IOWA STATE UNIVERSITY

Serialization in Java (Binary and XML)

Kyle Woolcock

ComS 430

4/4/2014

Table of Contents

Introduction

Serialization is the process of converting objects to bytes that can then be used to reconstruct the original object. This process has many applications including remote procedure calls and allowing for persistent data. Serialization is a general programming concept, present in many object-oriented languages. This tutorial will focus on implementations in Java, and how Java handles serialization behind the scenes, but the general concepts can be applied to many languages. Alternatives to certain problems serialization solves to exist. For data persistence, often times databases are used where we just save the information the object stores instead of the object state itself. Java also supports the ability to serialize to XML which is more human readable, and allows for communication between programs written at different times and in different languages. To help read the document, it is important to note that all variable names appear in a different typeset, all class names are bolded, all exceptions are italicized, all method names are followed by parentheses, and all Java keywords appear in bold and italics.

Why Serialize?

Serialization allows for a quick and easy way to store data after a program finishes execution. The serialized data is independent of the Java virtual machine (JVM) that generated it. This means that as long as a different computer has access to the class files and the serialized data, the object can be reconstructed just as it originally was. It also allows for remote procedure calls. To call a method on another machine, often an object is needed argument. Serialization converts an object to a byte stream that can then be sent over a network and deserialized on the target machine.

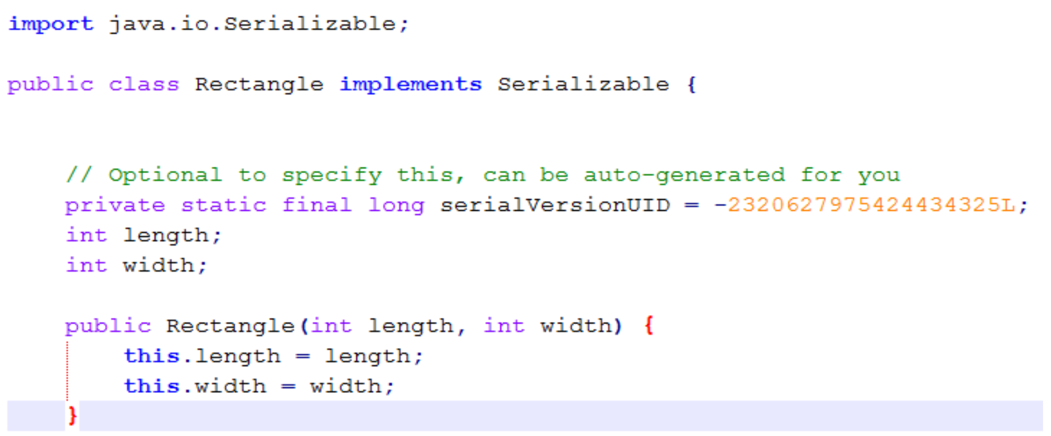
How to Serialize

Serializable Interface

Java provides two different ways to allow a class to be serialized. The first is to implement the Serializable interface. This is just a marker interface, meaning it contains no methods. Java will also implement a `serialVersionUID` variable, although it is advised manual assign the variable. It is the unique identifier Java uses to tell which class it is reconstructing from a byte stream (more on this later). This is the quickest and easiest way but gives you very little control over how the data is written. Figure 1 shows an implementation of Serializable with an example of a `serialVersionUID` variable. The one in the example was auto generated by Java, but set so that further changes to the class does not affect it. There will be more on this when we talk about versioning in the section on serialization problems.

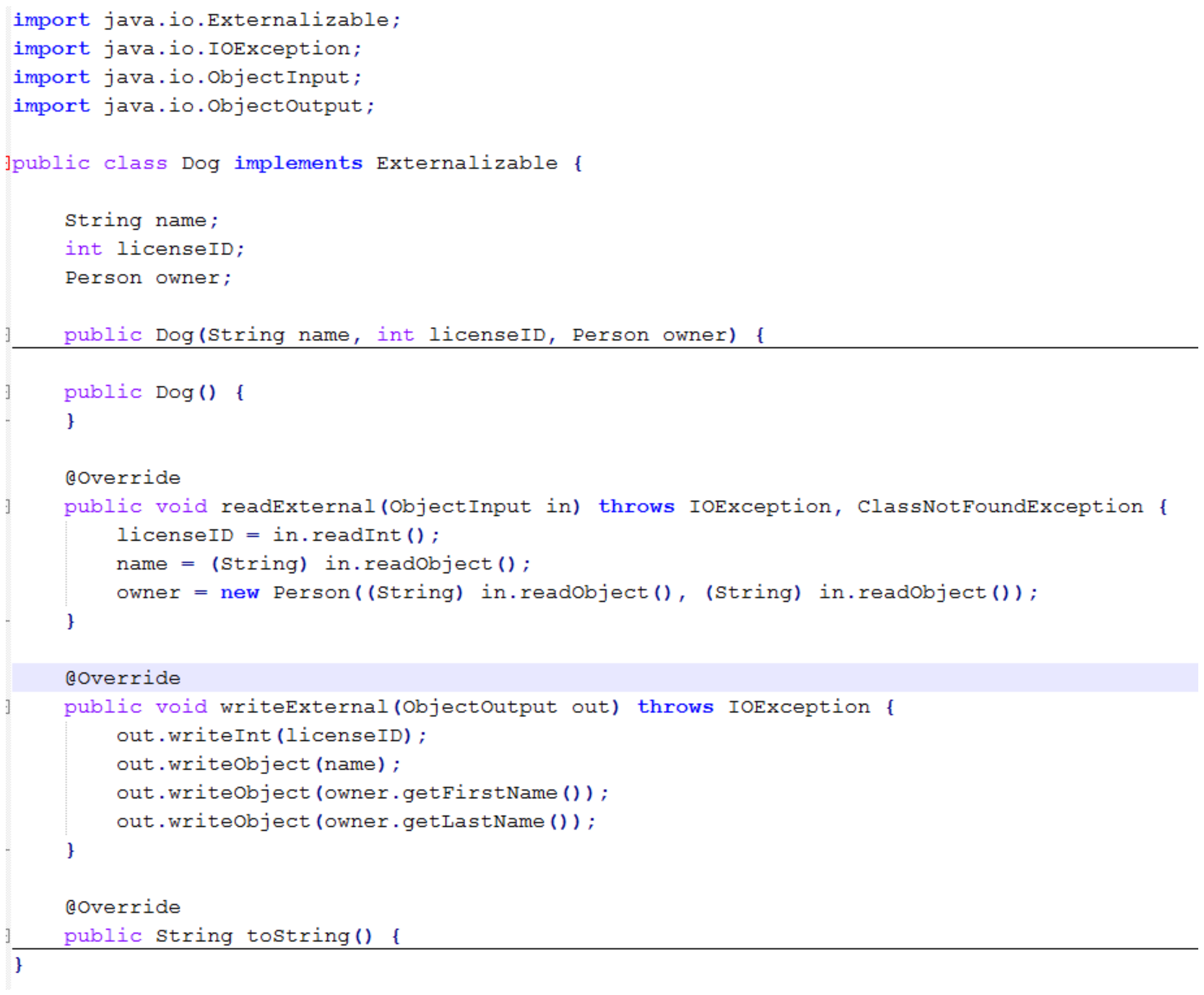
Externalizable Interface

Externalizable is an interface that extends Serializable. Unlike Serializable, Externalizable is not a marker interface. It requires an implementation of the methods readExternal() and writeExternal(). In addition to these methods, the class must also have a default constructor. This is because when using readExternal() and writeExternal(), a constructor is actually called for the object and then its variables are updates. This allows for a faster execution time than Serializable. To implement readExternal() and writeExternal() manually write the variables of the class to the output stream given. In Figure 2, there is an implementation of the Externalizable Interface along with the readExternal() and writeExternal().



```java
import java.io.Serializable;

public class Rectangle implements Serializable {


    // Optional to specify this, can be auto-generated for you
    private static final long serialVersionUID = -2320627975424434325L;
    int length;
    int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }
```

Figure 1

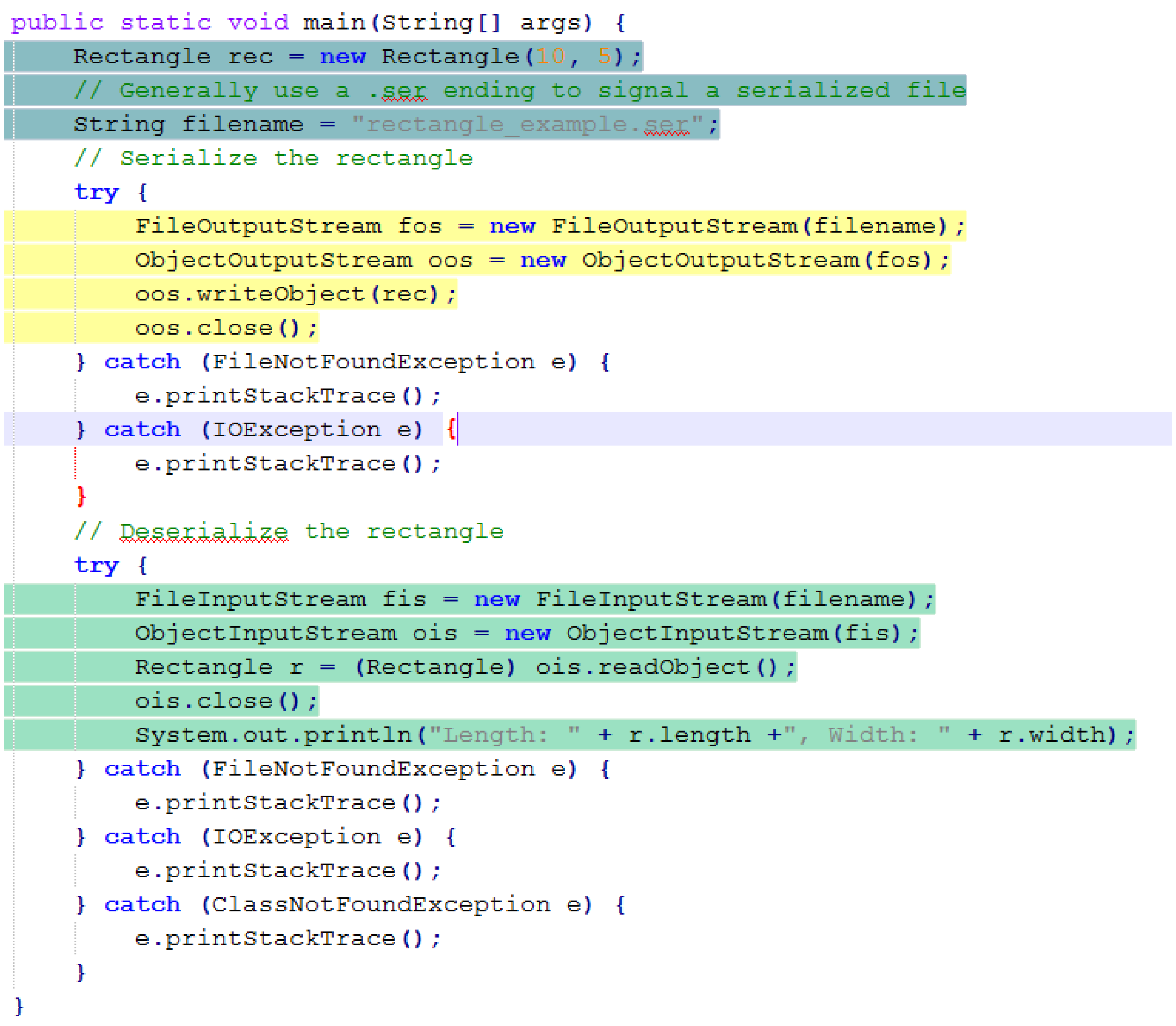


```java
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Dog implements Externalizable {

    String name;
    int licenseID;
    Person owner;

    public Dog(String name, int licenseID, Person owner) {

    public Dog() {
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        licenseID = in.readInt();
        name = (String) in.readObject();
        owner = new Person((String) in.readObject(), (String) in.readObject());
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(licenseID);
        out.writeObject(name);
        out.writeObject(owner.getFirstName());
        out.writeObject(owner.getLastName());
    }

    @Override
    public String toString() {
}
```

Figure 2

Using These Interfaces to Serialize

After implementing Serializable or Externalizable, Java provides two streams to read and write objects: ObjectOutputStream and ObjectInputStream. To create these streams, give them an instance of a file stream that was created with the file to be written to or read from. After doing that, ObjectOutputStream has various methods to write different objects and primitives to the file, notably writeObject(). In Figure 3, there is an example of a main method that is serializing and deserializing an instance of the rectangle class in Figure 1. The yellow highlighting shows the serialization steps whereas the teal highlighting shows the deserialization steps. Most of the exceptions are pretty standard, the only new one of note is *ClassNotFoundException*. This is thrown if the JVM cannot find a class with a `serialVersionUID` matching that of the one read from the file. This can happen if the class files needed to reconstruct the object are not found (either not present on your machine or not in the build path) or if there is a versioning problem, which will be discussed in a later section.



```java
public static void main(String[] args) {
    Rectangle rec = new Rectangle(10, 5);
    // Generally use a .ser ending to signal a serialized file
    String filename = "rectangle_example.ser";
    // Serialize the rectangle
    try {
        FileOutputStream fos = new FileOutputStream(filename);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(rec);
        oos.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    // Deserialize the rectangle
    try {
        FileInputStream fis = new FileInputStream(filename);
        ObjectInputStream ois = new ObjectInputStream(fis);
        Rectangle r = (Rectangle) ois.readObject();
        ois.close();
        System.out.println("Length: " + r.length +", Width: " + r.width);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

Figure 3

What Is Not Serialized?

Not everything can be serialized, and sometimes there is data that we don't want to be serialized for various reasons. Some native Java classes (like Thread for example) cannot be serialized. This is because they really wouldn't make sense after they were deserialized. //TODO EXPAND. Other things that are not serialized are the variables marked with the keyword ***transient***. This means that the variable should not be written to the byte stream. ***Transient*** is usually applied to variables that are do not implement Serializable or to variables we do not want to write for privacy reasons (think credit card data or social security numbers). Lastly, static variables are not serialized. This is because a static variable is the property of the class and not the specific instance we are serializing. When we deserialize the object, the static variable is still present in the class so we don't need to include it when we serialize.

Problems with Serializing

Versioning

There are a few problems that arise from serializing data. First, versioning can become a huge issue. For example, if someone created a serialized object, and then wanted to make a change to that class (say added another variable, or removed a method), Java might not know how to reconstruct that object. This is why manually managing the `serialVersionUID` is advisable. This is the number the JVM uses to decide which class it is trying to deserialize. If Java is left to manage the `serialVersionUID` it will recalculate it after the changes to the class and come up with a different number than was written with the serialized data. This results in a *ClassNotFoundException*. To avoid this, manually manage the `serialVersionUID` number by picking a unique number amongst the data to serialize to always represent that class. Now Java can always find the correct class but might create the class with null values for variables we added after serializing the object. This can cause some unexpected *NullPointerExceptions* if not properly accounted for.

Object References

Another common problem is object references within the object to be serialized. These are usually pointers to a location in memory where the object resides. It doesn't make sense to store these pointers sense when the data is deserialized, there is no guarantee that that object still exists there. This means that all the objects the objects to serialize depends on also need to be serialized (or marked transient). This happens through a process called pointer unswizzling which means we follow all the pointers the object to serialize, and serialize those objects as well. In Java, this is taken care of automatically, we just need to check that all the objects in our class are either serializable or marked as transient.

Serializing to XML

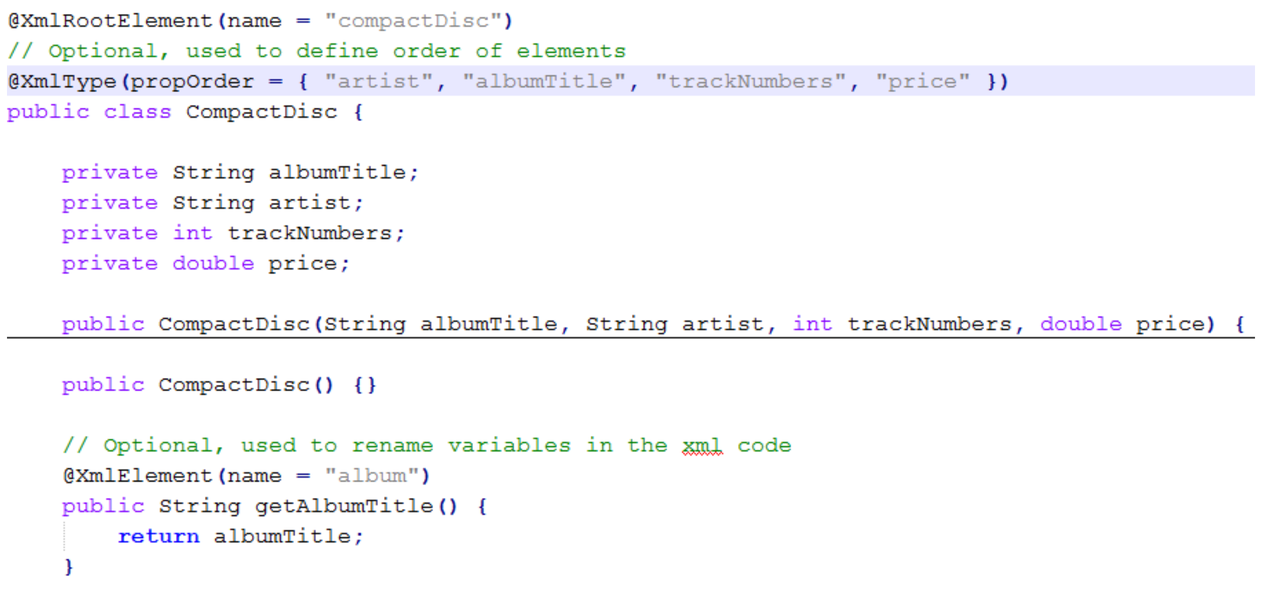
While standard serialization is extremely simple and effective for data storage and sending objects across a network, sometimes communication across programs or languages is desired. By converting an object to XML instead of bytes, we can then deserialize it in a wide variety of programs. It also has the added benefit of making the data human readable. There are many third party libraries that add support for XML serialization to Java, but Java has adopted one to come with the Java Development Kit (JDK). Since Java 1.6, Java Architecture for XML Binding has been included in the JDK. It provides a quick and easy way to produce a schema that we can use to convert objects to XML. To do this it uses Java annotations to mark out the elements. Since this is a basic introduction to XML serialization, only a few of the basic annotations are given; see Table 1.

@XmlRootElement	Specifies the root element
@XmlType	Allows you to specify the order of elements in the XML, among other things
@XmlElement	Allows you to rename an element (the default is the variable name)
@XmlElementWrapper	Creates a wrapper element to assist in schema building

Table 1

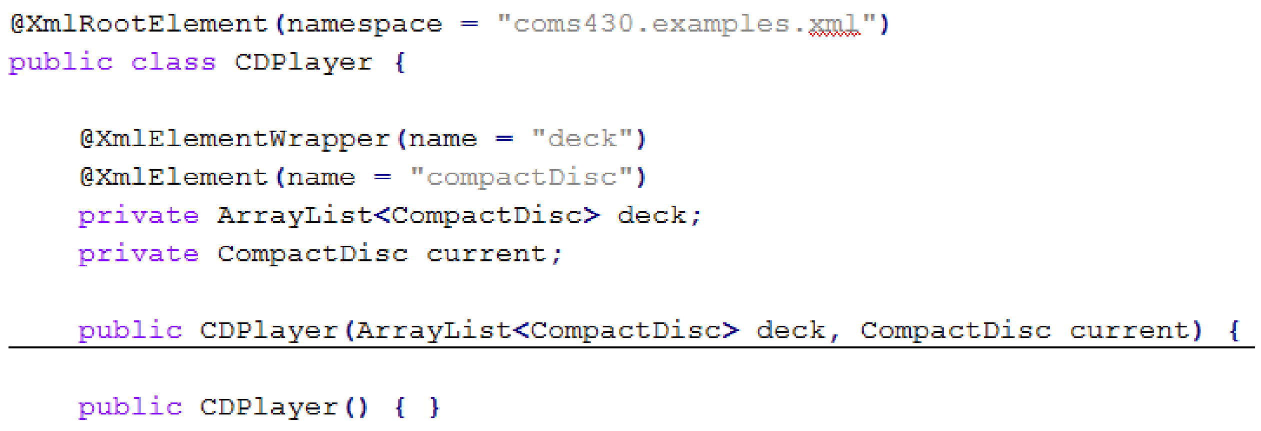
Creating Serializable Classes

Figure 4 and Figure 5, show an example of two classes that use the XML annotations. Figure 4 is a **CompactDisc** class that shows how to use the @XmlType annotation to change the order the variables are written (default is the order they appear in the code) as well as how to use the @XmlElement annotation to change the name of a variable in the XML code. Figure 5 is a **CDPlayer** class which contains a deck of **CompactDisc**s. It demonstrates the @XmlElementWrapper to assist with our schema. Both of these examples have a default constructor. This default constructor is necessary to deserialize the data, just like in the Externalizable interface implementation in Figure 2. Both of these figures are just partial implementations with all of the getters and setters left off to save space. There is nothing special about these methods and they would be written normally.



```java
@XmlRootElement(name = "compactDisc")
// Optional, used to define order of elements
@XmlType(propOrder = { "artist", "albumTitle", "trackNumbers", "price" })
public class CompactDisc {

    private String albumTitle;
    private String artist;
    private int trackNumbers;
    private double price;

    public CompactDisc(String albumTitle, String artist, int trackNumbers, double price) {

    public CompactDisc() {}

    // Optional, used to rename variables in the xml code
    @XmlElement(name = "album")
    public String getAlbumTitle() {
        return albumTitle;
    }
```

Figure 4



```
@XmlRootElement(namespace = "coms430.examples.xml")
public class CDPlayer {

    @XmlElementWrapper(name = "deck")
    @XmlElement(name = "compactDisc")
    private ArrayList<CompactDisc> deck;
    private CompactDisc current;

    public CDPlayer(ArrayList<CompactDisc> deck, CompactDisc current) {

    public CDPlayer() { }
```

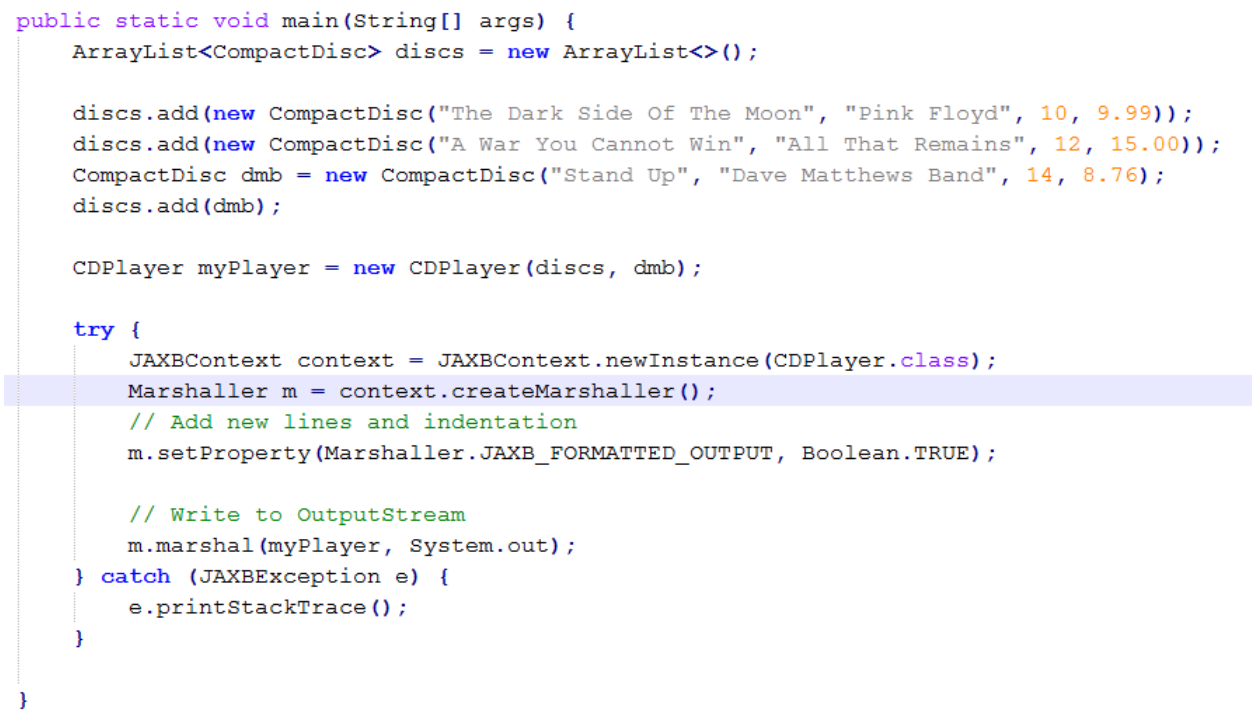
Figure 5

Using These Classes to Serialize

After creating and annotating the classes above, the final step is to create a main method to actually serialize the data. To do this, create an instance of **CDPlayer** as well as some **CompactDisc**s to put in the deck. After setting up the objects, create a **JAXBContext**. Again this is an introduction to serialization so a lot of what the class does will be left for more advanced discussions, but for now, give it the class we plan on serializing (in this example CDPlayer.class). After creating a **JAXBContext** we need to create a **Marshaller**. To do this, just call createMarshaller() on the **JAXBContext** created above. In this example, also use the setProperty() method of the marshaller to set **Marshaller**.`JAXB_FORMATTED_OUTPUT` to true. This step is optional but it automatically adds newlines and tabs to the XML code making it easier to read. Lastly, use the marshal() method of the marshaller to serialize the given object to the given output stream (in this example, System.out). While technically not synonymous, for the purposes of this tutorial, serialize and marshal are equivalent. Figure 6 shows the entire main method for serializing our **CompactDisc** and **CDPlayer** classes to XML.

Conclusion

Serialization is a quick and easy way to keep persistant data as well as use remote procedure calls. Java has very good native support for serialization and a class can be made serializable in a matter of minutes. When cross program and/or cross language communication is necessary, serialization to XML is a better option. This allows for the data to be stored in a standardized way that other programs and languages can understand.



```
public static void main(String[] args) {
    ArrayList<CompactDisc> discs = new ArrayList<>();

    discs.add(new CompactDisc("The Dark Side Of The Moon", "Pink Floyd", 10, 9.99));
    discs.add(new CompactDisc("A War You Cannot Win", "All That Remains", 12, 15.00));
    CompactDisc dmb = new CompactDisc("Stand Up", "Dave Matthews Band", 14, 8.76);
    discs.add(dmb);

    CDPlayer myPlayer = new CDPlayer(discs, dmb);

    try {
        JAXBContext context = JAXBContext.newInstance(CDPlayer.class);
        Marshaller m = context.createMarshaller();
        // Add new lines and indentation
        m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);

        // Write to OutputStream
        m.marshal(myPlayer, System.out);
    } catch (JAXBException e) {
        e.printStackTrace();
    }

}
```

Figure 6

Appendix A: Table of Figures

Appendix B: Acknowledgements

Most examples are adapted from similar examples from Lars Vogel's articles.

Works Cited

Mandliya, Arpit. *Java Tutorial for Beginners*. 10 March 2013. 31 March 2014.

Oracle. *The Java Tutorials: Serializable Objects*. n.d. 1 April 2014.

Paul, Javin. *Javarevisited: Top 10 Java Serialization Intervew Questions and Answers*. 16 April 2011. 2 April 2014.

Vogel, Lars. *Vogel/a: Java Object Serialization - Tutorial*. 4 Febuary 2014. 1 April 2014.

—. *Vogel/a: JAXB - Tutorial*. 23 November 2012. 1 April 2014.