



**Murdoch**  
UNIVERSITY

# Topic 5: Java – Streams and Persistence

ICT373: Software Architectures

# Recap

- Java overview
- Objects
- Java revision
- O-O design and Unified Modelling Language (UML)

# Overview

- Streams, Input, Output
- Persistence and Serialization
- RTTI
- Passing, Returning and Cloning Objects

## **Reading:**

Textbook (11<sup>th</sup> Ed.) Chapters 15, 17

# Objectives (1/2)

- Explain the concept of a stream and describe the main types of stream available in Java.
- Give an example of how streams can be used for -
  - (i) buffering input, (ii) implementing pipes,
  - (iii) extracting tokens, (iv) compressing data, and
  - (v) archiving classes.
- Describe the decorator pattern approach to the creation of streams.
- Explain the concept of **Random Access file**, and how this is implemented in Java.
- Explain the concept of persistence of objects and how this is implemented using **Serialization**. Give some uses for it.

# Objectives (2/2)

- Explain the use of RTTI and the **Class** object to determine the types of serialized objects.
- What is meant by Reflection and how is it used?
- Explain the method used by functions to pass objects as parameters and return values.
- Explain the uses and dangers of aliasing.
- Explain whether Java uses pass by value or pass by reference.
- How is a **clone** (a deep-copy) of an object created?
- Explain, using the String class as an example, the difference between mutable and immutable objects.



**Murdoch**  
UNIVERSITY

# Streams



**Murdoch**  
UNIVERSITY

# Streams, Input and Output

- Java uses streams for I/O. A stream is just a source or sink of bytes (generally external to a program).
- Managing I/O is complicated because:
  - there are many combinations of choices to make about:
    - ultimate source or destination
    - buffering
    - formatting (e.g., sequential, random, binary, character, by lines, by words)
  - there are many and varied types of source/destination:
    - The Console: `System.in`, `System.out`
    - arrays or Strings (byte array streams)
    - pipes to and from other parts of the program (piped streams)
    - files
    - network connections

# Streams, Input and Output


- In the change from Java 1.0 to Java 1.1, IO was internationalized (many new alphabets were added via the Unicode system) and this affected all basic IO facilities.
- formatting of output was initially neglected but has got better (e.g. with the availability of `printf` and `format` methods and the `Formatter` class).



# Stream

- **Stream:** is an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.) and delivers it to your program.
  - it acts as a buffer between the data source and destination
- **Input stream:** a stream that provides input to a program, eg `System.in`
- **Output stream:** a stream that accepts output from a program, eg `System.out`
- A stream connects a program to an I/O object
  - `System.in` connects a program to the keyboard
  - `System.out` connects a program to the screen

# java.io package

- The java.io package contains a collection of stream classes.
- The stream classes are divided in two class hierarchies
  - Byte streams: to read and write binary data
  - Character streams: to read and write textual info
- **The Base Classes in java.io package:**
- (abstract superclasses)
- **Byte-based super classes:**
  - Programs use *byte streams* to perform input and output of 8-bit bytes.
  - **InputStream** – an abstract superclass that contains the basic methods (read, skip, mark, reset, close, etc) for reading raw bytes of data from a stream
  - **OutputStream** – sends raw bytes of data to a target such as the console or a network server (methods include: write, flush, , close)

# java.io package

- **Character-based classes:**

- The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.
- **Reader** - an abstract superclass for reading character-based input. It has basic methods deliberately similar to the methods of InputStream class.
- **Writer** - an abstract superclass for writing out character-based output. It has basic methods deliberately similar to the methods of OutputStream class.

# Byte Stream Subclasses

- Derived from the abstract classes **InputStream** and **OutputStream**
- Typically used to read and write binary data such as images and sounds

BufferedInputStream	BufferedOutputStream	ByteArrayInputStream
ByteArrayOutputStream	DataInputStream	DataOutputStream
FileInputStream	FileOutputStream	FilterInputStream
FilterOutputStream	ObjectInputStream	ObjectOutputStream
PipedInputStream	PipedOutputStream	PrintStream
PushbackInputStream	SequenceInputStream	
LineNumberInputStream	StringBufferInputStream	

<http://docs.oracle.com/javase/7/docs/technotes/guides/io/io.html>

# Character Stream Subclasses

- Derived from the abstract classes **Reader** and **Writer**

BufferedReader	BufferedWriter	CharArrayReader
CharArrayWriter	FileReader	FileWriter
FilterReader	FilterWriter	InputStreamReader
LineNumberReader	PipedReader	PipedWriter
PrintWriter	PushbackReader	StringReader
StringWriter		

- Other streams from java.nio, java.util.zip, java.util.jar, java.security, and javax.crypto packages are also available. See Java API documentation.

<http://docs.oracle.com/javase/7/docs/technotes/guides/io/io.html>

- You can wrap an InputStream and turn it into a Reader by using the InputStreamReader class.

# Files: Why?

- Use files on disk to store data which is:
  - needed before or after program runs
  - needs to be transported
  - too large to be handled by a program all at once
  - needed several times when you don't want to type it into your program more than once
- All files (data and programs) are stored as 0s and 1s but there are two general types of encodings which you choose between depending on your purposes.
- A stream is like a real stream. It is a flow of bytes.
- A file may be the 'source' of a stream or the 'destination' of a stream.

# Binary vs Text Files

- Binary Files,
  - different types of values coded differently to maximize efficient use of space (eg, each integer takes 4 bytes)
  - can only be written and read by programs (eg Java programs) which know the types of values being stored
  - are transportable (esp. in Java)
  - Slow/less efficient (byte-at-a-time read/write)
- Text Files,
  - stores characters, one at a time, (2 bytes each)
  - not efficient (storage space) for other types of values (eg, integer 1832, stored as "1832" takes 8 bytes)
  - can be written, read and edited by programs and text editors
  - are very transportable (eg send by email)
  - Fast/efficient (buffer-at-a-time read and write)

# Open – Loop – Close

- I/O in Java consists of
  - OPENING = creating a stream object for each input source or output destination and associating the object with the external entity
  - LOOPING getting values in or sending values out by calling methods on the stream object and then
  - CLOSING the file or connection by calling a close method on the stream.
- Open once. You will need to create a stream object and say what external entity it corresponds to. In doing the main work of the program, just refer to the stream object.
- At the end make sure that you close the stream.
- There are different classes (available in **java.io package** and the subpackages of **java.nio**) of stream objects appropriate to the task.



# Classes Scanner and Formatter

- (in java.util package)
- The class Scanner can be used to easily read data from a text file, eg
  - `Scanner input= new Scanner(new File("data.txt"));`
  - `System.out.println(input.nextLine());`
- The class File is useful for retrieving information about files and directories from disk
- Objects of class File do not open files or provide any file-processing capabilities
- The class Formatter can be used to easily write data to a text file, eg
  - `Formatter output = new Formatter("data.txt");`
  - `output.format("%s", "Hello class");`
  - `output.close(); //output.flush();`
- A Formatter object outputs formatted strings to the specified stream using the same formatting capabilities as the System.out.printf method.

# The Decorator Pattern Approach

- Because of all the combinations of choices needed in managing a particular IO operation, the Java IO library designers chose to use a general pattern of design called the **decorator pattern**. This allows a basic stream object (input or output) to be successively decorated by layers of extra facilities.
- Here's an example (for output to a binary file):

```
DataOutputStream outputStream =  
    new DataOutputStream(  
        new FileOutputStream("numbers.dat"));
```

- Here the file name is fed to the FileOutputStream constructor to make a new FileOutputStream object. This object is fed to the DataOutputStream constructor and hence we get our DataOutputStream.
- DataInputStream/DataOutputStream enables us to perform I/O for primitive type values and strings.

```
public void viewFile(String fileName){

try {
    DataInputStream is = new DataInputStream(new FileInputStream(fileName));
    System.out.println("Here are the Strings stored in " + fileName + ", one per line.");
    String s;

    try{
        do {
            s = is.readUTF();
            System.out.println(s);
        } while (true);
    } catch (EOFException e){ }

    is.close();

    System.out.println("That was the contents of " + fileName);
} // end outer try block

catch (FileNotFoundException e){
    System.out.println("File "+fileName+" not found.");
}

catch (IOException e){
    System.out.println(" An output problem.");
}

} //end of viewFile
```

# File I/O Reference Table

	Constructors	I/O	End of File
Binary In	new DataInputStream(new FileInputStream("name"))  <b>Alternatively,</b> use ObjectInputStream class	n=is.readInt() s=is.readUTF() f=is.readFloat() d=is.readDouble()	EOFException
Binary Out	new DataOutputStream(new FileOutputStream("name"))  <b>Alternatively,</b> use ObjectOutputStream class	os.writeInt(n) os.writeUTF(s) os.writeFloat(f) os.writeDouble(d)	N/A
Text In	new BufferedReader(new FileReader("name"))  <b>Alternatively,</b> input=new Scanner(new File("name"))	s=br.readLine()  int n=br.read()  s=input.readLine() etc.	s==null (not "") n==-1 NO EXCEPTIONS!  input.hasNextLine() etc.
Text Out	new PrintWriter(new FileOutputStream(name))  <b>Alternatively,</b> output=new Formatter("name")	pw.println(s)  output.format ("%s", str)	N/A  N/A

	Constructors	I/O	End of File
Keyboard In	stdin=new BufferedReader(new InputStreamReader(System.in))  <b>Alternatively,</b> input=new Scanner(System.in)	s=stdin.readLine() int n=stdin.read()  s=input.nextLine() int n=input.nextInt() double d=input.nextDouble() and other similar methods	s==null etc.  input.hasNextLine() input.hasNext() input.hasNextInt() input.hasNextDouble() etc.
Screen out	just use System.out.print or, System.out.println		

# Random Access Files (1/4)

- Random access files allow non-sequential access to a file's contents. To access a file randomly involves opening the file, **seeking** a particular location, and reading from or writing to that file.
- The `java.io.RandomAccessFile` class implements a random access file
- Unlike the input and output stream classes in `java.io`, `RandomAccessFile` is used for both reading and writing files
- You create a `RandomAccessFile` object with different arguments depending on whether you intend to read or write

# Random Access Files (2/4)

- RandomAccessFile input = new  
RandomAccessFile("myfile.txt", "r");
- RandomAccessFile update = new  
RandomAccessFile("oldfile.txt", "rw");
- A random access file consists of a sequence of bytes. A special marker called **file pointer** is positioned at one of these bytes.
- A read or write operation takes place at the location of the file pointer. When a file is opened, the file pointer is set at the beginning of the file. When you read or write data to the file, the file pointer moves forward to the next data.

# Random Access Files (3/4)

- In addition to the normal file I/O methods (eg, read/write methods for writing text and binary files such as `readInt()`, `readDouble()`, `readLine()`, `writeInt()`, `writeDouble()` etc) that implicitly move the file pointer when the operation occurs, `RandomAccessFile` contains three methods for explicitly manipulating the file pointer:
- `int skipBytes(int)` - moves the file pointer offset forward the specified number of bytes
- `void seek(long)` - positions the file pointer offset (from the beginning of the file) just before the specified byte
- `long getFilePointer()` - returns the current byte location of the file pointer (i.e. the current offset from the beginning of the file)

# Random Access Files (4/4)

Other methods include:

- `long length()` - returns the length of the file.
- `setLength (Long newLength)` – sets new length of this file
- `read(byte[] b)` - reads up to `b.length` bytes of data from this file into an array of bytes
- `write(byte[] b)` - writes `b.length` bytes of data from the specified byte array to this file, starting at the current file pointer. See the Java API documentation for further details.



# Other Streamy things

- **local directories**: to get a list of Strings being the names of the files in a particular directory, see the File class (**java.io.File**) and its list() method. Some useful methods of File class include canRead(), canWrite(), exists(), isFile(), isDirectory(), getName(), getPath().
- **redirection of standard input/output**: System.in is an InputStream object being the standard input. To use it, decorate it with buffering. System.setIn(X) allows redirection of standard input stream so it comes from a given InputStream X. Similarly System.setOut (Y) reassigns the standard output stream to PrintStream Y. See the System class Java documentation for further details. Also see the Scanner class (jdk1.5 and later)

```
System.setIn(new FileInputStream(new File("input.txt")));  
...  
//read from file  
....  
  
System.setOut(new PrintStream(new File("filename.txt")));  
System.out.println(sum); // will be printed to the file
```

# Other Streamy things

- **Deprecation**: Note that while using the IO facilities you might accidentally use Java 1.0 library classes. You may get a warning from the compiler that such a class is “deprecated”. This means that it is thought likely that that code will no longer work in future versions of Java.
- **tokenizers**: StreamTokenizer class can be used to split an InputStream up into its separate words (it has a nextToken() method). There is a similar StringTokenizer class.
- **Pipes**: Pipes are synchronized communication channels between threads or processes. You can create byte-based pipes between objects in your program using PipedInputStream and PipedOutputStream, or char-based pipes using PipedReader and PipedWriter.
- **line numbering**: use the decorator class LineNumberReader to keep track of where you are up to. You can call getLineNumber() and setLineNumber(int) methods.

# Other Streamy things

- **Compression**: you can add decorators to allow your program to directly read or write compressed data, i.e. in a Zipped or Gzipped format. See **java.util.zip** package documentation for `ZipOutputStream`, `ZipInputStream`, `GZipOutputStream`, `GZipInputStream` classes.
- **jars**: JAR (Java ARchive) file format is another compression format which, like Zip, allows many files to be compressed into one compressed file. JAR is platform independent.
  - You can compress any files into Jars if you want but the main use is for Java internet programming.
  - Jarring allows all the class files needed for a particular applet to be collected in one server request from a browser.
  - The classes for this are in the **java.util.jar** package.

# java.nio.file package

- Java SE 7 introduced the java.nio.file package which provides extensive support for interacting with files and directories. Its key entry points are the 3 classes:
  - The **Path** class has methods for manipulating files paths (eg creating a path, retrieving info about a path, comparing two paths etc).
  - The **Files** class has methods for file operations, such as moving, copying, deleting files, and also methods for retrieving and setting file attributes.
  - The **FileSystem** class has a variety of methods for obtaining information about the file system.

# java.nio.file package

- The java SE 7 Path and Files classes are much more convenient to use than the File class which is available since jdk1.0.
- Whilst File class provided both file location and file system operations, the new API splits this into two. Path class provides just a file location and supports additional path related operations. Files class supports file manipulation with a number of new methods not available in File.
- See the java SE 8 API documentation for further information. Also, see the online tutorial at the following site: <http://docs.oracle.com/javase/tutorial/essential/io/fileio.html>



**Murdoch**  
UNIVERSITY

# Persistence



**Murdoch**  
UNIVERSITY

# Serialization

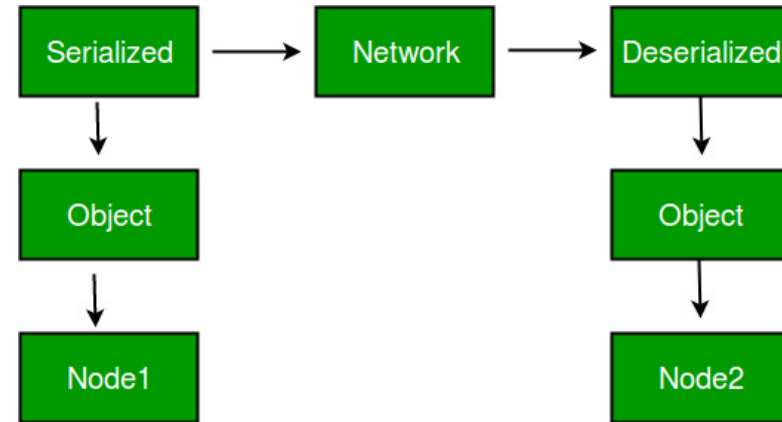
- The process of reading and writing objects is called **object serialization** (turning an object into a sequence of bytes).
- This allows an object (at a particular moment) to be frozen and kept or sent somewhere (eg, across a network) and then later restored to its previous state.
- We say that serialization allows the implementation of ***persistent*** objects, i.e. objects which can live/survive outside of programs.
- Java's serialization is only said to provide *lightweight* persistence because the programmer (as opposed to the system) has to do some of the work.

# Serialization – why important?

- RMI= remote method invocation, i.e. sending messages (method calling) to objects on other machines – involves communication between objects via sockets
- Cross JVM Synchronization: Serialization works across different JVMs that may be running on different architectures.
- Java Beans = objects, usually GUI components, which can be easily put into a program and immediately sprung to life.
- An object is serializable only if its class implements the **Serializable** interface.

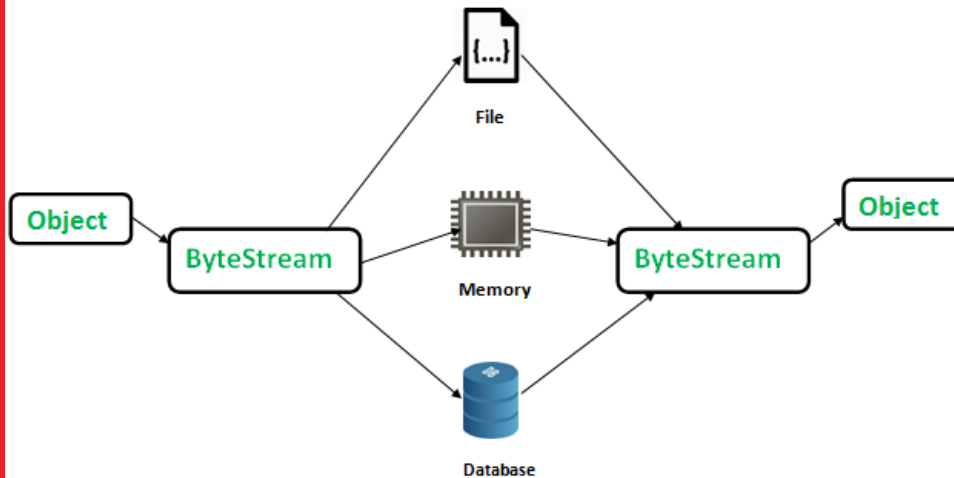


# Serialization (cont.)



Serialization

De-Serialization



# Serialization – how?

- (write to an ObjectOutputStream)

1. Declare that the class of the object, O say, implements Serializable (this is easy as this interface has no methods – it is a marker interface and enables the java serialization mechanism to automate the process of storing objects)

```
import java.io.*;  
  
public class MyClass implements Serializable{  
  
.....
```

2. Take the desired OutputStream and decorate it to an ObjectOutputStream to get X, say.

```
ObjectOutputStream X = new  
    ObjectOutputStream(new  
        FileOutputStream("myObjectFile.dat"));  
.....
```

# Serialization – how?

3. call `X.writeObject(O);`

.....

```
MyClass O = new MyClass();
```

.....

```
X.writeObject(O);
```

```
X.writeObject("13 April 2015"); // write a string object
```

.....

```
X.close();
```

.....

# How to de-serialize

(read from an ObjectInputStream)

1. Decorate the InputStream to an ObjectInputStream, Y.

```
ObjectInputStream Y = new ObjectInputStream(new  
FileInputStream("myObjectFile.dat"));
```

2. call `var = Y.readObject();` Eg,

```
MyClass var1 = (MyClass)Y.readObject();
```

```
String var2 = (String)Y.readObject();
```

# How to de-serialize

- **Note 1:** The objects must be read from the stream in the same order in which they were written.
- **Note 2:** what you get back from `readObject()` is just an `Object`. To use it you usually will need to *downcast* it to a more specific type. This means that the de-serializing program must have access to the class.
- **Note 3:** serializing automatically takes care of all the objects which are part of the serialized object and their parts and their parts etc (but their classes must all implement `Serializable`).
  - All those objects are serialized too and encoded in the byte stream for the big object so that it can be put back together as good as new.
  - The encoding is good enough to allow full restoration of the object without any need for the use of a constructor.

# Controlling/Customising Serialization

- With a Serializable object, all serialization happens automatically.
- Sometimes you need finer control (e.g., do not want to serialize portions of an object due to security issues – passwords etc) ...
- Implement the Externlizable interface (instead of the Serializable interface) and supplying writeExternal() and readExternal() methods which will be (automatically) called during serialization and deserialization.
  - Then nothing is automatically serialized and you can explicitly serialize only the necessary parts inside the writeExternal() method.

```
public class MyExternalizable implements Externalizable
{

    private String userName;
    private String passWord;
    private Integer roll;

    public MyExternalizable()
    {

    }

    public MyExternalizable(String userName, String passWord, Integer roll)
    {
        this.userName = userName;
        this.passWord = passWord;
        this.roll = roll;
    }

    @Override
    public void writeExternal(ObjectOutput oo) throws IOException
    {
        oo.writeObject(userName);
        oo.writeObject(roll);
    }

    @Override
    public void readExternal(ObjectInput oi) throws IOException, ClassNotFoundException
    {
        userName = (String)oi.readObject();
        roll = (Integer)oi.readObject();
    }
}
```

# Controlling/Customising Serialization

- Another way to control automatic Serialization is to have those parts of your object that are not to be serialised declared as **transient**. Then they will not be encoded during deserialization. (This is useful for secret parts.). Eg

```
public class MyClass implements Serializable{  
    .....  
    private Date today = new Date();  
    private String username;  
    private transient String password;  
        // password will not be stored on disk because it is  
        // declared transient.  
    .....  
}
```



# Serialization of several objects

- Make sure that you do this into one stream and at one moment of time (unless you have good reason not too).
- This allows reconstruction of shared parts to be done properly.
- In fact, it is a good idea (in the case of wanting to serialize a whole bunch of objects), to put them all in a collection (or some other big object) and just serialize that with one `writeObject()` call.
- Eg, an array is serializable if all its elements are serializable. In this case an entire array can be saved into a file using one `writeObject()` call, and later restored using one `readObject()` call.
- Thus serialization enables us to save a collection of objects to a disk file and retrieve it exactly as we stored it.

# RTTI = Run-Time Type Identification

- finding out more specific information about the type of an object as the program runs.
  - Suppose class A is a superclass of class B along with possibly some other different classes. Suppose we have an A type variable referring to some object. If you have some code that determines that the object is really a B type object then that is doing RTTI,
  - i.e., finding the exact type of an object when you have only a reference to the base type
- We can just downcast the object to the B class and catch any errors. A run-time error will be thrown if the object was not actually a B object.

# RTTI using the Class object

- To get a reference to a **Class** object (eg, for the class B), either call `Class.forName("B")` eg,  
`Class t = Class.forName("B");`  
`forName()` is a static method of `Class`. It takes a string containing the name of the particular class.
- Or use `B.class`; (eg, `Class t = B.class`;)
- These both return a reference for the `Class` object, if it has already been created in your program, or if the definition of B is available to your program. In the latter case the B `Class` object is created.
  - The `Class` object can then be queried for useful runtime information.
  - The 2<sup>nd</sup> method is best to use as the mention of class B is then checked at compile time.

# RTTI using instances of Classes

- There are two ways to perform RTTI using this form. Say we have an instance variable X and we want to determine whether it refers to an object of type B (i.e. belonging to a subclass B). You can

1. use the keyword *instanceof* which returns a boolean answer depending on whether an object is an instance of a particular type, eg

```
if ( X instanceof B )
```

```
((B)X).doSomething();    // call B method
```

- Note that you must explicitly mention the class B here.

2. ask ( X instanceof B ) for the same answer, if X is a variable of Class type currently referring to the class object of type B. This allows much more flexible code.

# RTTI using instances of Classes

- *Uses of this test:* do such a test just before you are about to downcast but are not sure of the exact type.
- *Other uses of Class objects:* there are several, eg,  
    `CVar.newInstance()`  
    creates a new object of the class currently referred to by Cvar.  
    You can also get lists of class methods etc.
- Finally, (as already mentioned) you can call  
    `var.getClass();`  
    to get a reference for the actual class of the object currently referred to by a variable.

# RTTI Limitation

- RTTI mechanism assumes that all the relevant classes are available at compile time. Increasingly there are situations in which this is not true:
  - In *Rapid Application Development* (RAD) tools like the Java Beans setup, GUI components can be picked up from various sources (eg, over the web) and dropped by the user onto the appropriate place on a form which they are designing. The RAD tool program has to find out about the new component so it can connect it into the working form. The RAD tool needs to find out method, field and constructor information about an object of a completely new class (to it).
  - In *Remote Method Invocation* (RMI), a program may want to create and execute some objects at a remote location (across a network connection, eg in distributed computing). The class information about those objects may only be kept on the remote machine (so that it is up-to-date) and so will only be completely known at run-time. Uses of RMI include keeping central repository of up to date class libraries for specific domains, or delegating computing to various types of computers that may be more appropriate.
  - Components based programming
  - Distributed computing

# Reflection (1/2)

- Java provides a mechanism, called **Reflection**, to overcome RTTI limitation. So, how to find out information about “alien” objects?
- determining class information (e.g. fields and methods) of objects completely at run time (objects whose classes may have been unavailable at compile time).
- This is done by allowing programs to proceed with alien objects (and get specific class information) as long as the program can get hold of the appropriate Class object (i.e. the .class file) when/if it is needed.

# Reflection (2/2)

- Reflection is supported in Java by
  1. methods in the Class class; and
  2. a specific library, **java.lang.reflect** including classes Field, Method and Constructor. (See Java documentation for details.)

Eg, ***Method[] m = X.getMethods();***

- will return an array of Methods being those belonging to the class whose Class object is referred to by X.
- You can extract useful information about the methods.
- It is particularly useful for programmers to be able to use such facilities when developing classes using inheritance: the Java source file containing the definition of such a class, recall, will only contain the new or overridden methods in the new class.
  - Reflection can be used to find out about all the methods belonging to a class.



# Don't use RTTI ...

- ... very often. You do not need to reflect, check instanceof or even downcast in solving most problems.
- Sometimes you need to know more information about a type than the information provided by the class Class, In this case, use the classes from the package java.lang.reflect.
- Use of RTTI might suggest that you are not using polymorphism correctly.
- A good design will have a place for general code and superclass references and will let dynamic binding handle the places where methods need to distinguish between subclasses.
- A bad design might use switch statements and RTTI to accomplish the same ends. The badly designed code will take a lot more work to update to accommodate changes in the implementation of subclasses or changes in the range of subclasses available.

# Don't use RTTI ...

You may have to use RTTI, if

- you cannot add desired methods to the superclass.
- handling one particular subclass causes efficiency problems
- class information is not available at compile time.



**Murdoch**  
UNIVERSITY

# Cloning



**Murdoch**  
UNIVERSITY

# Passing, Cloning and Returning Objects

- In Java, multiple references to the same object can result from:
  1. using =, eg `x=y`;
  2. passing arguments to methods (eg, calling `z.meth(x)` for a method `meth(y)` results in the parameter `y` referencing the same object as `x` does).
- Having different variables with the same reference is sometimes called **aliasing**.
  - Aliasing is dangerous. Modifying an object via one reference to it might surprise the users of other references to it.
  - Allowing a method to modify its arguments, such as `x` in the example above, is unusual code design - such a change is called a **side-effect** of the method call.

# Passing by Value or by Reference

- Does Java use pass by value or pass by reference?
  - *Answer:* it depends.
  - *Real Answer:*
- Consider an argument x being passed to a method parameter y in a method call.
  - primitives are passed by value: changing a primitive parameter does not change the argument.
  - objects are passed by reference: the parameter is a reference to the original object.
  - references to objects are passed by value: making y reference a completely different object will not change x's referencing.
  - So it is a bit complicated and therefore a question which can lead to much discussion. The important thing is to be able to understand what is really going on when a method is called.

# Cloning (1/2)

- Clone() is a method for object duplication.
  - Local copy of an object;
  - reserves some (heap) memory for the new object (of the same type as the original)
  - copy the values of each field from the original object to the copy object.
- You can do this yourself by writing a method for that class of objects which uses a constructor and some initialization statements (called **copy constructor**).

# Cloning (2/2)

- Or use the built-in **cloning** facilities.
  - declare that your class implements the **Cloneable** interface (this is an empty interface just acting as a flag to indicate that the class is designed to allow cloning).
  - override the **clone()** method (which is inherited from **Object**) by a **public clone()** method in your class (note `Object.clone()` is protected);
  - call **super.clone()** inside your **clone()** method as `Object.clone()` does the memory reservation and copying (incidentally without calling a constructor);
- If you don't publically override `clone()` then the user of your method will get a compile time error 'method not accessible error'. If you don't implement `Cloneable` then `Object.clone()` will throw a run-time error.

```

public class Fraction implements cloneable{
    private int numerator, denominator;
    Fraction() {numerator = 0; denominator = 1;}
    Fraction (int n, int d) {
        numerator = n; denominator = d;}
    public int getNumerator () {return numerator;}
    public int getDenominator () {return denominator;}

    public Object clone() {
        try { return super.clone(); }
        Catch (CloneNotSupportedException e){
            // this should not happen
            return null; // to keep the compiler happy
        }
    } // end method clone
}

```

// client code

Fraction myFrac = new Fraction (1, 2);

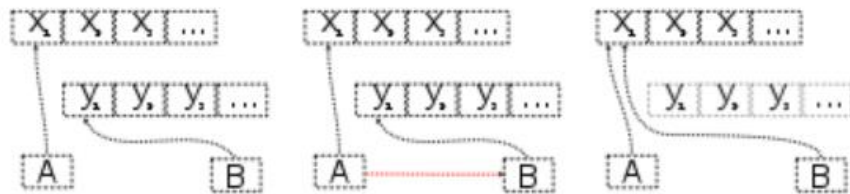
Fraction fracCopy = (Fraction)myFrac.clone();

// note the typecast (Fraction)

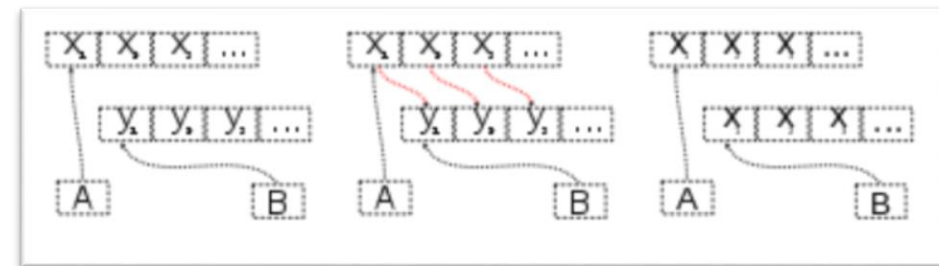


# Shallow and Deep copying

- Note that the above uses **shallow copy**. If the instance variables are all of the primitive types (as in the above case) and the String type, then this definition of clone will work fine.
- If the instance variables are of class type (object references), then **deep copying** would be required to make a complete copy.



Shallow copy



vs

Deep copy

# Shallow copying

- = two or more reference variables of the same type refer to (point to) the same object.
- If you just rely on `Object.clone()` or do your own copying by making component fields of your objects equal (say, by assignment), then you end up with a shallow copy. The immediate components (member fields) of both objects will refer to the same component objects.
- Suppose that there is a member field **score** of type **Marks** in your **Student** class. Suppose that you make a shallow copy of the **Student** object referred to by **x** and refer to it by **y**. Now suppose that you access the score of **y** and change it. Because **y** is only a shallow copy of **x** you will find that the score of **x** has also been changed.

# Deep copying (1/2)

- Each reference variable refers to its own object
- More often you need a deep copy.
  - Eg, to allow you to work on an argument given to a method without disrupting the original object.
- To do this you need to clone the object, clone all the component objects of the object, clone all their components, etc all the way down until you reach primitives.
  - To do this assumes that you have control over the code for all the classes involved or that you can trust some of them to do deep copies of themselves.
- Writing such code can be done in your own copying methods or, as before, by overriding the clone() method.

# Deep copying (2/2)

- In overriding `clone()` you need to call `Object.clone()` to get the right memory allocation for the main object. Then you need to downcast it to the right type (a reference to an `Object` comes back from `Object.clone()`).
- Then you need to go through each member field in turn, get a (deep) clone of the component object and use that as the corresponding component of the new object.
- When you ask the component object to be cloned you have to ensure that it follows the same procedure. And so on.

# Clone /Cloneable

- Why do we need to override clone() and implement Cloneable?
- clone() is protected in the Object class.
  - client programmers cannot just call clone() on the objects.
  - So the designer of new classes can assume that their objects are not cloneable unless they do something to allow that.
  - This is important with internet programming when there may be security issues.
- clone() is protected
  - it is available for designers of any class of objects for their own objects. This is because every class inherits from Object.
  - So every class can have a clone() method which uses Object.clone() on their own objects. However, to allow clients of your classes to call your clone() method you need to override Object.clone() by a public clone() method.

# Clone /Cloneable

- now every subclass of your class would inherit your public clone() method.
- Once again, there may be designers who want to inherit from your class but who do not want cloneability. That is why the **Cloneable** interface is added as an extra switch to allow/disallow cloning.
- Note that checking Cloneability is done in Object.clone() with the risk of a run-time error.

# Possible Alternative Approaches to deep cloning: Serialization

- It is possible to serialize an object into a stream (eg, into a buffer or file) and deserialize the stream back into an object. This will result in a deep copy of the object. It is easy to code this approach. However, (as shown in the text book - chapter 17) this is a very slow way of cloning.
- **A Copy Constructor:** Another alternative is to follow the C++ copy constructor approach. Eg, given a reference b1 to a Banana, you might want clients to be able to write

**Banana b2 = new Banana(b1);**

and call on a Banana constructor which takes a Banana argument and uses knowledge about Bananas to (deep) copy field values across to the new Banana object.

# Copy constructor (cont.)

- This will work as long as you are happy to only copy pure bananas. This approach runs into trouble if you try to copy subclass objects like CarnarvonBananas or Plantains which might have extra fields. Unless you go to extra lengths to watch out for these possibilities and take care of them explicitly, the copy you will get will be just a copy of the general Banana fields.
- Also, the copy will be just a Banana unless you arrange for a CarnarvonBanana constructor to be called.
- So beware of this approach.



# Copy constructor: subclass

```
public Book(Book b)
{
    super(b);
    ISBN = b.ISBN;
    issued = b.issued;
    author = b.author;
}
```

And the copy constructor of `EducationalBook` :

```
public EducationalBook(EducationalBook b)
{
    super(b);
    edition = b.edition;
    speciality = b.speciality;
}
```

# Immutable Objects

- The objects in read-only classes are sometimes called ***immutable objects***.
  - Wrapper classes, e.g., Integers;
  - String class
- You can make your own by ensuring that all fields are *final* and that no public methods change the object.
- Class can be final as well.
- Sometimes you want objects which are mostly unchanged but occasionally need quite a lot of work done on them (eg, Strings).
  - If you used immutable objects here then sometimes you would have to create a lot of new immutable objects which are used once only and then leave them to be cleared up.

# Immutable Objects

- A good design is to make the class read-only but provide a **companion class** which can be modified.
  - The user simply switches to the companion class when work needs to be done on the object and then when the work is finished create a new immutable object from the final companion object.

# Strings and StringBuffers

- Strings are very susceptible to aliasing problems. Therefore Java designers decided to make them *immutable*: check the documentation.
- Eg, **s.concat("abc");** does NOT change s but, instead returns a new string (being s with the extra characters tacked on afterwards).
- However, this leads to the creation/destruction overhead when we want to do work with strings. Eg, using the overloaded + operator  
**String s = "abc" + "def" + t + Integer.toString(47);**
- might involve the creation of a number of **String** objects. Mentioning String constants (like "abc") results in the creation of a new String object.

# Strings and StringBuffer

- Java designers decided to use the mutable companion class solution to this problem.
- Strings are converted to **StringBuffer** to allow a series of gradual changes to produce a finished product which is then converted into a String.
- There are lots of generally useful methods belonging to these classes. Eg, see how to make a StringBuffer from a String, change one character and make a new String from the result.



**Murdoch**  
UNIVERSITY

# Summary



# Summary: Streams and Persistence

- This topic looked at the development of Streams for i/o and data transfer.
- It then looked at the use of streams to implement persistence of data and objects beyond the lifetime of a program.
- Finally it examined RTTI, Java parameter passing, aliasing, cloning and immutable objects.



**Murdoch**  
UNIVERSITY

