Files

ACS-1904 LECTURE 7

Files

- Recall Scanner from 1903.
- A scanner object can reference a text file

```
Scanner f = new Scanner(new File("file name goes here"));
```

- Scanner methods can be applied to reading the file
  - `f.next()`
  - `f.nextInt()`
  - `f.hasNext()`
  - `etc.`

System.out

```
File f = new File("myfile.txt");
FileOutputStream fs = new FileOutputStream(f);
PrintStream ps = new PrintStream(fs);
System.setOut(ps);
System.out.println("your data goes here");
```

Redirecting output to a file instead of standard output

Redirecting output

RedirectOutputToFile.java

```
import java.io.File;
import java.io.PrintStream;
import java.io.FileOutputStream;
public class RedirectOutputToFile{
    public static void main(String[] args) throws Exception{
        System.out.println("1. to standard output");
        PrintStream standard = System.out;
        File f = new File("myfile.txt");
        FileOutputStream fs = new FileOutputStream(f);
        PrintStream ps = new PrintStream(fs);
        System.setOut(ps);
        System.out.println("2. to the other file");
        ps.close();
        System.setOut(standard);
        System.out.println("3. to standard output");
    }
}
```

Errors could occur:
file not found, out of space, etc

If this file already exists any existing lines are removed.

create a new PrintSteam that replaces the standard output

close the file

reset to its initial value

to the Terminal Window

Files - Binary vs XML Files

Binary files

- Machine-readable
- Efficient for space and time
- Data stored on disk just as stored in memory

Extensible Markup Language (XML) files

- Human-readable
- Stored in text format with XML tags
- Easily viewed using a browser or word processor

Storing values vs objects

First we consider storing and retrieving values in files

- Primitive values: int, double, boolean, …
- Strings

Then we consider storing and retrieving objects

- a whole object graph is unit of transfer

Storing values (primitive data & strings) in files

First we consider Binary files

Then we consider XML files

Binary files: examples using a file named `myData.ser`

XML files: examples using a file named `myData.xml`

Examples write/read an int array of 5 values:

```
5  20  30  2  7
```

Binary files: primitive data and strings

We will use Java classes DataOutputStream and DataInputStream

Methods of **`DataOutputStream`**	Methods of **`DataInputStream`**
`close()`	`close()`
`writeBoolean(boolean b)`	`readBoolean()`
`writeByte(byte b)`	`readByte()`
`writeChar(int c)`	`readChar()`
`writeDouble(double d)`	`readDouble()`
`writeFloat(float f)`	`readFloat()`
`writeInt(int i)`	`readInt()`
`writeLong(long m)`	`readLong()`
`writeShort(short s)`	`readShort()`
`writeUTF(String s)`	`readUTF()`
	`available()`

Writing primitive data & strings to binary file

WriteBinary.java

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class WriteBinary {
   public static void main(String[] args) throws IOException {
      int[] myData = {5, 20, 30, 2, 7};
      DataOutputStream  os = new DataOutputStream(
                 new FileOutputStream("myData.ser"));
      for (int i=0; i<5; i++)
         os.writeInt(myData[i]);
      os.close();
   }
}
```

Errors could occur: file not found, out of space, etc

5 values to write

File to create

Write values

Close the file

Note: the file `myData.ser` is stored in a binary form and so it is not human-readable; it is machine-readable.

Reading primitive data & strings from binary file ReadBinary.java

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
public class ReadBinary {
   public static void main(String[] args) throws IOException{
      int[] myData = new int[5];
      DataInputStream is = new DataInputStream
                (new FileInputStream("myData.ser"));
      // get values from file into array
      for (int i=0; i< myData.length; i++)
         myData[i] = is.readInt();
      // display values in array
      for (int i: myData)System.out.println(i);
      is.close();
   }
}
```

Errors could occur: file not found, data missing, …

File to read

Get values

Close the file

XML Files

XML files are text files – human readable

- Values are enclosed in XML tags, for example

```
<int>183</int>
```

- For output use class `XMLEncoder` and method `writeObject`
- For input use class `XMLDecoder` and method `readObject`

Writing primitive data & strings to XML

WritePrimitiveDataToXML.java

```
import java.beans.XMLEncoder;
import java.io.FileOutputStream;
import java.io.IOException;
public class WritePrimitiveDataToXML {
    public static void main(String[] args)throws IOException{
    XMLEncoder encoder = new XMLEncoder
            (new FileOutputStream("myData.xml"));
      int[] myData = {5, 20, 30, 2, 7};
      for (int i=0; i<5; i++)
            encoder.writeObject(myData[i]);
      encoder.close();
    }
}
```

Errors could occur: file not found, out of space, etc.

File to create

5 values to write

Write values

Close the file

Note: the file xml is stored in XML format and is human-readable

XML files – contents of myData.xml

Extensible Markup Language (XML) files are human-readable where data is encoded in XML tags

The 5 values 5, 20, 30, 2, 7 are written to myData.xml which is:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_60"
class="java.beans.XMLDecoder">
 <int>5</int>
 <int>20</int>
 <int>30</int>
 <int>2</int>
 <int>7</int>
</java>
```

Java data occurs between

Five integers each enclosed in XML tags <int> and </int>

Reading primitive data & strings from XML ReadPrimitiveDataFromXML.java

```
import java.beans.XMLDecoder;
import java.io.FileInputStream;
import java.io.IOException;
public class ReadPrimitiveDataFromXML {
    public static void main(String[] args)throws IOException{
      // decoder object references the XML file
      XMLDecoder decoder = new XMLDecoder(new
               FileInputStream("myData.xml"));
      // get the five int values
      int[] myData = new int[5];
      for (int i=0; i<myData.length; i++)
          myData[i] = (int) decoder.readObject();
      // display the array and close the file
      for (int i: myData)
          System.out.println(i);
      decoder.close();
    }
}
```

Errors could occur:
file not found, data missing, …

File to create

Get values

Read as an object,
*need to **cast** to int*

Close the file

Summary - primitive values & strings

- Programming for binary files is very similar to the programming required for XML files.
- XML files are human-readable.
- Processing binary files will always be more efficient – virtually no translation of information required.

Objects

Now we examine writing/reading whole **objects** to/from a file

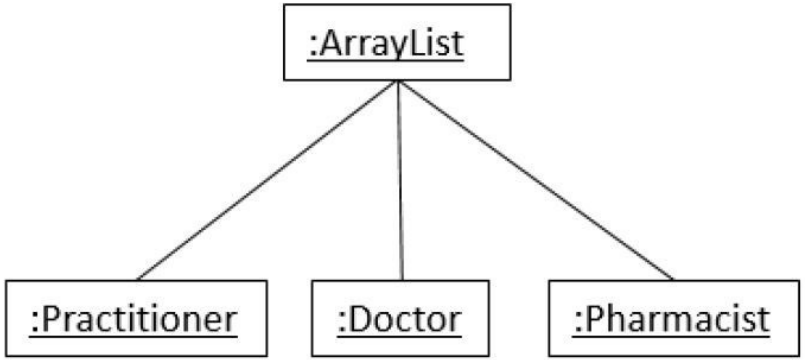
1. Binary files
2. XML files

Writing **objects** out to a file is referred to as ***serializing*** objects

Rather than one object, an ***object graph*** is read/written
An object graph consists of an object and all objects reachable from that object.

Objects

Our examples will use the object graph below which consists of an array list and its elements



- Binary files: examples use a file named `practitioners.ser`
- XML files: examples use a file named `practitioners.xml`

Objects and binary files

The objects must be instantiated from a class that implements Serializable

Recall the Practitioner class:

```
import java.io.Serializable;
public class Practitioner implements Serializable{
    private String firstName;
    private String lastName;
    private String gender;
    public Practitioner()      {
        firstName = lastName = gender   = "unknown";
    }
   …
```

Binary files: writing objects

Binary files are machine-readable, efficient use of storage and time

We will use class `ObjectOutputStream`

To write an object graph use `writeObject(…)`

Class must implement `Serializable`

Example writes to a file named `practitioners.ser`

Writing objects to a binary file

PractitionersToBinary.java

```
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
public class PractitionersToBinary {
   public static void main(String[] args) throws IOException {
      // List of practitioners
      ArrayList<Practitioner> practitioners = new ArrayList();
      // Create some practitioners
      Practitioner pr = new Practitioner("Sam","Smith","female");
      Doctor dr = new Doctor("Jill","Jones","female","Dermatology");
      Pharmacist ph = new Pharmacist("Eddy","Edwards","male","Drugco");
      practitioners.add(pr);
      practitioners.add(dr);
      practitioners.add(ph);
```

Needed to write objects

An ArrayList and its contents

(continued…)

...

```
    // create an object to reference practitioners.ser
    ObjectOutputStream os = new ObjectOutputStream ( new
                    FileOutputStream("practitioners.ser"));
    ...
       os.writeObject (practitioners);
       os.close();
    }
}
```

The file to create

One `writeObject` for the `ArrayList` object

Binary files: reading objects

We will use Java class `ObjectInputStream`

To read an object graph we use `readObject()`

Example reads from the file `practitioners.ser`

Reading Objects from a binary file

PractionersFromBinary.java

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
public class PractitionersFromBinary {
    public static void main(String[] args) throws IOException,
            ClassNotFoundException{
        ObjectInputStream is = new ObjectInputStream(new
            FileInputStream("practitioners.ser"));
```

Needed to read objects

The file to read

(continued…)

...

```
        // The JVM only knows the object read as being of type Object.
        // Since we know the object being read is of type ArrayList
        //    we include a cast to type ArrayList to the right of
        //    the assignment operator.
        ArrayList<Practitioner> practitioners =
                (ArrayList) is.readObject();
        is.close();
        for (Practitioner p: practitioners) {
            String type="practitioner";
            if (p instanceof Doctor) type="doctor";
            if (p instanceof Pharmacist) type="pharmacist";
            System.out.println(type+" "+p.getFirstName());
        }
    }
}
```

Get the whole ArrayList object in one read

Iterate through the array list and display each practitioner.

Objects and XML files

Use XMLEncoder and XMLDecoder

Class must have

- A no-arg constructor
- Getters
- Setters

Use `writeObject(…)` and `readObject()`

Write, read whole object graph at a time.

XML files: writing objects

XML files are human-readable

- Verbose - XML tags identify the type of element

We will use Java class `XMLEncoder`

To write an object graph use `writeObject(…)`

Class must have no-arg constructor, getters, setters

Example writes to a file named `practitioners.xml`

Writing Objects to XML file

PractitionersToXML.java

```
import java.beans.XMLEncoder;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
public class PractitionersToXML {
   public static void main(String[] args) throws IOException{
      // List of practitioners
      ArrayList<Practitioner> practitioners = new ArrayList();
      // Create some practitioners
      Practitioner pr = new Practitioner("Sam","Smith","female");
      Doctor dr = new Doctor("Jill","Jones","female","Dermatology");
      Pharmacist ph = new Pharmacist("Eddy","Edwards","male","Drugco");
      practitioners.add(pr);
      practitioners.add(dr);
      practitioners.add(ph);
```

Needed to write objects

An ArrayList and its contents

(continued…)

...

```
        // the encoder object references the file
        XMLEncoder encoder = new XMLEncoder(
             new FileOutputStream("practitioners.xml"));
        // write out the practitioner object graph
        encoder.writeObject(practitioners);
        // close the xml file
        encoder.close();
    }
}
```

The file to create

one `writeObject` for the `ArrayList` object

XML files: reading objects

We will use Java class `XMLDecoder`

To read an object graph we use `readObject()`

Example reads a file named `practitioners.xml`

Note the XML file is self-describing, contents next slide →

XML files: reading objects – contents of practitioners.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_60" class="java.beans.XMLDecoder">
 <object class="java.util.ArrayList">
  <void method="add">
   <object class="Practitioner">
    <void property="firstName">
     <string>Sam</string>
    </void>
    <void property="gender">
    .
    .
    .
```

First object to recreate

Second object to recreate

First field is firstName with value "Sam"

Reading objects from XML

PractitionersFromXML.java

```
import java.beans.XMLDecoder;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
public class PractitionersFromXML {
   public static void main(String[] args) throws IOException{
      // decoder object references the xml file
      XMLDecoder decoder = new XMLDecoder( new
           FileInputStream("practitioners.xml"));
```

Needed to read objects

The file to read

(continued…)

```
        // The JVM only knows the object read as being of type Object.
        // Since we know the object being read is of type ArrayList
        //    we include a cast to type ArrayList to the right of
        //    the assignment operator.
        ArrayList<Practitioner> practitioners =
                    (ArrayList) decoder.readObject();
        decoder.close();
        // display the practitioners, doctors, etc.
        for (Practitioner p: practitioners) {
            String type="practitioner";
            if (p instanceof Doctor) type="doctor";
            if (p instanceof Pharmacist) type="pharmacist";
            System.out.println(type+" "+p.getFirstName());
        }
    }
}
```

Get the whole ArrayList object in one read

Iterate through the array list and display each practitioner.

Summary

- Being able to serialize objects is important as it provides the information that enables objects to be re-instantiated at a later time by another program.
- Objects do not have to be destroyed when a program ends. Serialization provides a form of persistence similar to that provided in database systems, but not with all the features that make databases unique.
- Programming serialization of objects with binary files is very similar to the programming required for XML files.
- XML files are a human-readable.
- Processing binary files will always be more efficient – virtually no translation of information required.