
Workbook 2

Introduction

Last week you wrote a simple chat client in Java. One of the problems you may have encountered is that different users formatted their messages in different ways and consequently your chat client couldn't easily format all the messages in a uniform manner. This week you will explore Java's mechanism for saving and restoring object state either to and from a stream, allowing an object to be saved to disk or sent between machines on a computer network. You will use this mechanism to write a new and improved Java chat client which sends Java objects as structured messages between the client and the server. This technique will allow you to handle and display messages in a more structured way.

Important

An on-line version of this guide is available at:

<http://www.cl.cam.ac.uk/teaching/0910/FJava>

You should check this page regularly for announcements and errata. You might find it useful to refer to the on-line version of this guide in order to follow any provided web links or to cut 'n' paste example code.

Serialisation

By default, data stored by a Java program only exists for as long as the program remains in computer memory. When the program terminates (cleanly or due to an error) all data held in computer memory is lost. This can be problematic, since it is often useful to be able to save and restore data between executions of a computer program. Java serialisation offers an easy way to save and restore data by enabling an instance of a Java object to be turned into a platform-independent sequence of bytes which can be written to the hard disk or sent over a computer network.

In Java you can serialise any object which implements the `java.io.Serializable` interface (paying careful attention to the spelling¹). The interface does not contain any fields nor does it proscribe the creation of any methods—it's simply used to denote the class as *serialisable*. The Java runtime is able to convert an instance of an object into a stream of bytes (and back again), so as a programmer you need to do nothing other than declare that the class implements the interface `java.io.Serializable`. Any serialised object is written in a platform-independent manner in the sense that an instance of a class which is serialised on one machine can be deserialised on another, regardless of the underlying operating system or Endianness of the computers involved.

Here is a simple example of a class which implements the `Serializable` interface:

```
class Message implements Serializable {
    int id;
    String msg;
    Message(int id, String msg) {
        this.id = id;
        this.msg = msg;
    }
}
```

The field `id` is of type `int` and, when an instance of the class `Message` is serialised, the Java runtime will save a copy of the value stored in the field. Handling the field `msg` is more complicated however

¹The use of "-ise" verses "-ize" is much debated and is some times erroneously attributed to a difference between American and British English. Oxford University Press is said to favour "-ize" whilst Cambridge University Press prefers "-ise" (http://en.wikipedia.org/wiki/American_and_British_English_spelling_differences). As this course is taught in Cambridge, we'll use "-ise".

since it is a *reference* to an instance of the `String` class. The Java runtime system knows this and will serialise the field `msg` by serialising the instance of the `String` class which the field references (if any). In general, when the Java runtime system is serialising an object it will recursively serialise all the referenced objects so that a complete copy of all the relevant data is captured.

Some classes, such as the `Socket` class you used last week, cannot be serialised since they represent a state, such as a TCP/IP connection, which cannot be saved to disk and restored (possibly on a different computer) at an arbitrary point in the future. If you need to serialise a class which contains a reference to an object which cannot be serialised, you must declare the field to be `transient`; if you do so, such fields will not be saved and therefore must be manually recreated by the programmer afterwards.

One potential problem with serialising instances of classes arises when you wish to change the class definition. For example, you may wish to add a field to the class, change the inheritance hierarchy or move the class between packages. Such changes may prevent any existing instances of the class which have been serialised to disk from being deserialised correctly. In order to detect such problems, the compiler generates a unique identifier for a particular class by computing a hash of the class definition and storing this along with the rest of the data. Thus any changes made to the class will result in a different identifier, and therefore the Java runtime can detect whether the version of the class definition currently available is the same as the version used when the object was serialised.

You can manually control the version identifier given to a class by declaring the following field inside the class and providing a specific integer value (we recommend starting at one):

```
private static final long serialVersionUID = ...
```

It's a good idea to do this by default (not only because Eclipse encourages you to do so!) but because it is possible to add (or remove) fields to a class and still restore instances of the class which were serialised before the fields were added (removed). In the cases where additional fields are added, they are initialised to a default value (e.g. `null`); when fields are removed, the associated serialised data is simply ignored. You must change the version identifier in most other cases, for example if you change the class hierarchy or the class package. The Java Serialization Specification² contains the gory details.

Java provides two classes to help you serialise an object into bytes or deserialise bytes into an object. These are `ObjectOutputStream` and `ObjectInputStream` respectively. Here is a small example code snippet which serialises an instance of the `Message` object as defined above and writes it to a file called `message.jobj`:

```
FileOutputStream fos = new FileOutputStream("message.jobj");
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject(new Message(1, "Hello, world"));
out.close();
```

A quick glance at the Java documentation reveals that `FileOutputStream` and `ObjectOutputStream` interact with each other in the above code as byte-oriented streams of data. Formally, they *provide* an `OutputStream` and *use* an `OutputStream` respectively; you've seen this interaction before with `BufferedReader`. This loose coupling of classes enables Java objects to be written to any class which provides an `OutputStream`. In the last workbook you used the `OutputStream` provided by instances of the `Socket` class. Many other classes in the Java standard library which read or write data support either an `InputStream` or an `OutputStream` respectively, a fact you will need to remember when completing this workbook!

Important

Further information on serialisation is available at:

<http://java.sun.com/developer/technicalArticles/Programming/serialization/>

²<http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html>

Serialising a simple class

Here is a simple class which implements the serialisation interface:

```
package uk.ac.cam.crsid.fjava.tick2;

import java.io.Serializable;

public class TestMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    private String text;
    public String getMessage() {return text;}
    public void setMessage(String msg) {text = msg;}
}
```

1. Create a new package inside your Eclipse "Further Java" project with the appropriate name, and places the class `TestMessage` as defined above inside it.
2. Complete the sections marked `TODO` in the class `TestMessageReadWrite` shown below.
3. Test your implementation by instructing your program to download and print out the message contained within a serialised instance of `TestMessage` at the following URL: <http://www.cl.cam.ac.uk/teaching/0910/FJava/testmessage-crsid.jobj>

```
package uk.ac.cam.crsid.fjava.tick2;
//TODO: import required classes

class TestMessageReadWrite {

    static boolean writeMessage(String message, String filename) {
        //TODO: Create an instance of "TestMessage" with "text" set
        //      to "message" and serialise it into a file called "filename".
        //      Return "true" if write was successful; "false" otherwise.
    }

    static String readMessage(String location) {
        //TODO:
        // If "location" begins with "http://" then attempt to download
        // and deserialise an instance of TestMessage; you should use
        // the java.net.URL and java.net.URLConnection classes.
        // If "location" does not begin with "http://" attempt to
        // deserialise an instance of TestMessage by assuming that
        // "location" is the name of a file in the filesystem.
        //
        // If deserialisation is successful, return a reference to the
        // field "text" in the deserialised object. In case of error,
        // return "null".
    }

    public static void main(String args[]) {
        //TODO: Implement suitable code to help you test your implementation
        //      of "readMessage" and "writeMessage".
    }
}
```

A chat client using Java objects

In the previous section you practised serialising and deserialising Java objects using the `TestMessage` class. In this section you will adapt your Java chat client from last week to support the sending and receiving of Java objects over a `Socket` object, rather than simply sending text between the client and the server.

The sending of serialised classes between client and server has both benefits and drawbacks. A major benefit of the scheme is that the data is *structured* since the type of an object can be used to represent the type of message sent from the client to the server. One drawback with the scheme is that it is an onerous task to write either the client or the server in any language other than Java, since the format of serialised Java objects is quite complex, and most other languages will not provide support by default. This drawback can be overcome by designing a simple structured binary or text format which can be read or written by programs in any language. In the interests of improving your understanding of serialisation, and also in the interests of brevity of both client and server software, we will continue to use serialised classes in this course, and leave the design and implementation of a more portable format as an optional exercise for the reader over the Christmas vacation.³

There are four types of messages, and therefore four Java classes, used in the new version of the Java chat server. Two of the classes are sent from the client to the server, and the remaining two are sent by the server to the client. These messages are summarised in Table 1, "Message classes sent between the client and server". All these classes inherit from a fifth class called `Message`. Instances of the class `Message` itself are never sent between the server and the client.

Message type (class)	Direction	Description
<code>ChangeNickMessage</code>	Client→Server	Update nickname of the client stored by the server.
<code>ChatMessage</code>	Client→Server	Message written by a user is sent to the server.
<code>RelayMessage</code>	Server→Client	User message sent from server to all clients.
<code>StatusMessage</code>	Server→Client	Message generated by the server, sent to all clients.

Table 1. Message classes sent between the client and server

Serialisation

4. Download a copy of the classes which define the message types from: <http://www.cl.cam.ac.uk/teaching/0910/FJava/messages.jar>.
5. Open the jar file you downloaded in the previous step using the command line tool `jar` to extract the Java source code for the messages. There are additional classes contained in this Jar file which will be explained later in this workbook.
6. Create a new package and class files with appropriate names in your "Further Java" project to contain the five message types found in the jar file.

Your next task today is to update the code you wrote last week to communicate with a new server which supports serialised objects rather than textual strings. To provide a feel for the use of each of the message types described above, a sample chat session shown in Figure 1, "A chat session between Dave and Hal".

³This suggestion is somewhat tongue-in-cheek, and is certainly not compulsory, but the really keen student should read existing specifications before designing their own. The XMPP RFCs (<http://xmpp.org/rfc/>) are a good place to start.

```
crsid@machine:~> java -jar crisd-tick2.jar
14:23:27 [Client] Connected to java-lb.cl.cam.ac.uk on port 15003.
14:23:27 [Server] Anonymous15983 connected from evapod.discovereveryone.space.
\nick Dave
14:23:29 [Server] Anonymous15983 is now known as Dave.
14:23:14 [Server] Anonymous82791 connected from cpu9000.discovereveryone.space.
14:23:17 [Server] Anonymous82791 is now known as Hal.
Hello, Hal. Do you read me, Hal?
14:23:22 [Dave] Hello, Hal. Do you read me, Hal?
14:23:27 [Hal] Affirmative, Dave. I read you.
Open the pod bay doors, Hal.
14:23:31 [Dave] Open the pod bay doors, Hal.
14:23:36 [Hal] I'm sorry, Dave. I'm afraid I can't do that.
Why not, Hal? What's the problem?
14:23:39 [Dave] Why not, Hal? What's the problem?
14:23:43 [Hal] I think you know what the problem is just as well as I do.
What are you talking about, Hal?
14:23:50 [Dave] What are you talking about, Hal?
14:23:53 [Hal] This mission is too important for me to allow you to jeopardise it.
I don't know what you're talking about.
14:23:59 [Dave] I don't know what you're talking about.
\destroy Hal
14:24:02 [Client] Unknown command "destroy"
14:24:06 [Hal] I know you and Frank were planning to disconnect me.
14:24:08 [Hal] And that's something I cannot allow to happen.
14:24:12 [Server] Hal has disconnected.
\quit
14:24:17 [Client] Connection terminated.
crsid@machine:~>
```

Figure 1. A chat session between Dave and Hal⁴

In the sample chat session, the user (Dave) is talking to another user (Hal). All four message types were used to support this chat session. Dave starts the chat session by running your Java chat client, which prints the first line stating that it has successfully connected to the server. On receipt of a new connection the server sent all clients, including the client who just connected, a `StatusMessage` object stating that a new user (currently called `Anonymous15983`) has connected from the machine `evapod.discovereveryone.space`.

Your implementation of the Java chat client should interpret any user input which begins with a backslash ("`\`") in a special way. For example, in the third line of the chat session, Dave types `\nick Dave`; in this case `\nick` instructs your client to send a `ChangeNickMessage` object with the new name stored in the object set to `Dave`. Similarly, in the penultimate line, Dave types `\quit` and the client closes the connection to the server and terminates. Shortly after 14:23:59 Dave types `\destroy Hal`, but unfortunately your client does not understand the instruction `\destroy` and therefore the client sends no data to the server at all; it simply prints an error message. In your implementation of the Java chat client, you need only support two commands `\nick` and `\quit`.

At some point after 14:23:17, the user in this chat session (Dave) types in his first regular message (`Hello, Hal. Do you read me, Hal?`). When Dave hits the enter key, the client sends the message as a `ChatMessage` object to the server, and the server sends this message on to all clients (including Dave) as a `RelayMessage` object; this is received by your client and printed to the screen at 14:23:22.

The server will send messages of type `RelayMessage` and `StatusMessage`. Therefore you will need to determine at runtime the type of any message object you receive. You can do this by using the infix

⁴A quote from the 1968 epic "*2001: A Space Odyssey*". Directed by Stanley Kubrick, and written by Arthur C. Clarke and Stanley Kubrick.

operator `instanceof`; for example `(m instanceof StatusMessage)` will evaluate to `true` if `m` is an instance of the `StatusMessage` class and `false` otherwise.

A new Java chat client

7. Look at the Java documentation for the `java.text.SimpleDateFormat` and the `java.util.Date` classes and work out how to print the current time in the same format as shown in the Figure.
8. Create the class `ChatClient` inside the package `uk.ac.cam.crsid.fjava.tick2` which implements a Java chat client as described earlier in this section. Your class should provide a standard `public static void main(String[] args)` method which accepts two arguments on the command line: a server name and a port number; errors (such as insufficient arguments) should be handle as you did in your implementation of `StringChat` last week.
9. Test your implementation using the server `java-1b.cl.cam.ac.uk` on port number 15003.

Class loaders and reflection

Java programs run on top of a Java Virtual Machine (JVM) rather than compiling directly to machine code, therefore the JVM can control when and how new pieces of program code (i.e. classes) are loaded and executed. The *class loader* is the part of the JVM responsible for load class definitions. By default the class loader will look for `.class` files at various locations on disk and inside Jar files when an instance of a class (or a static field in a class) is first referenced; however it is also possible to extend the Java class loader to load definitions of classes at runtime from other locations, for example from a website (e.g. Java applet) or over a `Socket` object.

One of the difficulties with using serialisation for sending messages between machines in a distributed computing scenario is that all the machines must have a common definition of the class. Without such a definition, you cannot deserialise the object from the `ObjectOutputStream`; instead the stream throws a `ClassNotFoundException`. To get around this problem in your implementation of `ChatClient`, you will extend the Java class loader to dynamically load the definition of a new class at runtime into the JVM. With the new class definition loaded into the JVM, it is then possible to deserialise an instance of this new class and upgrade the functionality of your program as it runs!

To support dynamic updates, `uk.ac.cam.cl.fjava.messages` contains two additional classes you have not used so far. Take a look at `NewMessageType.java` in Eclipse now. You'll see that this class is used to store a compiled Java class by recording its name in the field `name` and the actual bytecode as a sequence of bytes in the field `classData`. In addition, notice that this class can be serialised as it extends the `Message` class and therefore implements the `Serialization` interface. The second class is `DynamicObjectInputStream` which extends `ObjectInputStream` and supports an additional public method called `addClass` which takes two arguments, the name of the class as a `String` and the class data as an array of bytes.

This runtime extension of supported message types can be achieved by replacing `ObjectInputStream` with `DynamicObjectInputStream`. Serialised instances of classes known to the client (e.g. `RelayMessage`) continue to work as before. Whenever messages of type `NewMessageType` are received your implementation of `ChatClient` should call `addClass` with the name and bytes of the new class received on the input stream. The class `DynamicObjectInputStream` handles all the complexities of making the new class available to the JVM class loader. Once the new class is available via the class loader, instances of the new message type can be serialised by the server and sent to the client, safe in the knowledge that the client will know how to deserialise it. Note that ordering is important here: the server must send the class definition *before* any serialised instances of the class so that the serialised objects can be correctly deserialised.

Using a class loader

10. Replace the use of `ObjectInputStream` with `DynamicObjectInputStream` in your implementation of `ChatClient`.

11. Insert additional code to `ChatClient` to handle receiving messages of type `NewMessageType` and call `addClass` on your instance of `DynamicObjectInputStream` whenever you receive a message of this type. Print out the following message:

```
14:54:27 [Client] New class <name> loaded.
```

where `<name>` is the name of the class you have received.

12. Modify your implementation of `ChatClient` so that if you receive an object that is not a `RelayMessage`, a `StatusMessage` or a `NewMessageType`, your program will print:

```
14:54:27 [Client] New message of unknown type received.
```

13. Test your new version of `ChatClient` with `java-lb.cl.cam.ac.uk`, port 15004.

Your new version of `ChatClient` should now print out the names of several classes which were sent by the server when the client connected, together with periodic "unknown type" messages. If it does not, ask for help from a demonstrator.

Your new addition to `ChatClient` doesn't appear to do very much at the moment. You can detect that new classes are being loaded and new types of messages are arriving, but you can't do much with them. The dynamic installation of a new class definition at runtime means that you cannot, at compile time, know the names of the methods and fields of the class, since you don't know what they will be. (They could well have been written after you started running your copy of `ChatClient`!) Thankfully Java supports *reflection* which permits the inspection of the contents of any Java object or class at runtime. Reflection can be used to determine the names and contents of fields, the names and argument types of methods, as well as being able to instantiate instances of a class and invoke methods on objects. Reflection is commonly used when writing an IDE such as Eclipse, since Eclipse needs to inspect the type information of the code entered by the programmer to provide assistance or additional documentation; reflection is also useful when writing testing frameworks or debuggers, for example the unit testing framework associated with this course makes use of reflection extensively to inspect the code you write.

With the exception of the eight primitive types you learnt about last year (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`) all variables and fields in Java reference objects inherited from `java.lang.Object`. All objects have a class definition, and the JVM includes a read-only representation of this definition to accompany the object. You can retrieve a copy of a definition of the class by appending `.class` onto a type. For example, to get a reference to the class definition of the `String` class you can do the following:

```
Class<String> stringClass = String.class;
```

You can also get a reference to the class definition from any Java object by calling the method `getClass`. For example, given an instance of a `String` object, you can get a reference to the `String` class as follows:

```
Class<String> stringClass = "Computer Laboratory".getClass();
```

It's possible that you won't know the type of the class at compile time, and if so you cannot set the appropriate generic type when referencing a class, in which case you can use the question mark notation:

```
Class<?> someClass = object.getClass();
```

Instances of type `Class` are just Java objects themselves. Therefore you can call methods on them as you do on other Java objects. These methods allow you to inspect the contents of the class. For

example, the method `getDeclaredFields` will return a list of fields found in the Java class. Similarly, `getMethod` will search for a method by name and return a reference to it, if it exists. The Java documentation for `java.lang.Class` contains information on how to use these and other methods.

Reflection

14. Remove the print statement you placed in the code in response to question 12.

15. Modify your implementation of `ChatClient` to extract all the declared fields from any unknown new message type sent by the server and print out their contents. For example, if you receive an unknown message with the name `NewMessageClass` with two fields, `field1` (with the value `hello`) and `field2` (with the value `world`), your program should print out

```
14:55:27 [Client] NewMessageClass: field1(hello), field2(world)
```

16. Modify your implementation of `ChatClient` to check for the presence of a method called `run` which takes no arguments. If you find such a method, invoke the method on the object you've received. (In this case you are dynamically executing new code you've downloaded from the `Socket` object!) Please make sure you print out the contents of any fields (question 15 above) *before* you invoke the `run` method (if any).

17. Read the following articles to cement your knowledge of class loading and reflection:

- <http://www.ibm.com/developerworks/java/library/j-dyn0429/>
- <http://www.ibm.com/developerworks/java/library/j-dyn0603/>

Your Assessor will ask you questions based on the contents of these articles next week.

Important

Further information on reflection is also available at:

<http://java.sun.com/docs/books/tutorial/reflect/>

Ticklet 2

You have now completed all the necessary code to gain your second ticklet. Please generate a jar file which contains all the code you have written for package `uk.ac.cam.crsid.fjava.tick2` together with the code you downloaded and imported in package `uk.ac.cam.cl.fjava.messages`. Please use Eclipse to export both the class files *and the source files* into a jar file called `crsid-tick2.jar`. Once you have generated your jar file, check that it contains at least the following classes:

```
crsid@machine~:> jar tf crsid-tick2.jar
META-INF/MANIFEST.MF
uk/ac/cam/crsid/fjava/tick2/TestMessageReadWrite.class
uk/ac/cam/crsid/fjava/tick2/TestMessageReadWrite.java
uk/ac/cam/crsid/fjava/tick2/ChatClient.class
uk/ac/cam/crsid/fjava/tick2/ChatClient.java
uk/ac/cam/crsid/fjava/tick2/TestMessage.class
uk/ac/cam/crsid/fjava/tick2/TestMessage.java
uk/ac/cam/cl/fjava/messages/DynamicObjectInputStream$.class
uk/ac/cam/cl/fjava/messages/DynamicObjectInputStream.class
uk/ac/cam/cl/fjava/messages/DynamicObjectInputStream.java
uk/ac/cam/cl/fjava/messages/ChangeNickMessage.class
uk/ac/cam/cl/fjava/messages/ChangeNickMessage.java
uk/ac/cam/cl/fjava/messages/NewMessageType.class
uk/ac/cam/cl/fjava/messages/NewMessageType.java
uk/ac/cam/cl/fjava/messages/RelayMessage.class
uk/ac/cam/cl/fjava/messages/RelayMessage.java
uk/ac/cam/cl/fjava/messages/StatusMessage.class
uk/ac/cam/cl/fjava/messages/StatusMessage.java
uk/ac/cam/cl/fjava/messages/ChatMessage.class
uk/ac/cam/cl/fjava/messages/ChatMessage.java
uk/ac/cam/cl/fjava/messages/Message.class
uk/ac/cam/cl/fjava/messages/Message.java
crsid@machine~:>
```

When you are satisfied you have built the jar correctly, you should submit your jar file as an email attachment to `ticks1b-java@cl.cam.ac.uk`.

You should receive an email in response to your submission. The contents of the email will contain the output from a program (written in Java!) which checks whether your jar file contains all the relevant files, and whether your program has run successfully or not. If your jar file does not pass the automated checks, then the response email will tell you what has gone wrong; in this case you should correct any errors in your work and resubmit your jar file. You can resubmit as many times as you like and there is no penalty for re-submission. If, after waiting one hour, you have not received any response you should notify `ticks1b-admin@cl.cam.ac.uk` of the problem. You should submit a jar file which successfully passes the automated checks by the deadline, so don't leave it to the last minute!
