Choosing output format

Dynamically choosing format

Problem

Our code should choose an output format based on input from the user.

So, we need a way to dynamically choose a formatter class based on some parameter.

E.g. if the user supplies a request for “xml” as format, our code should choose a class capable of XML formatting.

But our code should not have any knowledge of what formatters exist or what the formats are called. That’s not the responsibility of our client code.

Delegate the problem of choosing formatter class

Strategy: Let our code forward the format string to some other class and let that class return to us a formatter object that can handle the requested format.

Client sends “xml” -> Our code saves that in a variable called format, and asks a formatter factory to give us a formatter based on the value of the format variable:

```
Formatter formatter = null;
String    format    = args[0];
formatter = FormatterFactory.getFormatter(format);
```

Problem 2: how does FormatterFactory do it?

FormatterFactory has a factory method getFormatter that takes the String format as argument and chooses a suitable Format object.

But how can a method return different objects based on a String argument?

NOTE: We don’t want to hard code any known formats into the method!

But, generally, if we have a String and want an object, what datatype can we use for this?

Use a Map!

The getFormatter method could lookup the format parameter in an internal Map and return the result:

```
Formatter formatter = formatters.get(format);
if(formatter == null){
  throw new FormatNotSupportedException(format);
}
return formatter;


// formatters is a Map<String,Formatter> !
```

But how is the Map populated?

We still strongly discourage you to hard code in any values into your programs.

But how, then, is the Map populated with format strings and the corresponding Formatter objects?

Use a settings file

We could have a file listing all supported format names and the corresponding Formatter classes that are capable of producing such formats:

```
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>The following formats and classes exist</comment>
    <entry key="xml">org.henrikard.student.formats.XMLFormatter</entry>
    <entry key="json">org.henrikard.student.formats.JsonFormatter</entry>
</properties>
```

When do we read that file?

The file must be read before we use the FormatterFactory, so it can be used to populate FormatterFactory's Map. We can do that in a static-block.

But just reading the file is not enough, we must make sure that the corresponding classes are available too.

From the file we get for instance:

```
<entry key="xml">org.henrikard.student.formats.XMLFormatter</entry>
```

So how do we create a new XMLFormatter instance from that string?

We can do it like JDBC does it

We can use the old Class.forName() trick, to dynamically try and load the class names found in the XML settings file.

But what should for instance XMLFormatter do when it’s loaded?

Like in JDBC, we want it to register itself with some kind of manager.

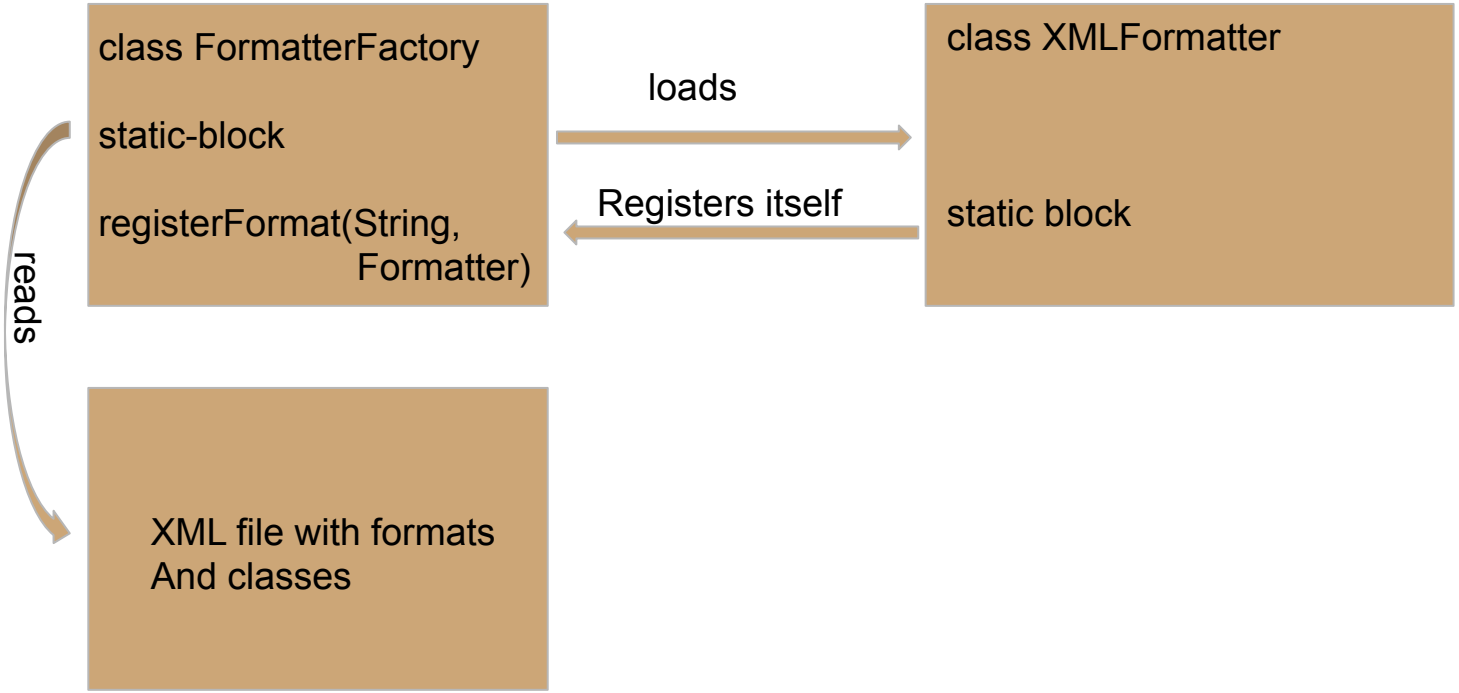
We could use FormatterFactory as this manager.

Class loading strategy

FormatterFactory has a static block in which the XML file (with formats and formatter classes) is read. For each class found, we do Class.forName() which loads the class into the JVM.

When for instance XMLFormatter is loaded, it registers an instance of itself with the FormatterFactory (which in turn, stores the instance and the format name in its internal Map).

Class diagram (colaborations)



How can getFormatter return different types?

The getFormatter(String) method of FormatterFactory can return either a JsonFormatter object, or an XMLFormatter object.

How is that possible?

The return type of getFormatter(String) is Formatter

So, both JsonFormatter and XMLFormatter are of type Formatter.

Therefore, we can guess that Formatter is .....

An interface

```
public interface Formatter{
    public void   loadFromList(List<Student> list);
    public String getDocument();
    public String getContentType();
}
```

Using the Storage interface from previous lecture

Previously, we saw that in order to hide the fact that we are using a database and JDBC from the client code (e.g. Main), we could use the Storage interface we created for this purpose.

We can use the Formatter interface in the same manner (to hide from Main what type of format we are using).

Remember the analogy of calling a Taxi - we request a Taxi but we don’t want to care about what color, make, etc the taxi actually has.

The taxi service could be thought of as a Taxi object factory class.

Using Storage and Formatter

Main can now look something like this (simplified):

```
Formatter formatter    = null;
String    format       = args[0]; // could be a GET parameter as well...
formatter              = FormatterFactory.getFormatter(format);
StudentStorage storage = StudentStorageFactory.getStorage();
List<Student> students = storage.getAllStudents();

formatter.loadFromList(students);
// Use stderr to separate data and logging/debug
System.err.println("All students as " + formatter.getContentType() + ":");
System.out.println(formatter.getDocument());
```

The static block of FormatterFactory

```
static{
  try{
    String formatsXML = System.getProperty("formatsXML");
    Properties formats = new Properties();
    formats.loadFromXML(new FileInputStream(formatsXML));
    for(String format : formats.stringPropertyNames()){
      String className = formats.getProperty(format);
      System.err.println(format + " formatter found, loading class: " + className);
      try{
        Class.forName(className); // will trigger the static block of this class!
      }catch(ClassNotFoundException cnfe){ System.err.println(cnfe.getMessage()); }
    }
  }catch(FileNotFoundException fne){
    System.err.println(fne.getMessage());
  }catch(IOException ioe){
    System.err.println(ioe.getMessage());
  }
}
```

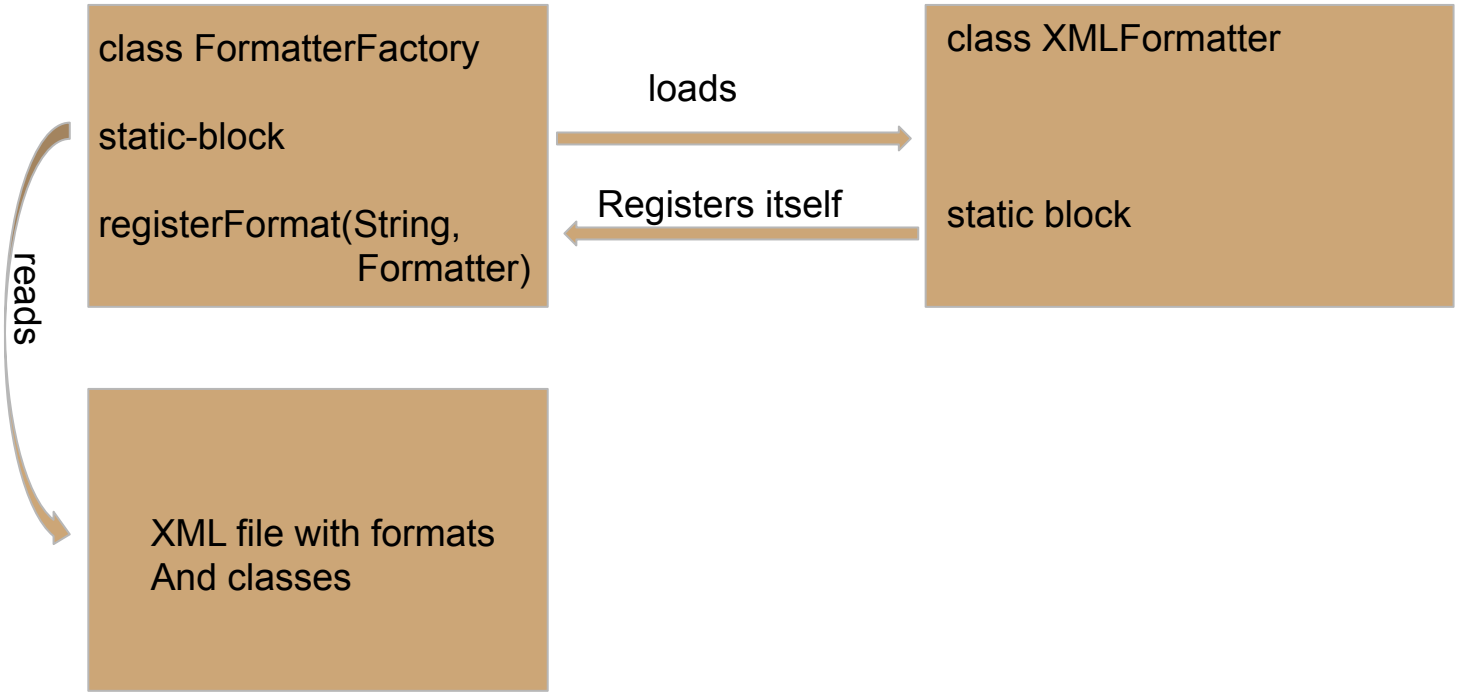
Output from the static block when run

```
xml formatter found, loading class: org.henrikard.student.formats.XMLFormatter
json formatter found, loading class: org.henrikard.student.formats.JsonFormatter
```

Static block of JsonFormatter

```
private static JsonFormatter instance;
static{
  instance = new JsonFormatter();
  FormatterFactory.registerFormatter("json", instance);
}
```

Class diagram (reprise)



Example run

```
java -DformatsXML=formats.xml -cp .:sqlite.jar:json.jar org.henrikard.student.main.Main xml
All students as application/xml:
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<STUDENTS>
  <STUDENT id="1">
    <NAME>Anna Andersson</NAME>
  </STUDENT>
  <STUDENT id="2">
    <NAME>Beata Bengtssson</NAME>
  </STUDENT>
  <STUDENT id="3">
    <NAME>Cecilia Carlsson</NAME>
  </STUDENT>
  <STUDENT id="4">
    <NAME>David Davidsson</NAME>
  </STUDENT>
  <STUDENT id="5">
    <NAME>Erik Eskilsson</NAME>
  </STUDENT>
  <STUDENT id="6">
    <NAME>Fader Fourah</NAME>
  </STUDENT>
</STUDENTS>
```

Generating the format

Since we have abstracted away the database, and only deal with List<Student>, it is quite simple to loop through all students and create either XML or Json elements.

This means that the Storage abstraction benefited both the writers of Main and the writers of the Formatter-implementations.

loadFromList() - Json version

```
public void loadFromList(List<Student> list){
  StringBuilder page = new StringBuilder();
  Map<String, Boolean> config = new HashMap<String, Boolean>();
  config.put(JsonGenerator.PRETTY_PRINTING, true);
  StringWriter writer = new StringWriter();
  JsonWriterFactory jwf = Json.createWriterFactory(config);
  JsonWriter jWriter = jwf.createWriter(writer);
  JsonObjectBuilder job = Json.createObjectBuilder();
  JsonArrayBuilder jab = Json.createArrayBuilder();
  for( Student student : list ){
    jab.add(Json.createObjectBuilder()
       .add("studentName", student.name())
       .add("studentID", student.id()));
  }
  job.add("students", jab.build());
  jWriter.writeObject(job.build());
  jWriter.close();
  page.append(writer.toString());
  this.document = page.toString();
}
```

loadFromList() - XML version

```
public void loadFromList(List<Student> list){
  try {
    DocumentBuilderFactory docFactory =
              DocumentBuilderFactory.newInstance();
    DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
    Document doc = docBuilder.newDocument();
    Element rootElement = doc.createElement("STUDENTS");
    doc.appendChild(rootElement);
    for(Student stud : students ){
      Element student = doc.createElement("STUDENT");
      student.setAttribute("id", stud.id()+"");
      Element name    = doc.createElement("NAME");
      name.appendChild(doc.createTextNode(stud.name()));
      student.appendChild(name);
      rootElement.appendChild(student);
    }
```

loadFromList() - XML version, cont.

```
    TransformerFactory transformerFactory =
                TransformerFactory.newInstance();
    Transformer transformer = transformerFactory.newTransformer();
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer
      .setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
    DOMSource source = new DOMSource(doc);
    StringWriter sw = new StringWriter();
    StreamResult result = new StreamResult(sw);
    transformer.transform(source, result);
    this.document = sw.toString();
  } catch (ParserConfigurationException pce) {
    pce.printStackTrace();
  } catch (TransformerException tfe) {
    tfe.printStackTrace();
  }
}
```

Complete main

```
public static void main(String[] args){
  if(args.length != 1){
    System.err.println("No format argument found.");
    System.exit(1);
  }
  try{
    Formatter formatter = null;
    String format = args[0];
    formatter = FormatterFactory.getFormatter(format);
    StudentStorage storage = StudentStorageFactory.getStorage();
    List<Student> students = storage.getAllStudents();
    formatter.loadFromList(students);
    System.err.println("All students as " + formatter.getContentType() + ":");
    System.out.println(formatter.getDocument());
  }catch(FormatNotSupportedException fnse){ System.err.println(fnse.getMessage());
  }catch(StorageException se){
    System.err.println("Error accessing the storage: " + se.getMessage());
  }
}
```

System.getProperty("formatsXML")

We ran the program with the -D flag:

```
java -DformatsXML=formats.xml
```

Which allowed us to get the name of the format xml file like this:

```
String formatsXML = System.getProperty("formatsXML");
```

This allows us to specify the location of the xml file on the command line.

Parsing the XML file into a Properties object

```
Properties formats = new Properties();
formats.loadFromXML(new FileInputStream(formatsXML));
for(String format : formats.stringPropertyNames()){
  String className = formats.getProperty(format);
  System.err.println(format + " formatter found, loading class: " + className);
  // etc...
}
```

How would we go about adding a format?

We'd add one line to the formats.xml file:

```
<entry key="csv">org.henrikard.student.formats.CSVFormatter</entry>
```

We'd add one class to the formats package, `CSVFormatter implements Formatter` and make sure it registers itself with FormatterFactory.

Do we have to change Main.main() ?

How would we go about adding a format?

We'd add one line to the formats.xml file:

```
<entry key="csv">org.henrikard.student.formats.CSVFormatter</entry>
```

We'd add one class to the formats package, `CSVFormatter implements Formatter` and make sure it registers itself with FormatterFactory.

Do we have to change Main.main() ?

No!

The user only has to give the argument csv and Main will do the right thing.

Directory layout and files

```
|-- formats.xml
`-- org
    `-- henrikard
        `-- student
            |-- db
            |   `-- DBUtil.java
            |-- domain
            |   `-- Student.java
            |-- formats
            |   |-- FormatNotSupportedException.java
            |   |-- FormatterFactory.java
            |   |-- Formatter.java
            |   |-- JsonFormatter.java
            |   `-- XMLFormatter.java
            |-- main
            |   `-- Main.java
            |-- resources
            |   |-- javax.json.jar
            |   |-- sqlite-jdbc-3.8.11.2.jar
            |   `-- students.db
            `-- storage
                |-- StorageException.java
                |-- StudentStorageDB.java
                |-- StudentStorageFactory.java
                `-- StudentStorage.java
```

Complete build and run command

```
javac -cp .:org/henrikard/student/resources/javax.json.jar */*/*/*/*.java && \
java -DformatsXML=formats.xml \
-cp .:org/henrikard/student/resources/sqlite-jdbc-3.8.11.2.jar:org/henrikard/student/resources/javax.json.jar \
org.henrikard.student.main.Main json
```

Classpath for the compiler is for the json library:

```
org/henrikard/student/resources/javax.json.jar
```

Classpath for the JVM is for the SQLite and json libraries:

```
org/henrikard/student/resources/javax.json.jar
org/henrikard/student/resources/sqlite-jdbc-3.8.11.2.jar
```

The `-D` flag is to pass the `formatsXML` location to the System properties as being `formats.xml` (in current directory)

The argument `json` is the user choosing json output

Note: On Windows, you need to separate class paths with a semicolon instead of colon

Summary

Main requests a Formatter object from FormatterFactory, passing along a String from the user with the format.

FormatterFactory loads an XML file with all formats and their formatter classes' class names. It then loads all Formatter classes.

The formatter classes register themselves with FormatterFactory which stores the format name and an instance in a private Map.

The getFormatter(String) method returns the correct Formatter instance using its Map (or throws a FormatNotSupportedException if it can't).