CPSC 501 Assignment 3

Stephen Armstrong

00306924

**Final Code**

**Sender.java**

**import** java.awt.Point;

**import** java.io.\*;

**import** java.util.\*;

**import** org.jdom.\*;

**import** org.jdom.output.Format;

**import** org.jdom.output.XMLOutputter;

**import** java.net.Socket;

**public** **class** Sender {

**static** List<Object> *objList*;

**public** **static** **void** main(String[] args) **throws** Exception {

String server = "localhost";

**int** port = Integer.*parseInt*("3333");

*objList* = **new** ArrayList<Object>();

**boolean** exit\_flag = **false**;

**while** (!exit\_flag) {

System.*out*.println("Create an object?");

System.*out*.println("1. Yes");

System.*out*.println("2. No");

System.*out*.println("3. List Objects");

System.*out*.println("4. Exit");

**try** {

BufferedReader bufferRead = **new** BufferedReader(

**new** InputStreamReader(System.*in*));

String input = bufferRead.readLine();

**if** (input.equals("1")) {

System.*out*.println("what kind of object?");

System.*out*.println("1. Simple Object(int,boolean)");

System.*out*

.println("2. Simple Object With Reference(simpleObject)");

System.*out*.println("3. Array(int[])");

System.*out*

.println("4. Array with References(simpleObject[])");

System.*out*

.println("5. Java Collection Class(list<simpleObject>)");

System.*out*.println("6. Back");

bufferRead = **new** BufferedReader(**new** InputStreamReader(

System.*in*));

input = bufferRead.readLine();

**if** (input.equals("1")) {

simpleObject obj = *createSimpleObject*();

*objList*.add(obj);

} **else** **if** (input.equals("2")) {

*createRefObject*();

} **else** **if** (input.equals("3")) {

*createSimpleArray*();

} **else** **if** (input.equals("4")) {

*createRefArray*();

} **else** **if** (input.equals("5")) {

*createCollection*();

} **else** **if** (input.equals("6")) {

// do nothing

} **else** {

System.*out*.println("invalid input");

}

} **else** **if** (input.equals("2")) {

*serialize*(server, port);

} **else** **if** (input.equals("3")) {

System.*out*.println(*objList*.toString());

} **else** **if** (input.equals("4")) {

System.*out*.println("Exiting...");

exit\_flag = **true**;

} **else** {

System.*out*.println("Invalid input");

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**private** **static** **void** serialize(String server, **int** port) **throws** IOException,

Exception {

System.*out*.println("Serialize and transfer to receiver?");

System.*out*.println("1. Yes");

System.*out*.println("2. No");

BufferedReader bufferRead = **new** BufferedReader(**new** InputStreamReader(

System.*in*));

String input = bufferRead.readLine();

**if** (input.equals("1")) {

**for** (Object obj : *objList*) {

System.*out*.println("Deserializing object...");

Document doc = Serializer.*serialize*(obj);

File aFile = *createFile*(doc);

*transferFile*(server, port, aFile);

}

} **else** **if** (input.equals("2")) {

} **else** {

System.*out*.println("Invalid input");

}

}

**public** **static** File createFile(Document doc) **throws** IOException {

XMLOutputter out = **new** XMLOutputter(Format.*getPrettyFormat*());

File aFile = **new** File("sentdata.xml");

BufferedWriter writer = **new** BufferedWriter(**new** FileWriter(aFile));

out.output(doc, writer);

writer.close();

**return** aFile;

}

**private** **static** **void** transferFile(String server, **int** port, File aFile) {

System.*out*.println("Transferring file...");

**try** {

Socket s = **new** Socket(server, port);

OutputStream output = s.getOutputStream();

FileInputStream fileInputStream = **new** FileInputStream(aFile);

**byte**[] buffer = **new** **byte**[1024 \* 1024];

**int** bytesRead = 0;

**while** ((bytesRead = fileInputStream.read(buffer)) > 0) {

output.write(buffer, 0, bytesRead);

}

fileInputStream.close();

s.close();

System.*out*.println("Transfer Complete");

} **catch** (IOException e) {

System.*out*.println("connection refused");

}

}

**private** **static** **void** createCollection() **throws** IOException {

**boolean** flag = **false**;

**while** (!flag) {

System.*out*.println("add an object to collection?");

System.*out*.println("1. Yes");

System.*out*.println("2. No");

BufferedReader bufferRead = **new** BufferedReader(

**new** InputStreamReader(System.*in*));

String input = bufferRead.readLine();

ArrayList<simpleObject> list = **new** ArrayList<simpleObject>();

**if** (input.equals("1")) {

simpleObject obj = *createSimpleObject*();

list.add(obj);

} **else** **if** (input.equals("2")) {

Object obj = **new** collectionObject(list);

*objList*.add(obj);

flag = **true**;

} **else** {

System.*out*.println("Invalid input");

}

}

}

**private** **static** **void** createSimpleArray() **throws** IOException {

System.*out*.println("Enter array values. Eg: '1,2,3'");

BufferedReader bufferRead = **new** BufferedReader(**new** InputStreamReader(

System.*in*));

String input = bufferRead.readLine();

String[] fields = input.split(",");

**int**[] ints = **new** **int**[fields.length];

**for** (**int** i = 0; i < ints.length; i++) {

**try** {

ints[i] = Integer.*parseInt*(fields[i]);

} **catch** (Exception e) {

System.*out*.println("You can only enter numbers");

}

}

Object obj = **new** simpleArray(ints);

*objList*.add(obj);

}

**private** **static** **void** createRefArray() **throws** IOException {

System.*out*.println("Enter length of array");

BufferedReader bufferRead = **new** BufferedReader(**new** InputStreamReader(

System.*in*));

String input = bufferRead.readLine();

**int** length = Integer.*parseInt*(input);

simpleObject[] simpleObjArr = **new** simpleObject[length];

**for** (**int** i = 0; i < length; i++) {

simpleObject obj = *createSimpleObject*();

simpleObjArr[i] = obj;

}

refArray obj = **new** refArray(simpleObjArr);

*objList*.add(obj);

}

**private** **static** **void** createRefObject() **throws** IOException {

simpleObject simpleobj = *createSimpleObject*();

Object obj = **new** refObject(simpleobj);

*objList*.add(obj);

}

**private** **static** simpleObject createSimpleObject() **throws** IOException {

**boolean** flag = **false**;

**while** (!flag) {

System.*out*.println("Enter field values. Eg: '1,true'");

BufferedReader bufferRead = **new** BufferedReader(

**new** InputStreamReader(System.*in*));

String input = bufferRead.readLine();

String[] fields = input.split(",");

**if** (fields.length != 2) {

System.*out*.println("You must enter exactly 2 parameters");

} **else** {

**boolean** bool = **false**;

**if** (fields[1].toLowerCase().equals("true")) {

bool = **true**;

} **else** **if** (fields[1].toLowerCase().equals("false")) {

bool = **false**;

} **else** {

System.*out*

.println("You must enter true or false for a boolean");

}

**try** {

**int** i = Integer.*parseInt*(fields[0]);

simpleObject obj = **new** simpleObject(i, bool);

**return** obj;

} **catch** (Exception e) {

System.*out*.println("You must enter a number");

}

}

}

**return** **null**;

}

}

**Receiver.java**

**import** java.io.\*;

**import** java.net.\*;

**import** org.jdom.input.SAXBuilder;

**import** org.jdom.Document;

**import** org.jdom.JDOMException;

**public** **class** Receiver {

**public** **static** **void** main(String[] args) **throws** IOException {

**int** port = 3333;

ServerSocket serverSocket = **null**;

// set the server socket

**try** {

serverSocket = **new** ServerSocket(port);

} **catch** (IOException e) {

System.*err*.println("Could not listen on port: " + port);

System.*exit*(1);

}

**while** (**true**) {

System.*out*.println("Waiting for client request");

File aFile = **new** File("recdata.xml");

// listen for client

Socket s = **null**;

**try** {

s = serverSocket.accept();

System.*out*.println("Client connected");

} **catch** (IOException e) {

System.*err*.println("Accept failed.");

System.*exit*(1);

}

*receiveFile*(aFile, s);

Object obj = *buildObject*(aFile);

Inspector inspector = **new** Inspector();

inspector.inspect(obj, **false**);

}

}

**private** **static** **void** receiveFile(File aFile, Socket s) **throws** IOException,

FileNotFoundException {

InputStream input = s.getInputStream();

FileOutputStream out = **new** FileOutputStream(aFile);

**byte**[] buffer = **new** **byte**[1024 \* 1024];

**int** bytesReceived = 0;

System.*out*.println("receiving file");

**while** ((bytesReceived = input.read(buffer)) > 0) {

out.write(buffer, 0, bytesReceived);

System.*out*.println(bytesReceived + " Bytes received");

**break**;

}

}

**private** **static** Object buildObject(File aFile) {

SAXBuilder builder = **new** SAXBuilder();

Object obj = **null**;

**try** {

Document doc = (Document) builder.build(aFile);

obj = Deserializer.*deserialize*(doc);

} **catch** (JDOMException e) {

e.printStackTrace();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** obj;

}

}

**Serialize.java**

**import** java.lang.reflect.\*;

**import** java.util.\*;

**import** org.jdom.Document;

**import** org.jdom.Element;

//adapted from examples in the textbook

**public** **class** Serializer {

**public** **static** Document serialize(Object obj) **throws** Exception {

**return** *serializeHelper*(obj, **new** Document(**new** Element("serialized")),

**new** IdentityHashMap());

}

**private** **static** Document serializeHelper(Object source, Document target,

Map table) **throws** Exception {

String id = Integer.*toString*(table.size());

table.put(source, id);

Class sourceclass = source.getClass();

Element oElt = **new** Element("object");

oElt.setAttribute("class", sourceclass.getName());

oElt.setAttribute("id", id);

target.getRootElement().addContent(oElt);

**if** (!sourceclass.isArray()) {

Field[] fields = sourceclass.getDeclaredFields();

**for** (**int** i = 0; i < fields.length; i++) {

**if** (!Modifier.*isPublic*(fields[i].getModifiers()))

fields[i].setAccessible(**true**);

Element fElt = **new** Element("field");

fElt.setAttribute("name", fields[i].getName());

Class declClass = fields[i].getDeclaringClass();

fElt.setAttribute("declaringclass", declClass.getName());

Class fieldtype = fields[i].getType();

Object child = fields[i].get(source);

**if** (Modifier.*isTransient*(fields[i].getModifiers()))

child = **null**;

fElt.addContent(*serializeVariable*(fieldtype, child, target,

table));

oElt.addContent(fElt);

}

} **else** {

Class componentType = sourceclass.getComponentType();

**int** length = Array.*getLength*(source);

oElt.setAttribute("length", Integer.*toString*(length));

**for** (**int** i = 0; i < length; i++)

oElt.addContent(*serializeVariable*(componentType,

Array.*get*(source, i), target, table));

}

**return** target;

}

**private** **static** Element serializeVariable(Class<?> fieldtype, Object child,

Document target, Map table) **throws** Exception {

**if** (child == **null**)

**return** **new** Element("null");

**else** **if** (!fieldtype.isPrimitive()) {

Element reference = **new** Element("reference");

**if** (table.containsKey(child))

reference.setText(table.get(child).toString());

**else** {

reference.setText(Integer.*toString*(table.size()));

*serializeHelper*(child, target, table);

}

**return** reference;

} **else** {

Element value = **new** Element("value");

value.setText(child.toString());

**return** value;

}

}

}

**Deserialize.java**

**import** org.jdom.Document;

**import** org.jdom.Element;

**import** java.lang.reflect.\*;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

//adapted from examples in the textbook

**public** **class** Deserializer {

**public** **static** Object deserialize(Document document) **throws** Exception {

List objList = document.getRootElement().getChildren();

Map table = **new** HashMap();

*createInstances*(table, objList);

*assignFieldValues*(table, objList);

**return** table.get("0");

}

**private** **static** **void** createInstances(Map table, List objList) **throws** Exception {

**for** (**int** i=0; i<objList.size(); i++) {

Element oElt = (Element) objList.get(i);

Class cls = Class.*forName*(oElt.getAttributeValue("class"));

Object instance = **null**;

**if** (!cls.isArray()) {

Constructor c = cls.getDeclaredConstructor(**null**);

**if** (!Modifier.*isPublic*(c.getModifiers()))

c.setAccessible(**true**);

instance = c.newInstance(**null**);

} **else**

instance = Array.*newInstance*(cls.getComponentType(),

Integer.*parseInt*(oElt.getAttributeValue("length")));

table.put(oElt.getAttributeValue("id"), instance);

}

}

**private** **static** **void** assignFieldValues(Map table, List objList) **throws** Exception {

**for** (**int** i=0; i < objList.size(); i++) {

Element oElt = (Element) objList.get(i);

Object instance = table.get(oElt.getAttributeValue("id"));

List fElts = oElt.getChildren();

**if** (!instance.getClass().isArray()) {

**for** (**int** j=0; j<fElts.size(); j++) {

Element fElt = (Element) fElts.get(j);

String className = fElt.getAttributeValue("declaringclass");

Class fieldDC = Class.*forName*(className);

String fieldName = fElt.getAttributeValue("name");

Field f = fieldDC.getDeclaredField(fieldName);

**if** (!Modifier.*isPublic*(f.getModifiers()))

f.setAccessible(**true**);

Element vElt = (Element) fElt.getChildren().get(0);

f.set(instance, *deserializeValue*(vElt, f.getType(), table));

}

} **else** {

Class comptype = instance.getClass().getComponentType();

**for** (**int** j=0; j<fElts.size(); j++)

Array.*set*(instance, j, *deserializeValue*((Element)fElts.get(j), comptype, table));

}

}

}

**private** **static** Object deserializeValue(Element vElt, Class fieldType, Map table) {

String valtype = vElt.getName();

**if** (valtype.equals("null"))

**return** **null**;

**else** **if** (valtype.equals("reference"))

**return** table.get(vElt.getText());

**else** {

**if** (fieldType.equals(**boolean**.**class**)) {

**if** (vElt.getText().equals("true"))

**return** Boolean.*TRUE*;

**else**

**return** Boolean.*FALSE*;

}

**else** **if** (fieldType.equals(**byte**.**class**))

**return** Byte.*valueOf*(vElt.getText());

**else** **if** (fieldType.equals(**short**.**class**))

**return** Short.*valueOf*(vElt.getText());

**else** **if** (fieldType.equals(**int**.**class**))

**return** Integer.*valueOf*(vElt.getText());

**else** **if** (fieldType.equals(**long**.**class**))

**return** Long.*valueOf*(vElt.getText());

**else** **if** (fieldType.equals(**float**.**class**))

**return** Float.*valueOf*(vElt.getText());

**else** **if** (fieldType.equals(**double**.**class**))

**return** Double.*valueOf*(vElt.getText());

**else** **if** (fieldType.equals(**char**.**class**))

**return** **new** Character(vElt.getText().charAt(0));

**else**

**return** vElt.getText();

}

}

}

**SimpleArray.java**

**public** **class** simpleArray {

**public** simpleArray() {

}

**public** simpleArray(**int**[] a) {

f1 = a;

}

**private** **int**[] f1;

}

**RefObject.java**

**public** **class** refObject

{

**public** refObject(){}

**public** refObject(simpleObject a) {

f1 = a;

}

**public** simpleObject f1;

}

**CollectionObject.java**

**import** java.util.ArrayList;

**public** **class** collectionObject

{

**public** collectionObject() {

}

**public** collectionObject(ArrayList a) {

f1 = a;

}

**private** ArrayList f1;

}

**SimpleObject.java**

**public** **class** simpleObject

{

**public** simpleObject() {

}

**public** simpleObject(**int** a, **boolean** b) {

f1 = a;

f2 = b;

}

**private** **int** f1;

**private** **boolean** f2;

}

**RefArray.java**

**import** java.awt.Point;

**public** **class** refArray {

**public** refArray() {

}

**public** refArray(Object[] a) {

f1 = a;

}

**private** Object[] f1;

}

**Inspector.java**

/\* -----------------------------------------------------------------------------------------------------------

\* CPSC 501 Assignment 2

\* Stephen Armstrong

\* 00306924

\* Methods inspect, inspectFieldClasses, and inspectFields based off of Jordan Kidney's ObjectInspector.java.

\* -----------------------------------------------------------------------------------------------------------\*/

import java.util.\*;

import java.lang.reflect.\*;

public class Inspector {

public Inspector() {

}

/\*

\* inspects the object returning information about methods, constructor,

\* interfaces, fields and the super class. If recursive is set to true then

\* the function will also inspect the superclass

\*/

public void inspect(Object obj, boolean recursive) {

Vector objectsToInspect = new Vector();

Class objClass = obj.getClass();

System.out.println("inside inspector: "

+ obj.getClass().getSimpleName() + " (recursive = " + recursive

+ ")");

System.out.println("Name of declaring class: "

+ objClass.getSimpleName());

System.out.println("Name of immediate superclass: "

+ objClass.getSuperclass().getSimpleName());

inspectInterfaces(obj, objClass);

inspectMethods(obj, objClass);

inspectConstructor(obj, objClass);

inspectFields(obj, objClass, objectsToInspect);

if ((objClass.getSuperclass() != null)

&& (objClass.getSuperclass() != Object.class)) {

inspectSuperclass(obj, objClass, objectsToInspect);

}

if (recursive)

inspectFieldClasses(obj, objClass, objectsToInspect, recursive);

}

/\*

\* Inspects the methods of an object, returns the method name, parameter

\* types, modifiers, exception types and return type.

\*/

private void inspectMethods(Object obj, Class objClass) {

System.out.println();

System.out.println("'" + objClass.getSimpleName() + "' Method(s):");

Method[] methods = objClass.getDeclaredMethods();

if (methods.length >= 1) {

System.out.println(methods.length + " method(s) detected");

for (int i = 0; i < methods.length; i++) {

Method aMethod = methods[i];

String params = getMethodParameters(aMethod);

String except = getMethodExceptions(aMethod);

System.out.println("Method: '" + aMethod.getName()

+ "'\n\t-Parameter Type(s): " + params

+ "\n\t-Modifier(s): "

+ Modifier.toString(aMethod.getModifiers())

+ "\n\t-Return Type(s): " + aMethod.getReturnType()

+ "\n\t-Exception Type(s): " + except);

}

System.out.println("End of '" + objClass.getSimpleName()

+ "' methods");

} else {

System.out.println("No methods detected");

}

}

/\*

\* returns a list of the exceptions thrown by a method

\*/

private String getMethodExceptions(Method aMethod) {

Class[] exceptions = aMethod.getExceptionTypes();

String except = "";

if (exceptions.length == 0)

except = "none";

else

for (Class aException : exceptions) {

except += aException.getSimpleName() + " ";

}

return except;

}

/\*

\* returns a list of parameters for a method

\*/

private String getMethodParameters(Method aMethod) {

Class[] parameters = aMethod.getParameterTypes();

String params = "";

if (parameters.length == 0)

params = "none";

else

for (Class aParam : parameters) {

params += aParam.getSimpleName() + " ";

}

return params;

}

/\*

\* Inspects the interfaces of the object

\*/

private void inspectInterfaces(Object obj, Class objClass) {

System.out.println();

System.out.println("'" + objClass.getSimpleName() + "' Interface(s):");

Class[] interfaces = objClass.getInterfaces();

if (interfaces.length > 0) {

System.out.println(interfaces.length + " Interface(s) found");

for (int i = 0; i < interfaces.length; i++) {

System.out.println();

System.out.println("Interface: " + interfaces[i].getName());

System.out.println("Inspecting interface '"

+ interfaces[i].getSimpleName() + "':");

inspectMethods(obj, interfaces[i]);

inspectConstructor(obj, interfaces[i]);

}

System.out.println("End of '" + objClass.getSimpleName()

+ "' interfaces");

} else {

System.out.println("No interfaces found");

}

}

/\*

\* Inspects the constructor(s) of an object

\*/

private void inspectConstructor(Object obj, Class objClass) {

System.out.println();

System.out

.println("'" + objClass.getSimpleName() + "' Constructor(s):");

Constructor[] constructors = objClass.getConstructors();

if (constructors.length > 0) {

System.out

.println(constructors.length + " Constructor(s) Detected");

for (int i = 0; i < constructors.length; i++) {

Constructor aConstructor = constructors[i];

String params = getConstructorParameters(aConstructor);

System.out.println("Constructor: " + aConstructor.getName()

+ "\n\t-Parameters: " + params + "\n\t-Modifiers: "

+ Modifier.toString(aConstructor.getModifiers()));

}

System.out.println("End of '" + objClass.getSimpleName()

+ "' constructors");

} else {

System.out.println("No constructors Detected");

}

}

/\*

\* returns a list of the parameters for a constructor

\*/

private String getConstructorParameters(Constructor aConstructor) {

Class[] parameters = aConstructor.getParameterTypes();

String params = "";

if (parameters.length == 0)

params = "none";

else

for (Class aParam : parameters) {

params += aParam.getSimpleName() + " ";

}

return params;

}

/\*

\* Inspects the superclass of an object

\*/

private void inspectSuperclass(Object obj, Class objClass,

Vector objectsToInspect) {

System.out.println();

System.out

.println("'" + objClass.getSimpleName() + "' Superclass(es):");

Class superclass = objClass.getSuperclass();

inspectMethods(obj, superclass);

inspectConstructor(obj, superclass);

inspectFields(obj, superclass, new Vector());

}

/\*

\* Inspects the fields of the class. prints the number of fields detected,

\* their value, their type and whether it is private, public etc.

\*/

private void inspectFields(Object obj, Class objClass,

Vector objectsToInspect) {

System.out.println();

System.out.println("Inspecting '" + objClass.getSimpleName()

+ "' Field(s):");

if (objClass.getDeclaredFields().length >= 1) {

Field[] fields = objClass.getDeclaredFields();

System.out.println(fields.length + " Field(s) detected");

for (int i = 0; i < fields.length; i++) {

Field aField = fields[i];

aField.setAccessible(true);

if (!aField.getType().isPrimitive())

objectsToInspect.addElement(aField);

printFields(obj, aField);

}

System.out.println("End of '" + objClass.getSimpleName()

+ "' fields");

} else {

System.out.println("No fields detected");

}

if (objClass.getSuperclass() != null)

inspectFields(obj, objClass.getSuperclass(), objectsToInspect);

}

/\*

\* prints the field information to console

\*/

private void printFields(Object obj, Field aField) {

try {

if (aField.getType().isArray()) {

Object array = aField.get(obj);

int length = Array.getLength(array);

System.out.print("Field: '" + aField.getName() + "' = {");

for (int i=0;i<length-1;i++){

Object element = Array.get(array, i);

System.out.print(element+",");

}

Object element = Array.get(array, length-1);

System.out.print(element);

System.out.println("}"

+ "'\n\t-Type: " + aField.getType().getComponentType()

+ "\n\t-Modifier: "

+ Modifier.toString(aField.getModifiers()));

} else {

System.out.println("Field: '" + aField.getName() + "' = "

+ aField.get(obj) + "\n\t-Type: " + aField.getType()

+ "\n\t-Modifier: "

+ Modifier.toString(aField.getModifiers()));

}

} catch (Exception e) {

}

}

/\*

\* Inspects object's field's which are also objects.

\*/

private void inspectFieldClasses(Object obj, Class objClass,

Vector objectsToInspect, boolean recursive) {

if (objectsToInspect.size() > 0)

System.out.println("'" + objClass.getSimpleName()

+ "' Field Class(es):");

Enumeration e = objectsToInspect.elements();

while (e.hasMoreElements()) {

Field f = (Field) e.nextElement();

System.out.println("Inspecting Field: '" + f.getName() + "'");

try {

System.out

.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

inspect(f.get(obj), recursive);

System.out

.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

} catch (NullPointerException nullExp) {

System.out.println("Field not instantiated at runtime");

System.out

.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

} catch (Exception exp) {

exp.printStackTrace();

}

}

System.out.println("End of '" + objClass.getSimpleName()

+ "' field classes");

}

}

**Unit Tests**

**Receiver Test.java**

import static org.junit.Assert.\*;

import java.io.FileOutputStream;

import java.io.PrintStream;

import org.jdom.Document;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

public class ReceiverTest {

simpleObject obj;

@Before

public void setUp() throws Exception {

obj = new simpleObject(1,true);

}

@After

public void tearDown() throws Exception {

}

@Test

//Tests to see that after you serialize an object and then deserialize it, you end up with

//essentially the same object.

public void testSerialize() throws Exception {

Document serializedDoc = Serializer.serialize(obj);

simpleObject deserializedObject = (simpleObject) Deserializer.deserialize(serializedDoc);

FileOutputStream fos1 = new FileOutputStream("testSerializeBefore.txt");

PrintStream ps = new PrintStream(fos1);

System.setOut(ps);

Inspector inspector = new Inspector();

inspector.inspect(obj, false);

FileOutputStream fos2 = new FileOutputStream("testSerializeAfter.txt");

ps = new PrintStream(fos2);

System.setOut(ps);

inspector.inspect(deserializedObject, false);

fos1.close();

fos2.close();

assertEquals(obj, deserializedObject);

}

}

**Sender Test.java**

import static org.junit.Assert.\*;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileOutputStream;

import java.io.FileReader;

import java.io.PrintStream;

import org.jdom.Document;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

public class SenderTest {

String testString;

@Before

public void setUp() throws Exception {

testString = "this is a test";

}

@After

public void tearDown() throws Exception {

}

@Test

//Test to see that the ObjectSerializer prints out the CMLDocument to a file in the proper format

public void testCMLDocOutput() throws Exception {

Document doc = Serializer.serialize(testString);

Sender.createFile(doc);

BufferedReader inStream = new BufferedReader(new FileReader("sentdata.xml"));

StringBuffer output = new StringBuffer();

while (inStream.ready())

output.append((char)inStream.read());

String result = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<serialized><object class=\"java.lang.String\" id=\"0\"><field name=\"value\" declaringclass=\"java.lang.String\"><reference>1</reference></field><field name=\"offset\" declaringclass=\"java.lang.String\"><value>0</value></field><field name=\"count\" declaringclass=\"java.lang.String\"><value>14</value></field><field name=\"hash\" declaringclass=\"java.lang.String\"><value>0</value></field><field name=\"serialVersionUID\" declaringclass=\"java.lang.String\"><value>-6849794470754667710</value></field><field name=\"serialPersistentFields\" declaringclass=\"java.lang.String\"><reference>2</reference></field><field name=\"CASE\_INSENSITIVE\_ORDER\" declaringclass=\"java.lang.String\"><reference>3</reference></field></object><object class=\"[C\" id=\"1\" length=\"14\"><value>t</value><value>h</value><value>i</value><value>s</value><value /><value>i</value><value>s</value><value /><value>a</value><value /><value>t</value><value>e</value><value>s</value><value>t</value></object><object class=\"[Ljava.io.ObjectStreamField;\" id=\"2\" length=\"0\" /><object class=\"java.lang.String$CaseInsensitiveComparator\" id=\"3\"><field name=\"serialVersionUID\" declaringclass=\"java.lang.String$CaseInsensitiveComparator\"><value>8575799808933029326</value></field></object></serialized>";

assertEquals(output.toString(), result);

}

}

**Refactors**

Long Function: Main function was basically all of the code in the Sender.java file so I broke it into several smaller parts. I extracted a method for each of the create Object functions as well as the serializer

**private** **static** **void** createCollection() **throws** IOException {

**boolean** flag = **false**;

**while** (!flag) {

System.*out*.println("add an object to collection?");

System.*out*.println("1. Yes");

System.*out*.println("2. No");

BufferedReader bufferRead = **new** BufferedReader(

**new** InputStreamReader(System.*in*));

String input = bufferRead.readLine();

ArrayList<simpleObject> list = **new** ArrayList<simpleObject>();

**if** (input.equals("1")) {

simpleObject obj = *createSimpleObject*();

list.add(obj);

} **else** **if** (input.equals("2")) {

Object obj = **new** collectionObject(list);

*objList*.add(obj);

flag = **true**;

} **else** {

System.*out*.println("Invalid input");

}

}

}

**private** **static** **void** createSimpleArray() **throws** IOException {

System.*out*.println("Enter array values. Eg: '1,2,3'");

BufferedReader bufferRead = **new** BufferedReader(**new** InputStreamReader(

System.*in*));

String input = bufferRead.readLine();

String[] fields = input.split(",");

**int**[] ints = **new** **int**[fields.length];

**for** (**int** i = 0; i < ints.length; i++) {

**try** {

ints[i] = Integer.*parseInt*(fields[i]);

} **catch** (Exception e) {

System.*out*.println("You can only enter numbers");

}

}

Object obj = **new** simpleArray(ints);

*objList*.add(obj);

}

**private** **static** **void** createRefArray() **throws** IOException {

System.*out*.println("Enter length of array");

BufferedReader bufferRead = **new** BufferedReader(**new** InputStreamReader(

System.*in*));

String input = bufferRead.readLine();

**int** length = Integer.*parseInt*(input);

simpleObject[] simpleObjArr = **new** simpleObject[length];

**for** (**int** i = 0; i < length; i++) {

simpleObject obj = *createSimpleObject*();

simpleObjArr[i] = obj;

}

refArray obj = **new** refArray(simpleObjArr);

*objList*.add(obj);

}

**private** **static** **void** createRefObject() **throws** IOException {

simpleObject simpleobj = *createSimpleObject*();

Object obj = **new** refObject(simpleobj);

*objList*.add(obj);

}

**private** **static** simpleObject createSimpleObject() **throws** IOException {

**boolean** flag = **false**;

**while** (!flag) {

System.*out*.println("Enter field values. Eg: '1,true'");

BufferedReader bufferRead = **new** BufferedReader(

**new** InputStreamReader(System.*in*));

String input = bufferRead.readLine();

String[] fields = input.split(",");

**if** (fields.length != 2) {

System.*out*.println("You must enter exactly 2 parameters");

} **else** {

**boolean** bool = **false**;

**if** (fields[1].toLowerCase().equals("true")) {

bool = **true**;

} **else** **if** (fields[1].toLowerCase().equals("false")) {

bool = **false**;

} **else** {

System.*out*

.println("You must enter true or false for a boolean");

}

**try** {

**int** i = Integer.*parseInt*(fields[0]);

simpleObject obj = **new** simpleObject(i, bool);

**return** obj;

} **catch** (Exception e) {

System.*out*.println("You must enter a number");

}

}

}

**return** **null**;

}

I then further broke down the serialize function into even smaller parts creating a function for file creation and file transfer.

**private** **static** File createFile(XMLOutputter out, Document doc)

**throws** IOException {

File aFile = **new** File("sentdata.xml");

BufferedWriter writer = **new** BufferedWriter(

**new** FileWriter(aFile));

out.output(doc, writer);

writer.close();

**return** aFile;

}

**private** **static** **void** transferFile(String server, **int** port, File aFile) {

System.*out*.println("Transferring file...");

**try** {

Socket s = **new** Socket(server, port);

OutputStream output = s.getOutputStream();

FileInputStream fileInputStream = **new** FileInputStream(

aFile);

**byte**[] buffer = **new** **byte**[1024 \* 1024];

**int** bytesRead = 0;

**while** ((bytesRead = fileInputStream

.read(buffer)) > 0) {

output.write(buffer, 0, bytesRead);

}

fileInputStream.close();

s.close();

System.*out*.println("Transfer Complete");

} **catch** (IOException e) {

System.*out*.println("connection refused");

}

}

In Receiver.java I broke up the main function into several functions. I created functions for receiving the files and building the jdom document.

**private** **static** **void** receiveFile(File aFile, Socket s) **throws** IOException,

FileNotFoundException {

InputStream input = s.getInputStream();

FileOutputStream out = **new** FileOutputStream(aFile);

**byte**[] buffer = **new** **byte**[1024 \* 1024];

**int** bytesReceived = 0;

System.*out*.println("receiving file");

**while** ((bytesReceived = input.read(buffer)) > 0) {

out.write(buffer, 0, bytesReceived);

System.*out*.println(bytesReceived + " Bytes received");

**break**;

}

}

**private** **static** Object buildDocument(File aFile) {

SAXBuilder builder = **new** SAXBuilder();

Object obj = **null**;

**try** {

Document doc = (Document) builder.build(aFile);

obj = Deserializer.*deserialize*(doc);

} **catch** (JDOMException e) {

e.printStackTrace();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** obj;

}

**Version Control Logs**