Java I/O

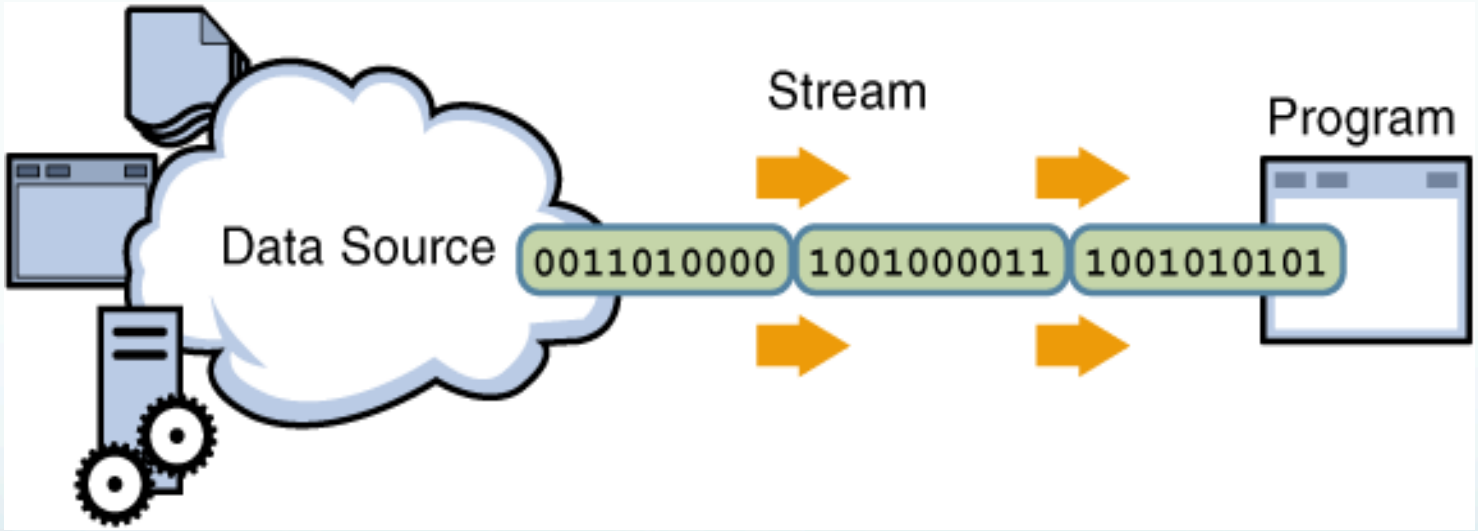
COMP3111 tutorial

Outline

- Most applications need to process some input and produce corresponding output
- I/O Streams in Java
  - Reference: http://docs.oracle.com/javase/tutorial/essential/io/streams.html
- Introduction of Xstream
  - An external library to handle XML
  - Learn how to set up an external library in Eclipse

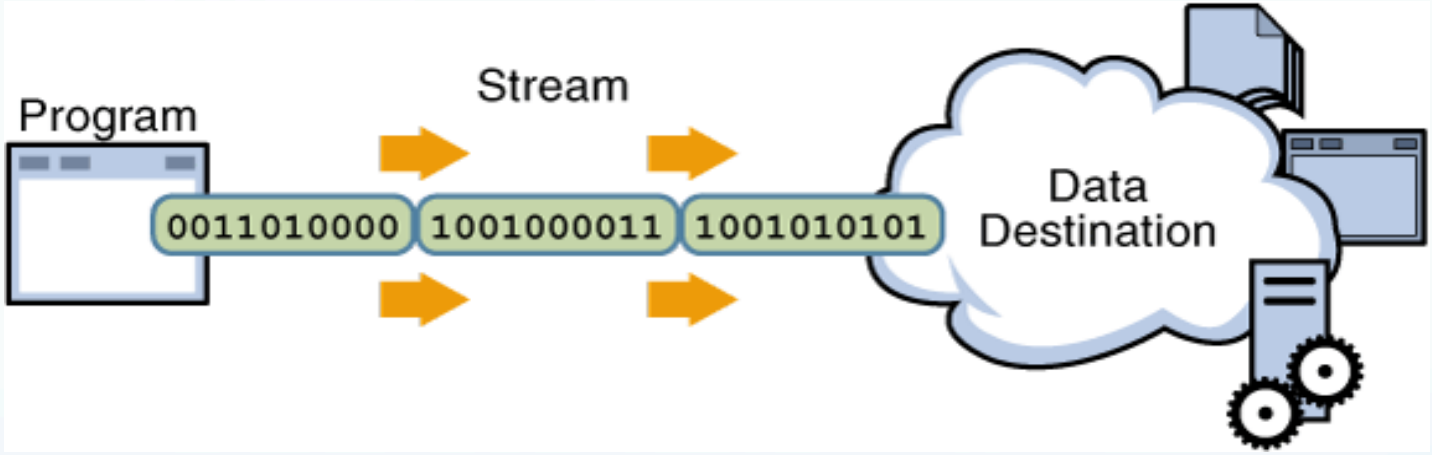
I/O Streams

- Input stream: Reading information into a program



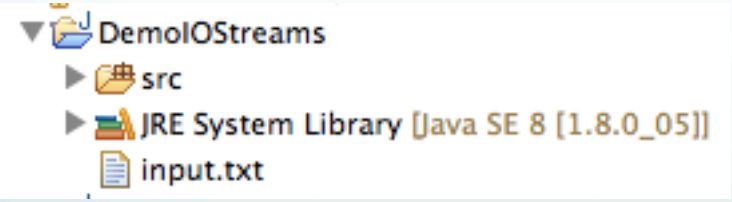
I/O Streams

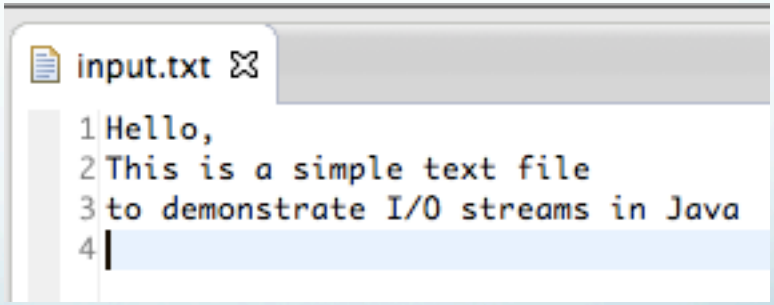
- Output stream: Writing information from a program



Setup: Java Project with a text file

- You need to first create a Java project
- Create a text file (e.g. input.txt)
  - The file must be outside the “src” folder
  - Type in a few sentences and save it (i.e. make sure that no * appears next to the filename)





Byte Streams

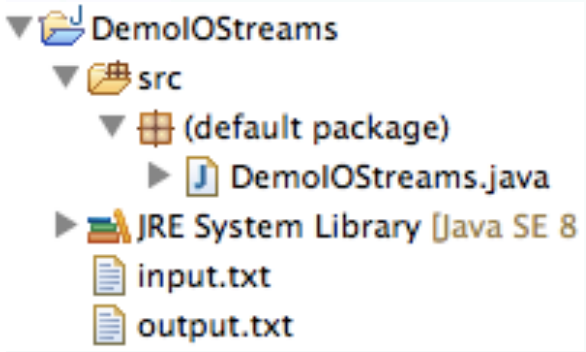
- Byte Streams are used to perform input and output of 8-bit bytes
- Abstract classes: InputStream and OutputStream
  - superclass of all classes representing an input/ output stream of bytes
- FileInputStream/FileOutputStream
  - Read/Write a file in a byte-by-byte manner
  - General purpose – suitable for any file type

Copy a file using Byte Streams

- Exercise:
  - Write a Java program to copy “input.txt” to “output.txt” using FileInputStream and FileOutputStream
- Most operations on Java I/O streams may throw IOException objects
  - There are 2 possible ways to handle:
    - Not handle it, throw it to an upper level
    - Use try-catch-finally blocks to handle the exceptions

Example: Without handling IOException

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class DemoIOStreams {
    public static void main(String[] args)
            throws IOException {
        // IOExceptions are not handled in this method
        FileInputStream in = null;
        FileOutputStream out = null;
        in = new FileInputStream("input.txt");
        out = new FileOutputStream("output.txt");
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
        // Remember to close the I/O streams
        in.close();
        out.close();
    }
}
```



Press F5 on the project icon if you are unable to see "output.txt"

Example: Adding Exception Handling

- Good practice: Close streams inside the finally block
  - It ensure that the resources (i.e. files in this example) will be closed
- Please note that the close operations within the finally block may also generate IOException, and they should be handled separately

```
public static void main(String[] args)  {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
        in = new FileInputStream("input.txt");
        out = new FileOutputStream("output.txt");
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            in.close();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        } // end of inner try-catch
    } // end of outer try-catch
} // end of main method
```

Character Streams

- 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 (i.e. ASCII => Unicode (Java program) => ASCII)
- Abstract classes: Reader/Writer
- Character streams file I/O: FileReader / FileWriter
  - Suitable for text files

Example: Using FileReader/FileWriter

- The program is exactly the same, except
  - FileInputStream is replaced by FileReader
  - FileOutputStream is replaced by FileWriter

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class DemoReaderWriter {
    public static void main(String[] args) throws IOException {
        FileReader in = null;
        FileWriter out = null;
        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                in.close();
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
            } // end of inner try-catch
        } // end of outer try-catch
    } // end of main method
}
```

Advanced: Forcing output encoding

- Other classes are necessary if you would like to force a particular output encoding method (e.g. UTF-16)
- Example:

```
in = new FileReader("input.txt");
//out = new FileWriter("output.txt");
out = new OutputStreamWriter(
        new FileOutputStream("output.txt"),"UTF-16");
```

- Unicode handling in Java is not perfect, especially when BOM (Byte order mark) is added
  - A known bug in Java: http://bugs.java.com/view_bug.do?bug_id=4508058

Result

- input.txt

Resource	
Path:	/DemoIOStreams/input.txt
Type:	File (Text)
Location:	/Users/cspeter/Documents/
Size:	69 bytes

- output.txt

Resource	
Path:	/DemoIOStreams/output.txt
Type:	File (Text)
Location:	/Users/cspeter/Documents/works
Size:	140 bytes

```
1 Hello,
2 This is a simple text file
3 to demonstrate I/O streams in Java
4
```

But… the content looks identical!
Character encoding can be very problematic

Buffered Streams

- Most of the examples we've seen so far use unbuffered I/O
  - Each read or write request is handled directly by the underlying OS
  - This can make a program much less efficient, since each such request often triggers disk access, that is relatively expensive
- To reduce this kind of overhead, the Java platform implements buffered I/O streams
  - Example: Using BufferedReader/BufferedWriter or BufferedInputStream / BufferedOutputStream

Example: Using Buffered Streams

- Simple. A program can convert an un-buffered stream into a buffered stream using the wrapping idiom

```
BufferedReader in = null;
BufferedWriter out = null;
try {
    in = new BufferedReader(new FileReader("input.txt"));
    out = new BufferedWriter(new FileWriter("output.txt"));
```

flush()

- What is flush()?
  - It only applies on buffered writer classes
  - Write out a buffer at critical points, without waiting for it to fill
  - No effect unless the stream is buffered
  - flush() at least once before closing the resource

Data Streams

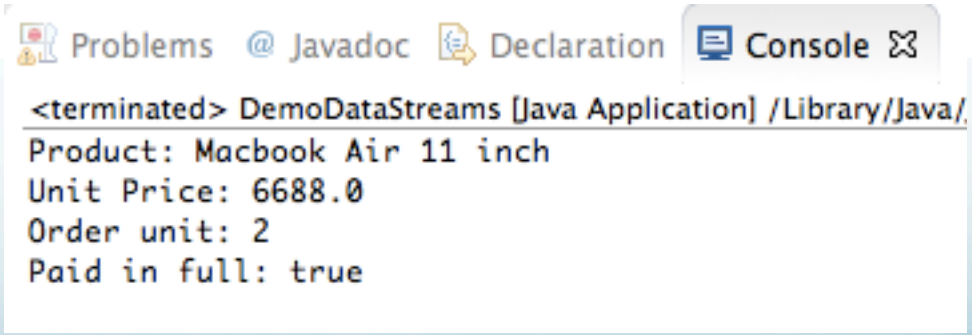
- Data streams support
  - binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double)
  - String values (with Unicode support)
- Interfaces: DataInput and DataOutput
- Classes: DataInputStream and DataOutputStream

Example: Using DataOutputStream

```
public static void main(String[] args)
 throws IOException { // IOException unhandled

    String product = "Macbook Air 11 inch" ;
    double unitPrice = 6688.0;
    int orderUnits = 2;
    boolean paidInFull = true;

    DataOutputStream dout = new DataOutputStream(
            new BufferedOutputStream(
                    new FileOutputStream("output_data.txt")));

    dout.writeUTF(product);
    dout.writeDouble(unitPrice);
    dout.writeInt(orderUnits);
    dout.writeBoolean(paidInFull);

    dout.flush(); // ensure all data is out
    dout.close();
```

Example: Using DataInputStream

```
DataInputStream din = new DataInputStream(
        new BufferedInputStream(
                new FileInputStream("output_data.txt")
                ));
System.out.println("Product: " + din.readUTF() );
System.out.println("Unit Price: " + din.readDouble());
System.out.println("Order unit: " + din.readInt());
System.out.println("Paid in full: " + din.readBoolean());
din.close();
```

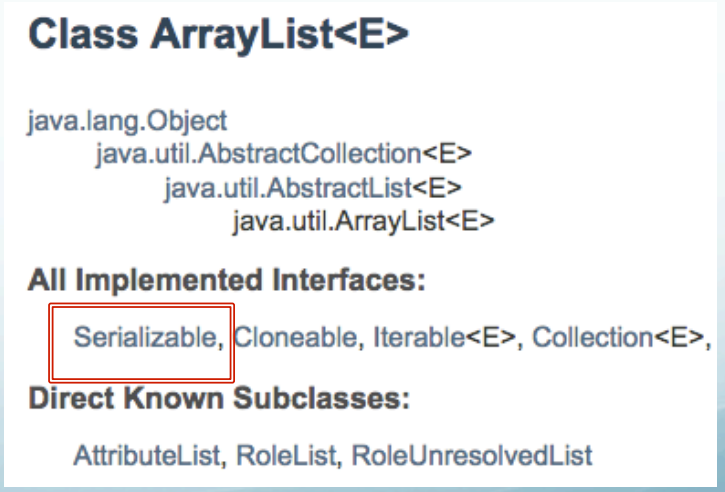


Object Streams (Serialization)

- Object streams support I/O of objects
- Most, but not all, standard classes support serialization of their objects
- If a class support serialization, it must implement the “Serializable” interface

Object Stream Classes

- Interfaces: ObjectInput and ObjectOutput
- Classes: ObjectInputStream and ObjectOutputStream
- ArrayList<E> implements “Serializable” and let’s use it as an exmaple



Example: Using ObjectOutStream

```
public static void main(String[] args)
        throws IOException { // IOException unhandled
    List<Integer> ls = Arrays.asList(1,2,3,4,5);

    ObjectOutputStream oos = new ObjectOutputStream(
            new BufferedOutputStream(
                    new FileOutputStream("test.tmp")));
    oos.writeObject(ls);
    oos.flush();
    oos.close();
```

Example: Using ObjectInputStream

- Type-casting is necessary and it will cause runtime error if the type doesn't match

```
ObjectInputStream ois = new ObjectInputStream(
        new BufferedInputStream(
                new FileInputStream("test.tmp")));
try {
    List<Integer> newList = (List<Integer>) ois.readObject();
    System.out.println(newList);
} catch (ClassNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Example: TransactionRecord class

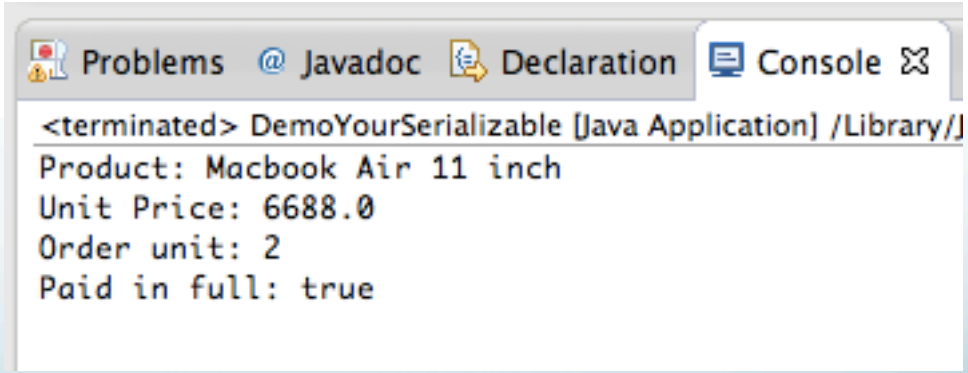
```
class TransactionRecord implements Serializable {
    public TransactionRecord(String p, double up,
            int ou, boolean paid) {
        product = p;
        unitPrice = up;
        orderUnits = ou;
        paidInFull = paid;
    }
    public void print() {
        System.out.println("Product: " + product);
        System.out.println("Unit Price: " + unitPrice);
        System.out.println("Order unit: " + orderUnits);
        System.out.println("Paid in full: " + paidInFull);
    }
    // All fields must be seralizable
    private String product ;
    private double unitPrice ;
    private int orderUnits ;
    private boolean paidInFull;
}
```

Example: Writing a transaction record

```
public static void main(String[] args)
throws IOException {
    String product = "Macbook Air 11 inch" ;
    double unitPrice = 6688.0;
    int orderUnits = 2;
    boolean paidInFull = true;

    TransactionRecord tr = new TransactionRecord(
            product, unitPrice, orderUnits, paidInFull);

    ObjectOutputStream oos = new ObjectOutputStream(
            new BufferedOutputStream(
                    new FileOutputStream("myobj.tmp")));
    oos.writeObject(tr);
    oos.flush();
    oos.close();
```

Example: Reading back a transaction record

```
ObjectInputStream ois = new ObjectInputStream(
        new BufferedInputStream(
                new FileInputStream("myobj.tmp")));
try {
    TransactionRecord newTr = (TransactionRecord) ois.readObject();
    newTr.print();
} catch (ClassNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```



Using “transient”

- Variables may be marked transient to indicate that they are not part of the persistent state
- Example: Adding a “total” amount. We may consider to re-calculate this field when reading and creating an object
- If readObject is not override, totalPrice will be zero (i.e. not saved)

```
// All fields must be seralizable
private String product ;
private double unitPrice ;
private int orderUnits ;
private boolean paidInFull;

// If a field is transient, won't save that field
private transient double totalPrice ;

// Override readObject to have some special handling...
private void readObject(ObjectInputStream inputStream)
        throws IOException, ClassNotFoundException
{
    inputStream.defaultReadObject();
    totalPrice = unitPrice * orderUnits;
}
```

Supplementary

Using an external library: Xstream for XML I/O

Other external libraries: XStream

- XStream is very useful to handle XML input/output
  - Before running this example, you need to set up the Xstream library first

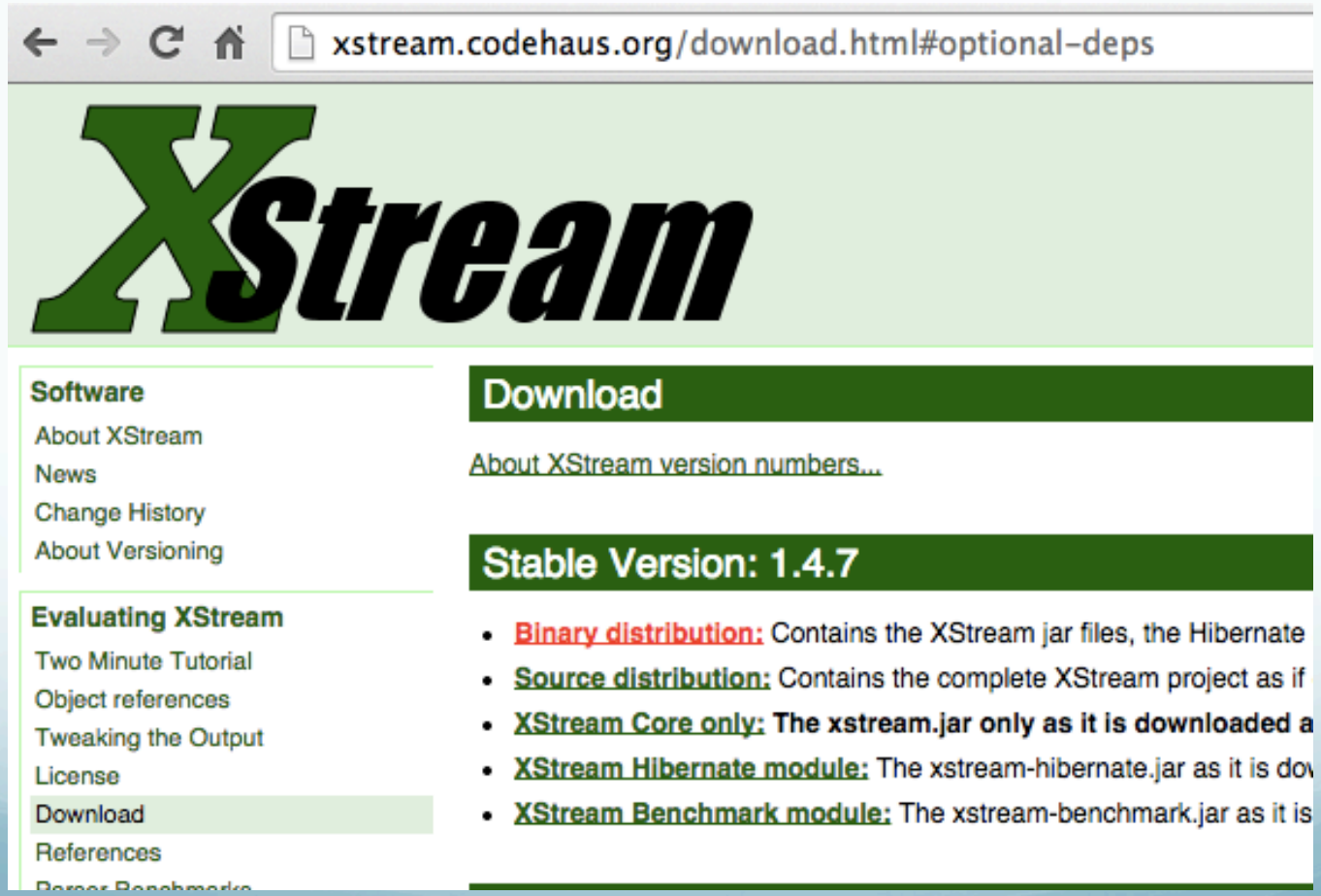
```
1  import java.io.*;
2  import com.thoughtworks.xstream.*;
3
4  public class DemoXStream {
5      public static void main(String[] args) {
6          XStream xstream = new XStream();
7          String product = "Macbook Air 11 inch" ;
8          double unitPrice = 6688.0;
9          int orderUnits = 2;
10         boolean paidInFull = true;
11         TransactionRecord tr = new TransactionRecord(
12                 product, unitPrice, orderUnits, paidInFull);
13
14         // Convert any serializable Java object as XML
15         String xml = xstream.toXML(tr);
16         System.out.println(xml);
17
18         // Recreate TransactionRecord
19         TransactionRecord newTr =
20             (TransactionRecord) xstream.fromXML(xml);
21         newTr.print();
22     }
23 }
```

Problems | Javadoc | Declaration | Console

```
<terminated> DemoXStream [Java Application] /Library/Java/Jav
<TransactionRecord>
  <product>Macbook Air 11 inch</product>
  <unitPrice>6688.0</unitPrice>
  <orderUnits>2</orderUnits>
  <paidInFull>true</paidInFull>
</TransactionRecord>
Product: Macbook Air 11 inch
Unit Price: 6688.0
Order unit: 2
Paid in full: true
```

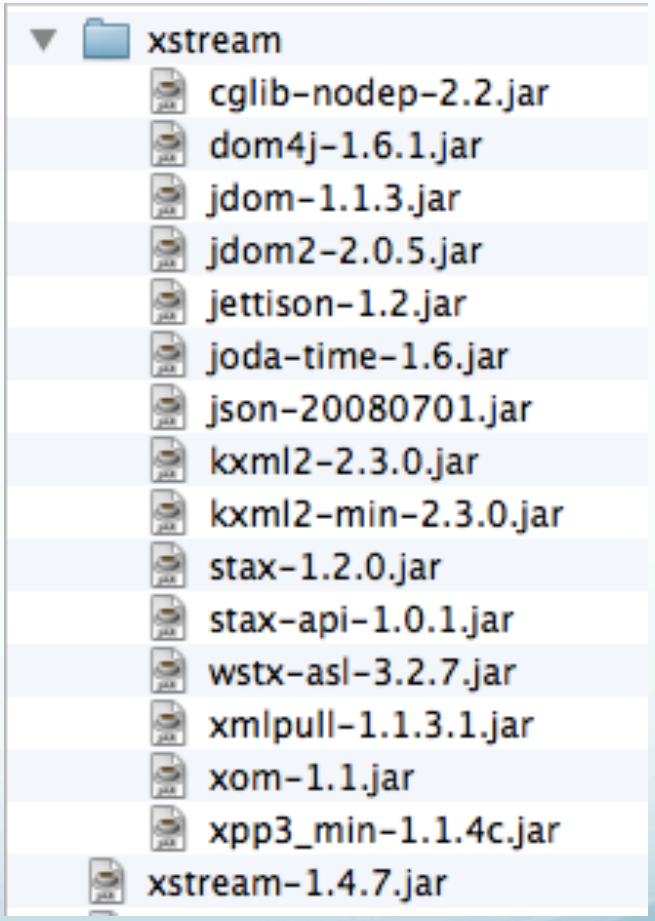
Setup: Download the Xstream library

- Choose binary distribution (with all dependencies)

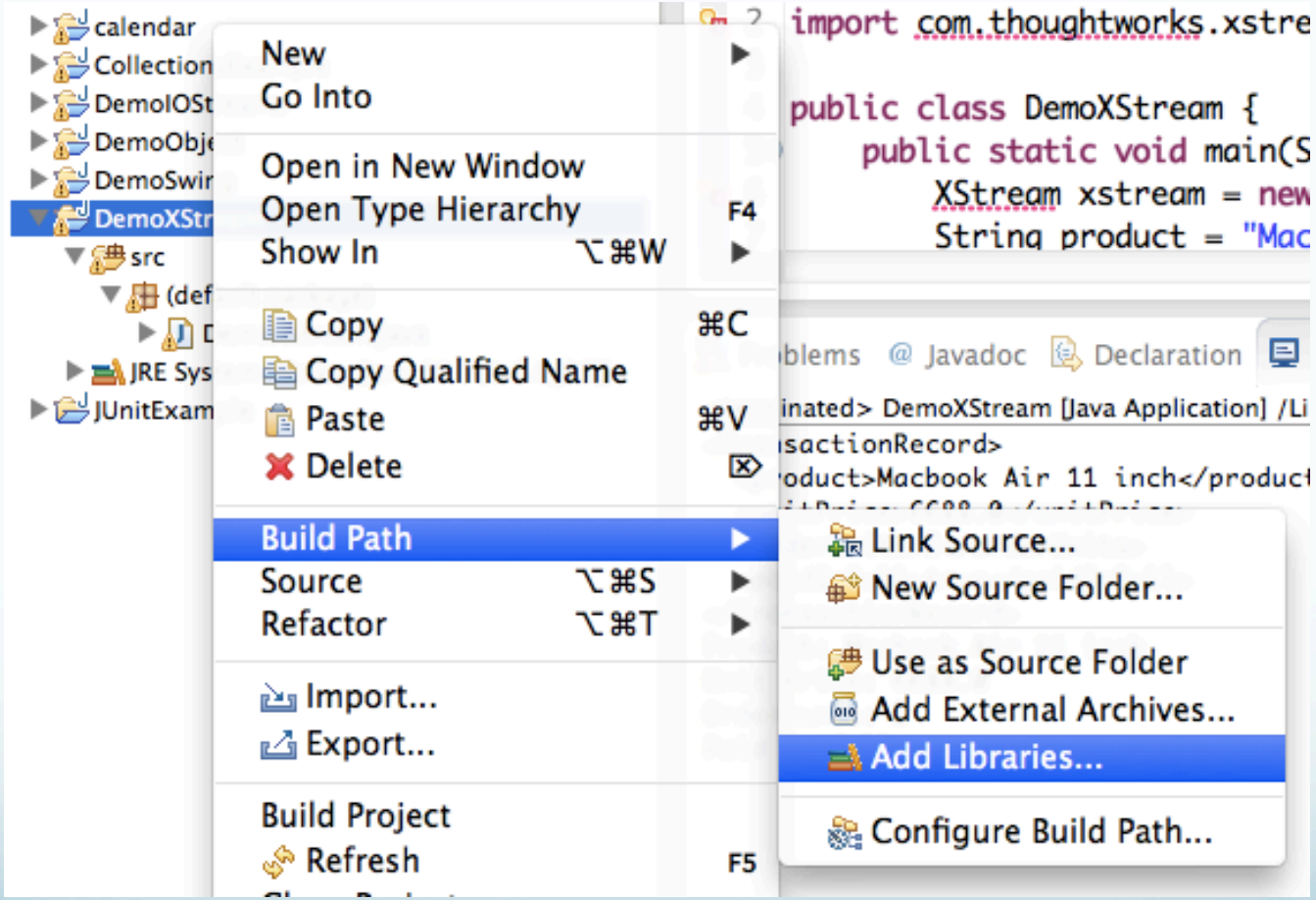


Setup: Unzip and identify all the JAR files

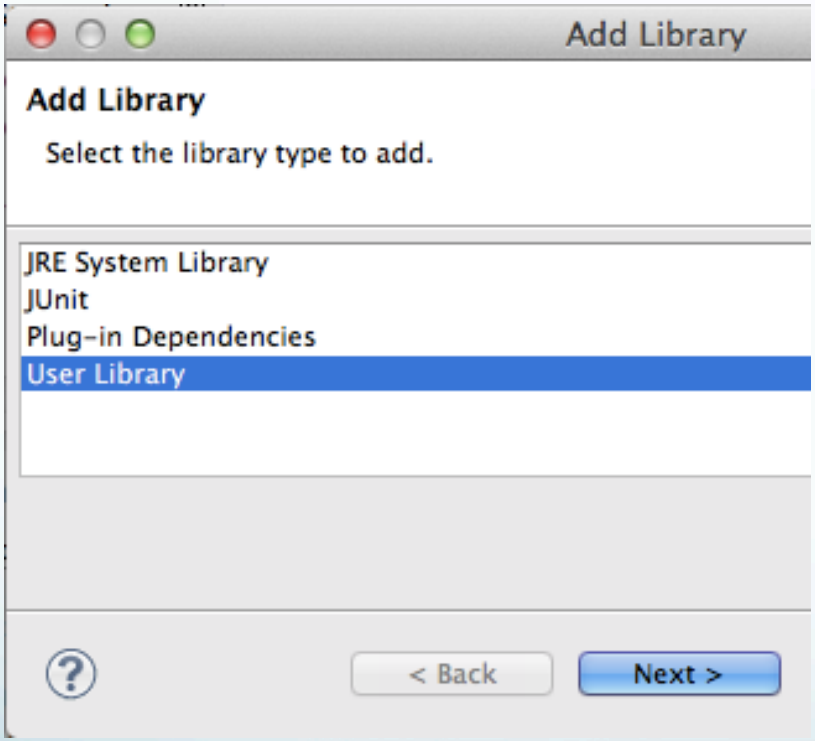
- Usually, a library comes with the core library (i.e. xstream-1.4.7.jar in this example)
- It may also come with a number of dependencies (i.e. the jar files inside the xstream subfolder)
- The next step is to configure a build path to include all these JAR files

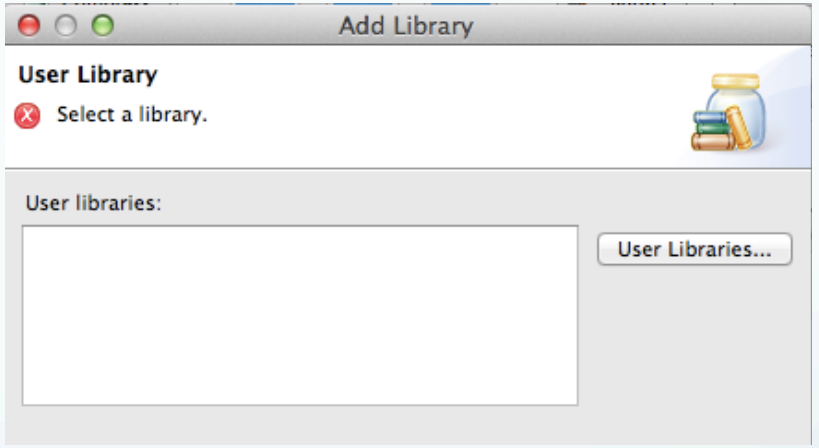


Setup: Add Libraries to your Java project

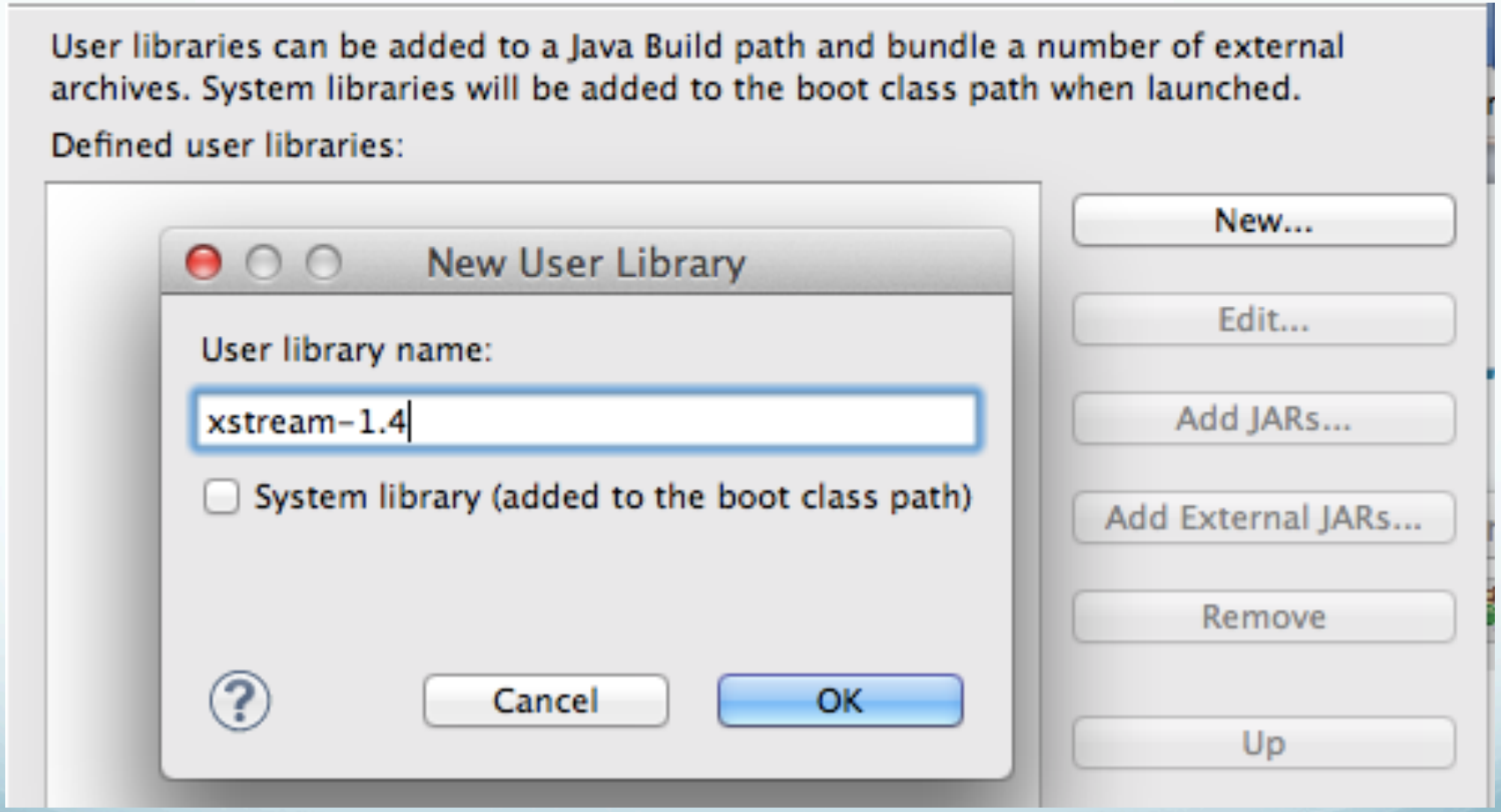


Configure a “User Library”





Click “New” to create a new user library



Add external JARs

- Ensure that ALL JAR files, including the dependencies are included