Object Serialization

Online References

Java Object Serialization Specification:
http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html

5 Things you didn't know about Object Serialization:
http://www.ibm.com/developerworks/library/j-5things1/

Overview

- Serialization
- readObject, writeObject
- readResolve, writeReplace

Serialization

Serialization

```
public class CustomerInformation {
    String name;
    int id;

    CustomerInformation(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String toString() {
        return name + " : " + id;
    }
}
```

Suppose we have a CustomerInformation class (or any other class) that we would like to be able to transfer over a Socket (or any ObjectInputStream).

Serialization

```
import java.io.*;
import java.net.*;

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

        if (args.length != 4) {
            System.err.println(
                "Usage: java EchoClient <host name> <port number> <client name> <client id>");
            System.exit(1);
        }

        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);

        try {
            Socket echoSocket = new Socket(hostName, portNumber);

            ObjectOutputStream out = new ObjectOutputStream(echoSocket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(echoSocket.getInputStream());

            out.writeObject( new CustomerInformation(args[2], Integer.parseInt(args[3])) );
            CustomerInformation response = (CustomerInformation)in.readObject();
            System.out.println("Received customer information: " + response);

        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                hostName);
            System.exit(1);
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException caught: " + e);
            e.printStackTrace();
         }

    }
}
```

With some modifications to the EchoClient, we can instead use ObjectOutputStream and ObjectInputStream to write and read a class over a socket.

Serialization

```
import java.net.*;
import java.io.*;

public class SerializationServer {
    public static void main(String[] args) throws IOException {
       if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try {
            ServerSocket serverSocket = new ServerSocket(portNumber);

            System.out.println("The server is listening at: " + serverSocket.getInetAddress() + " on port " +
serverSocket.getLocalPort());

            Socket clientSocket = serverSocket.accept();
            ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(clientSocket.getInputStream());

            CustomerInformation info = (CustomerInformation)in.readObject();

            System.out.println("Received customer information: " + info);

            info.id *= 2;
            info.name = info.name.toUpperCase();

            out.writeObject(info);

        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException caught: " + e);
            e.printStackTrace();
        }
    }
}
```

With some modifications to the EchoClient, we can instead use ObjectOutputStream and ObjectInputStream to write and read a class over a socket.

ClassNotFoundException

```
import java.net.*;
import java.io.*;

public class SerializationServer {
    public static void main(String[] args) throws IOException {
       if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try {
            ServerSocket serverSocket = new ServerSocket(portNumber);

            System.out.println("The server is listening at: " + serverSocket.getInetAddress() + " on port " +
serverSocket.getLocalPort());

            Socket clientSocket = serverSocket.accept();
            ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(clientSocket.getInputStream());

            CustomerInformation info = (CustomerInformation)in.readObject();

            System.out.println("Received customer information: " + info);

            info.id *= 2;
            info.name = info.name.toUpperCase();

            out.writeObject(info);

        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException caught: " + e);
            e.printStackTrace();
        }
    }
}
```

Note that when you write or read an object from a socket you also need to catch a ClassNotFoundException, in the case the object read is not available to the JVM (i.e., it's not in the JVMs classpath).

ClassNotFoundException

```
import java.net.*;
import java.io.*;

public class SerializationServer {
    public static void main(String[] args) throws IOException {
       if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try {
            ServerSocket serverSocket = new ServerSocket(portNumber);

            System.out.println("The server is listening at: " + serverSocket.getInetAddress() + " on port " +
serverSocket.getLocalPort());

            Socket clientSocket = serverSocket.accept();
            ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(clientSocket.getInputStream());

            CustomerInformation info = (CustomerInformation)in.readObject();
            System.out.println("Received customer information: " + info);

            info.id *= 2;
            info.name = info.name.toUpperCase();

            out.writeObject(info);

        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException caught: " + e);
            e.printStackTrace();
        }
    }
}
```

Also, Java doesn't know what class readObject returns (the return type is just Object), so you need to cast it the correct type. Using the **instanceof** operator can be immensely useful here if you could potentially receive objects with different types.

Problems

When we try to run this, we get some errors. The client reponds:

```
$ java SerializationClient localhost 4444 "Travis Desell" 2398423
Couldn't get I/O for the connection to localhost
```

And the actual problem shows up on the server side:

```
$ java SerializationServer 4444
The server is listening at: 0.0.0.0/0.0.0.0 on port 4444
Exception caught when trying to listen on port 4444 or listening for a connection
writing aborted; java.io.NotSerializableException: CustomerInformation
```

The CustomerInformation class is not serializable, so an exception is thrown.

Fixes

```
public class CustomerInformation implements Serializable {
    String name;
    int id;

    CustomerInformation(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String toString() {
        return name + " : " + id;
    }
}
```

The solution is deceptively simple. The class simply needs to implement the java.io.Serializable class and you’re good to go.

Fixes

```
public class CustomerInformation implements Serializable {
    String name;
    int id;
    String someVeryLargeArray[1000000];
    SomeOtherObject anotherObject;

    CustomerInformation(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String toString() {
        return name + " : " + id;
    }
}
```

When you write a serializable Object over a stream, it will copy every field. Fields that refer to objects are also copied. This is a *deep copy*, and it might not be desirable as an object and everything it refers to is copied. This could also cause NotSerializableExceptions if an object is referred to that doesn't implement Serializable.

Overloading readObject and writeObject

readObject, writeObject

```
 1 import java.io.*;
 2
 3 public class OverloadedCustomerInformation implements java.io.Serializable {
 4     String name;
 5     int id;
 6
 7     OverloadedCustomerInformation(String name, int id) {
 8         this.name = name;
 9         this.id = id;
10     }
11
12     public String toString() {
13         return name + " : " + id;
14     }
15
16     private void writeObject(ObjectOutputStream out) throws IOException {
17         System.out.println("Inside writeObject");
18         out.defaultWriteObject();
21         System.out.println("finished writeObject");
22     }
23
24     private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
25         System.out.println("Inside readObject");
26         in.defaultReadObject();
29         System.out.println("finished readObject");
30     }
31 }
```

You can explicitly specify what is written and read over the stream by overloading the readObject and writeObject methods. Note that these also will need to take care of any information that needs to be sent by superclasses.

readObject, writeObject

```
 1 import java.io.*;
 2
 3 public class OverloadedCustomerInformation implements java.io.Serializable {
 4     String name;
 5     int id;
 6
 7     OverloadedCustomerInformation(String name, int id) {
 8         this.name = name;
 9         this.id = id;
10     }
11
12     public String toString() {
13         return name + " : " + id;
14     }
15
16     private void writeObject(ObjectOutputStream out) throws IOException {
17         System.out.println("Inside writeObject");
18         out.defaultWriteObject();
21         System.out.println("finished writeObject");
22     }
23
24     private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
25         System.out.println("Inside readObject");
26         in.defaultReadObject();
29         System.out.println("finished readObject");
30     }
31 }
```

defaultReadObject and defaultWrite object do the same thing as Java's default object serialization. Sometimes you want to use them if you have a subclass which might call readObject or writeObject and you don't want to implement the methods yourself.

readObject, writeObject

```
 1 import java.io.*;
 2
 3 public class OverloadedCustomerInformation implements java.io.Serializable {
 4     String name;
 5     int id;
 6
 7     OverloadedCustomerInformation(String name, int id) {
 8         this.name = name;
 9         this.id = id;
10     }
11
12     public String toString() {
13         return name + " : " + id;
14     }
15
16     private void writeObject(ObjectOutputStream out) throws IOException {
17         System.out.println("Inside writeObject");
19         out.writeObject(name);
20         out.writeInt(id);
21         System.out.println("finished writeObject");
22     }
23
24     private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
25         System.out.println("Inside readObject");
27         this.name = (String)in.readObject();
28         this.id = in.readInt();
29         System.out.println("finished readObject");
30     }
31 }
```

Alternately you can specify which objects you read/write over the stream explicitly. Which can significantly improve performance because you only transfer the things you want. Note that readObject will have called the default constructor to the object before readObject is called.

Using readResolve, writeReplace

readResolve, writeReplace

```
import java.io.*;

public class ProxyCustomerInformation implements java.io.Serializable {
    String name;
    int id;

    ProxyCustomerInformation(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String toString() {
        return name + " : " + id;
    }

    public Object writeReplace() throws java.io.ObjectStreamException {
        System.out.println("Replacing proxy customer information with serialized name, " + name + " and id: " + id);
        return new SerializedCustomerInformation( name, id );
    }

    public static class SerializedCustomerInformation implements java.io.Serializable {
        String name;
        int id;

        SerializedCustomerInformation(String name, int id) {
            this.name = name;
            this.id = id;
        }

        public Object readResolve() throws java.io.ObjectStreamException {
            System.out.println("Resolving a serailized customer information with name, " + name + " and id: " + id);
            return new ProxyCustomerInformation(this.name, this.id);
        }
    }
}
```

Sometimes you don't actually want to transfer the object. Also, note that serialization *copies* the object. So if you serialize an object to a remote server, then serialize it back you will have a duplicate of the object locally.

readResolve, writeReplace

```
import java.io.*;

public class ProxyCustomerInformation implements java.io.Serializable {
    String name;
    int id;

    ProxyCustomerInformation(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String toString() {
        return name + " : " + id;
    }

    public Object writeReplace() throws java.io.ObjectStreamException {
        System.out.println("Replacing proxy customer information with serialized name, " + name + " and id: " + id);
        return new SerializedCustomerInformation( name, id );
    }

    public static class SerializedCustomerInformation implements java.io.Serializable {
        String name;
        int id;

        SerializedCustomerInformation(String name, int id) {
            this.name = name;
            this.id = id;
        }

        public Object readResolve() throws java.io.ObjectStreamException {
            System.out.println("Resolving a serailized customer information with name, " + name + " and id: " + id);
            return new ProxyCustomerInformation(this.name, this.id);
        }
    }
}
```

You can use readResolve and writeReplace to create a replacement serializable object to do the transfer.

readResolve, writeReplace

```
import java.io.*;

public class ProxyCustomerInformation implements java.io.Serializable {
    String name;
    int id;

    ProxyCustomerInformation(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String toString() {
        return name + " : " + id;
    }

    public Object writeReplace() throws java.io.ObjectStreamException {
        System.out.println("Replacing proxy customer information with serialized name, " + name + " and id: " + id);
        return new SerializedCustomerInformation( name, id );
    }

    public static class SerializedCustomerInformation implements java.io.Serializable {
        String name;
        int id;

        SerializedCustomerInformation(String name, int id) {
            this.name = name;
            this.id = id;
        }

        public Object readResolve() throws java.io.ObjectStreamException {
            System.out.println("Resolving a serailized customer information with name, " + name + " and id: " + id);
            return new ProxyCustomerInformation(this.name, this.id);
        }
    }
}
```

Note that this creates a new object which will be written to the stream (note you can overload readObject and writeObject in the newly created object too). This can be very useful if you don't want the Object to be copied or if you do things in the default constructor which shouldn't be done again.

readResolve, writeReplace

```
import java.io.*;

public class ProxyCustomerInformation implements java.io.Serializable {
    String name;
    int id;

    ProxyCustomerInformation(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String toString() {
        return name + " : " + id;
    }

    public Object writeReplace() throws java.io.ObjectStreamException {
        System.out.println("Replacing proxy customer information with serialized name, " + name + " and id: " + id);
        return new SerializedCustomerInformation( name, id );
    }

    public static class SerializedCustomerInformation implements java.io.Serializable {
        String name;
        int id;

        SerializedCustomerInformation(String name, int id) {
            this.name = name;
            this.id = id;
        }

        public Object readResolve() throws java.io.ObjectStreamException {
            System.out.println("Resolving a serailized customer information with name, " + name + " and id: " + id);
            return new ProxyCustomerInformation(this.name, this.id);
        }
    }
}
```

This does, however, require that you make an entirely new class. However if you are moving lots of objects around and you want to make sure that there is only one copy of each (for distributed garbage collection or other reasons) this approach can be very useful.