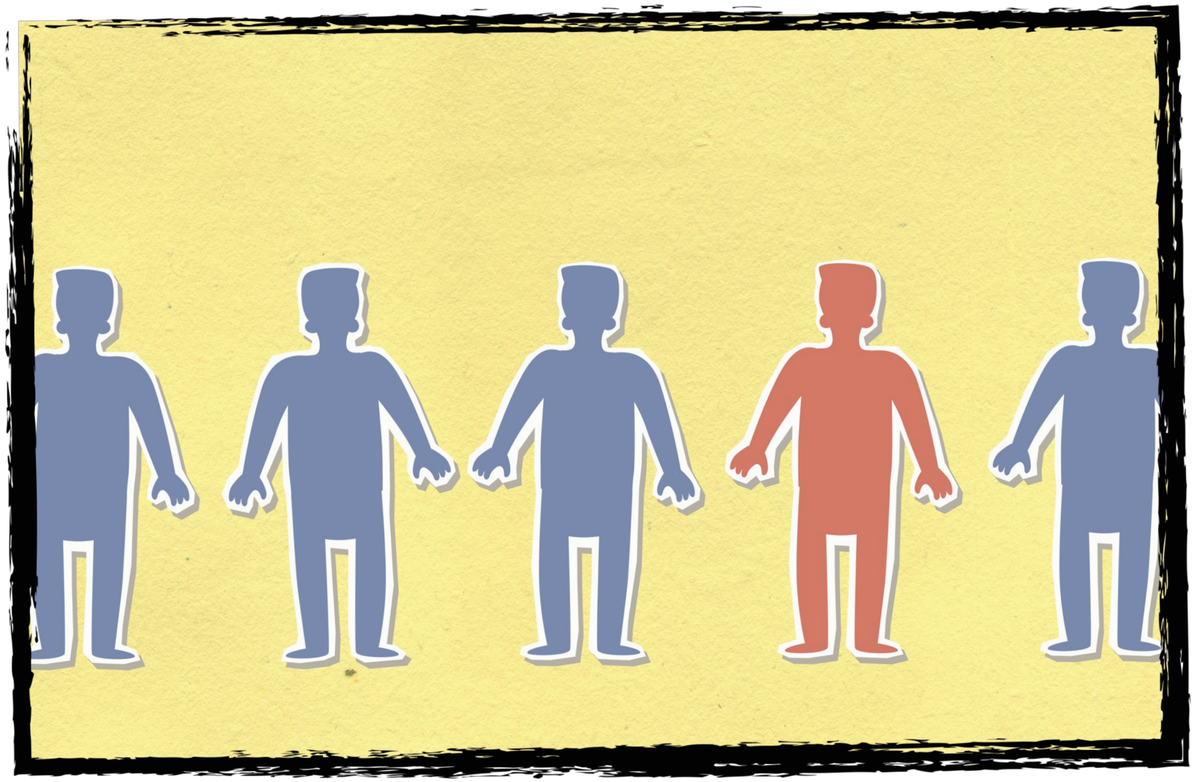
Notes on File Handling

- Close files as soon as possible;
- The OS must track what files are open;
- The OS remembers where you were in the file;
- Resources available for tracking are limited:
  - Run out and the OS will terminate your program.
- Don't leave files open – clean them up early:
  - Java doesn't free objects immediately - waits for garbage collector;
  - Always explicitly `close()` a file once finished with it.

Java File Reading: One Line at a Time

```java
public static void readFile(String pFileName)
{
    FileInputStream fileStream = null;
    InputStreamReader isr;
    BufferedReader bufRdr;
    int lineNum;
    String line;
    try
    {
        fileStream = new FileInputStream(pFileName);
        isr        = new InputStreamReader(fileStream);
        bufRdr     = new BufferedReader(isr);
        lineNum    = 0;
        line       = bufRdr.readLine();
        while(line != null)
        {
            lineNum++;
            System.out.println(line);
            line = bufRdr.readLine();
        }
            fileStream.close();
    }
    catch(IOException errorDetails)
    {
        if(fileStream != null)
        {
            try
            {
                fileStream.close();
            }
            catch(IOException ex2)
            { }
        }
        System.out.println("Error in fileProcessing: " + errorDetails.getMessage());
    }
}
```

Exceptions



File Handling – Exceptions

- You must handle `IOException`'s;
- `IOException`'s must be caught;
- Check the `Exception` or the compiler complains;
- This forces implementing `try{} catch(){}` around close
- A method signature can end with `throws IOException` clause and the method does not need to catch the `IOException`;
- The calling method must catch the `IOException`.

File Reading: One Line at a Time - Java

```java
public void readFileMultipleException() throws IOException
{ // Why would it be bad to write "throws Exception" here?
    Scanner sc = new Scanner(System.in);
    String inFileName; // File Code omitted - see slide 22 boolean noFile;
    do
    {
        noFile = true; inFileName = sc.next();
        try
        {
            while(line != null)
            {
                processLine(line);
            }
            noFile = false;
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Couldn't find your file " + e.getMessage() + " Try again!");
        }
        catch(IOException e)
        {
            throw e;
        }
        catch(Exception e)
        {
            throw new Exception("I am a bad programmer: ", e);
        }
    } while(noFile);
}
```

Parsing

Parsing

- When dealing with text, it's often necessary to take it apart and organise it ready for processing/storage;
- This is called parsing – to determine and extract the structure of a piece of text:
  - The word originates from written language syntax analysis.
- Examples of where parsing is needed:
  - Natural language processing (e.g., spelling/grammar checks);
  - Building Web search indexes; and
  - Compilers – must parse code to detect statements and variables.

Tokenizing

- Tokenizing: the first step in parsing;
- Breaking a text stream into basic elements;
- These elements are called tokens;
- What they are depends on what is being parsed:
  - e.g., Single words, entire lines, equation terms, etc.
- Tokens: broken up by character(s) that delimits the boundary:
  - lines are separated by a '\n' newline character;
  - words are separated by spaces, commas and periods; and/or
  - equation operands are separated by operators + / * -

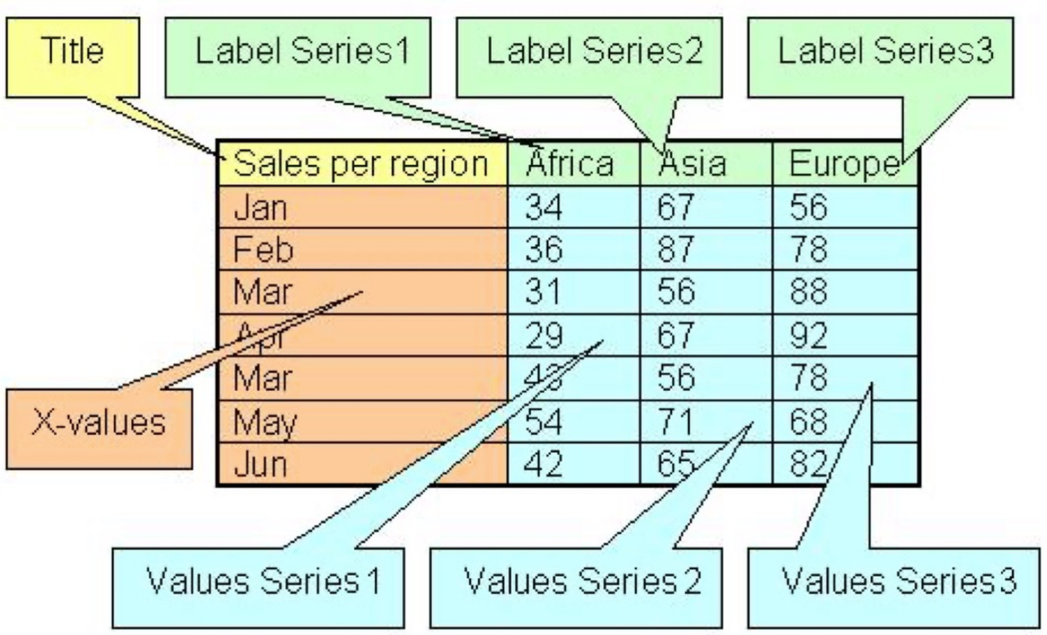
Tokenizing with Java

- Java provides two classes to assist in tokenizing:
  - StringTokenizer: used with Strings;
  - StreamTokenizer: used with Streams;
    - PDI will focus on split (String regex) (below)
- Java provides the split(String regex) method in the String class for simple String tokenizing:
  - is than the tokenizers, does NOT return delimiting character;
  - the regex is yet unknown, for PDI, we can use basic regex to split on.

Comma Separated Values (csv)

- Let's detour and introduce comma separated values;
  - We'll use this as an example for split() later on.
- We often need to store data to a file:
  - The question is, what format should we store it in?
- Table or matrix form, can be written as rows and columns:
  - One row per line;
  - Each row contains multiple fields, one per column;
  - Each field's value is separated by a comma ',' .

CSV Example



- The CSV file:

```
Sales per region,Africa,Asia,Europe
Jan,34,67,56
Feb,36,87,78
Mar,31,56,88
Apr,29,67,92
Mar,43,56,78
May,54,71,68
Jun,42,65,82
```

Multiple catch Clauses

- Previous example caught all IO Exceptions;
- What about handling various exceptions in different ways?
  - e.g., if a file doesn't exist, ask the user for a different file name, but if anything else goes wrong with the file, terminate the program.

- Every `try` must be followed by one or more `catch` clauses:
  - Order is important, it will attempt to catch the first Exception written first, if that is a less specific type then it may not catch a more precise type later
  - Explained more in Inheritance in DSA (COMP1002)

CSV Notes

- The delimiting commas have no trailing space;
- Column sizes don't have to be consistent across rows:
  - e.g., the first row (headings) has much longer fields than the same columns in subsequent rows.
- Numeric data is converted into its textual equivalent:
  - If we saved integers directly, we might get things that look like text '\n' (ASCII 13) or ',' (ASCII 44) but are merely part of the data.

Text vs. Binary

- You don't have to save table data in CSV format:
  - Dumping raw binary data is more efficient, and you know how large each field is (e.g., ints = 4 bytes).
- Advantages of CSV:
  - Easy for humans to read and edit; and
  - Highly portable to different platforms.
    - Big endian, vs. little endian
- Fields are explicitly separated with commas;
- CSV is a widely-known format:
  - Large amounts of spatial data supplied in CSV format;
  - XML is another standard format for data interchange.

String.split() and CSV Data

- Lets parse a CSV file with String.split();
- To do that: call a String variable's split method with what you want to "split" it on;
- It will return a String[] containing each part of the split in each element of the array;
- The array can be iterated over to extract the required data.

Parsing a Single CSV Row

```
private static void processLine(String csvRow)
{
    String[] splitLine;
    splitLine = csvRow.split(",");
    int lineLength = splitLine.length;

    for(int i = 0; i < lineLength; i++)
    {
        System.out.print(splitLine[i] + " ");
    }
    System.out.println("");
}
```

Output

- Given the following **CSV** data file:

97452,James,88,96,82,86

99576,Alan,6,46,34,38

9888,Geoff,100,68,72,75

- The following would be the output of having **readFileExample()** call the method **processLine()**:

97452 James 88 96 82 86

99576 Alan 6 46 34 38

9888 Geoff 100 68 72 75

File Writing

Writing Text Files

- Writing files is simpler than reading them, no parsing required;
- The approach is the same:
  1. Open a **FileOutputStream**
  2. Create a **Writer**
  3. Output the data to the file using the **Writer**
- Careful: you must ensure the newlines and commas are in the correct place.
  - Assuming output is to a **CSV** formatted file.

PrintWriter

- `OutputStreamWriter` can be used for this:
  - `write()` method requires an array of bytes, ensure the data is copied into an array before writing.
- Easier if you could write to a file similar like printing messages in terminal:
  - You can, simply use a `PrintWriter` object;
  - Actually, `System.out` is an instance of a `PrintWriter`;
  - Thus writing to files can be identical to printing to the screen.
  - Remember to `close()` the file after finishing.

PrintWriter Example

- Following slide shows an example of writing to a file;
- Assumptions:
  - students marks were read in from the **CSV** file earlier;
  - one line of data is being saved to a new **CSV** file;
  - commas are inserted between the fields, with no spaces added.

Writing a CSV Row

```
private static void writeOneRow(String pFilename, int pID, String pName, double pAssign,
                                double pTest, double pExam, double pOverall)
{
    FileOutputStream fileStrm = null;
    PrintWriter pw;
    try
    {
        fileStrm = new FileOutputStream(pFilename);
        pw = new PrintWriter(fileStrm);
        pw.println(pID + "," + pName + "," + pAssign + "," + pTest + "," + pExam + "," + pOverall);
        pw.close();
    }
    catch(IOException e)
    {
        System.out.println("Error in writing to file: " + e.getMessage());
    }
}
```

Things Can be Different

- We explored the ‘long’ way to open files for I/O;
- Java does provide alternative methods for I/O;
- This way demonstrates how Java actually deals with File IO;
- Checkout Scanner for File I/O.

Primitive & Reference Variables

Reference Variables

- Creating an object, you must:
  - Declare a variable to be of the desired class;
  - Use the new operator to create an instance of that class;
  - The variable is set to the object's memory location;
  - It becomes a reference to the object ('it points to the object').
  - Example:

```
Frog froggie;
froggie = new Frog("Kermit");
froggie.leap();
```

Primitive vs. Reference: Variables

- Primitive Variables:
  - Direct access to the data;
  - Modifying the data involves modifying the contents of the variable;
- Reference Variables:
  - Indirect access to the data;
  - Modifying an object's data must be done via object methods;
  - In Java, reference variables contain an object's address.

Primitive vs. Reference Example

```
Frog froggie;
String frogName = froggie.getName();
froggie.setName("Kermit the Second");
```

```
Frog froggie;
Frog froggie2;
froggie = froggie2;
```

froggie now has the object address of froggie2

Java – Primitive vs. Reference: Parameters

- Supplying a primitive variable as a parameter to a method, the content contains the actual data, hence a copy of its contents is passed into the method.
- Supplying a reference variable as a parameter to a method, its contents contains the memory address of the data, hence a copy of the address is passed into the method.

Array Revision

- Arrays are variables containing many elements;
- Array elements are located sequentially in memory;
- Arrays can be initialised to any size;
- Array elements are accessed via an index or subscript.

```
int[] theArray = new int[10];
int[] anotherArray = {0,1,2,3,4,5,6,7,8,9};

int arrayLength = anotherArray.length;

for(int i = 0; i < arrayLength; i++)
{
    System.out.println(anotherArray[i]);
}
```

Passing and Returning Arrays

- Arrays can be passed as a parameter to a method;
- Passing an array does **NOT** copy it, a reference to the array is passed;
- If the method changes the passed-in-array, the ‘original’ array is actually changed;
- Arrays can also be returned from a method.

Passing and Changing Arrays: Java (1)

```
import java.util.*;
public class PlayingWithArrays
{
    public static void main(String [] args)
    {
        int [] myArray = {1,2,3,4,5,6,7,8,9,10};

        System.out.print("myArray before method call: ");
        printArray(myArray); //Call method to print the array

        System.out.println(); //Blank line for formatting

        playWithArray(myArray);

        System.out.print("myArray after method call:  ");
        printArray(myArray);//Call method to print the array

        System.out.println(); //Blank line for formatting
    }
```

Passing and Changing Arrays: Java (2)

```
    public static void playWithArray(int [] anArray)
    {
        anArray[9]=100;
        anArray[8]=800;
    }

    public static void printArray(int [] anArray)
    {
        int arrayLength = anArray.length;
        for(int i = 0; i < arrayLength; i++)
        {
            System.out.print(anArray[i] + " ");
        }
    }
}
```

Passing & Copying/Returning Arrays: Java (1)

```
public class PassingCopyingReturnArrays
{
    public static void main(String [] args)
    {
        int [] anArray = new int[3];
        int [] copyOfAnArray;

        anArray[0] = 1;
        anArray[1] = -16;
        anArray[2] = 5;

        LectureFiveSlide33.printArray(anArray);//Call method to print the array

        System.out.println(); //Blank line for formatting

        copyOfAnArray = copyIntArray(anArray);

        LectureFiveSlide33.printArray(copyOfAnArray);//Call method to print the array

        System.out.println(); //Blank line for formatting
    }
```

Passing & Copying/Returning Arrays: Java (2)

```
    public static int[] copyIntArray(int[] arrayToCopy)
    {
        int[] dupArray;
        dupArray = new int[arrayToCopy.length];

        for(int i = 0; i < arrayToCopy.length; i++)
        {
            dupArray[i] = arrayToCopy[i];
        }
        return dupArray;
    }
}
```