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:

https://www.cl.cam.ac.uk/teaching/current/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 spelling[1]). 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 then 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

---

[1]The 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.wikipedi-a.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 your development environment may encourage 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 Specification[2] 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!

_____

[2]http://download.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html

---

Serialising a simple class

Here is a simple class which implements the serialisation interface:

```
package uk.ac.cam.your-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. Start the ticklet2 exercise on Chime: https://www.cl.cam.ac.uk/teaching/current/FJava/ticklet2.

2. Clone a copy of your repository from Chime onto your local machine. You should see an implementation of `uk.ac.cam.your-crsid.fjava.tick2.TestMessage` in the repository.

3. Complete the `TODO`s in
   `uk.ac.cam.your-crsid.fjava.tick2.TestMessageReadWrite`.

4. 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, replacing "crsid" with your CRSID: https://www.cl.cam.ac.uk/teaching/current/FJava/testmessage/crsid.jobj

---

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
ChangeNickMessage	Client→Server	Update nickname of the client stored by the server.
ChatMessage	Client→Server	Message written by a user is sent to the server.

---

[3]This 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/rfcs/) are a good place to start.

Message type (class)	Direction	Description
RelayMessage	Server→Client	User message sent from server to all clients.
StatusMessage	Server→Client	Message generated by the server, sent to all clients.

Table 1. Message classes sent between the client and server

Serialisation

5. Read through the classes which define the message types by inspecting the classes inside the package `uk.ac.cam.cl.fjava.messages`. Since these messages are used to support communication between the server and the client you should not modify the contents of this package.

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".

```
14:23:27 [Client] Connected to java-1b.cl.cam.ac.uk on port 15003.
14:23:27 [Server] Anonymous15983 connected from evapod.discoveryone.space.
\nick Dave
14:23:29 [Server] Anonymous15983 is now known as Dave.
14:23:14 [Server] Anonymous82791 connected from cpu9000.discoveryone.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.
```

Figure 1. A chat session between Dave and Hal[4]

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

---

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

stating that a new user (currently called `Anonymous15983`) has connected from the machine `evap-od.discoveryone.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 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

6. 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.

7. Create the class `ChatClient` inside the package `uk.ac.cam.your-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.

8. 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 your repository 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 `byte`s.

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

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

10. 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.

11. 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.
    ```

12. Test your new version of `ChatClient` with `java-1b.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.

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 IntelliJ, since IntelliJ 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 `get-Class`. 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**
>
> 13 Remove the print statement you placed in the code in response to question 12.
>
> 14 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)
> ```

Annotations

More recent releases of Java (since 1.5) support *annotations*. Annotations provide information about programs, often called *meta-data*, and are commonly used to (1) provide information to the compiler, (2) control compilation or deployment, or (3) aid program execution. The definition or use of an annotation in Java is always prefixed with the at symbol (`@`).

First, a simple example. Many software development companies have certain conventions on style and quality. A common requirement is to record pertinent information concerning the authors and changes made to a particular piece of code. This is traditionally done by writing a comment at the start of the file. Java annotations provide an opportunity to structure this meta-data and therefore provide better control of presentation in documentation or even make use of this information when compiling or running software. For example we might like to annotate classes for the Further Java practical classes as follows:

```
@FurtherJavaPreamble(
author = "Maurice V. Wilkes",
date = "4th May 1949",
crsid = "mvw1",
summary = "Calculate the table of squares between 1 and 99",
ticker = FurtherJavaPreamble.Ticker.A)
class TableOfSquares {
...
```

Annotations can be added to programming elements such as classes, methods and fields by including the annotation directly above the element. Annotations are typically made up of zero or more name-value pairs. In the above example, `Maurice V. Wilkes` is the value associated with the name `author`. Before you can use an annotation, you must define it. In Java you define an annotation using the keyword

@interface. Here is an example definition for the preamble used above which would be placed in a file called `FurtherJavaPreamble.java`:

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

public @interface FurtherJavaPreamble {
 enum Ticker {A, B, C, D};
 String author();
 String date();
 String crsid();
 String summary();
 Ticker ticker();
}
```

In addition to user-defined annotations, Java supports three built-in annotations:

`@Deprecated`	Indicates that the marked class, method or field is deprecated and should not be used. The compiler will then generate a warning if a program is compiled which makes use of a deprecated element.
`@Override`	Indicates that the marked class, method or field is supposed to override an element in a superclass. The compiler will generate an error if the method does not override a method in the superclass.
`@SuppressWarnings`	Indicates that the compiler should suppress a warning which it would otherwise generate. For example, `@SuppressWarnings("deprecation")` can be used to suppress a compiler warning generated by the use of a method which is marked by the deprecated annotation described above; another common example is `SuppressWarnings("unchecked")` which removes the warnings which result when interfacing with code which was written before the introduction of Java generics.

Annotations can even be applied to annotations. For example, if we wish to access an annotation via reflection at runtime, the annotation `@Retention(RetentionPolicy.RUNTIME)` must be added above the definition of the annotation. You can see an example of this in `uk.ac.cam.cl.fjava.messages.Execute` in your respository.

Reflection and annotations

15 Add the definition of the annotation `FurtherJavaPreamble` as shown above to the correct package.

16 Use the annotation `FurtherJavaPreamble` by annotating your `ChatClient` class.

17 Annotate the `FurtherJavaPreamble` annotation so that its values are accessible to the Further Java testing engine when it tests your code.

18 Modify your implementation of `ChatClient` to check for the presence of declared methods from any unknown new message type. If such a method takes no arguments and is annotated with the `Execute` annotation (as defined in the code repository), 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 any methods.

Ticklet 2

Important

Read the following articles to cement your knowledge of class loading, reflection and annotations:

- http://www.ibm.com/developerworks/java/library/j-dyn0429/
- http://www.ibm.com/developerworks/java/library/j-dyn0603/
- http://download.oracle.com/javase/tutorial/reflect/
- http://docs.oracle.com/javase/tutorial/java/annotations/

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

You have now completed all the necessary code to gain your ticklet. Your repository should contain the following source files:

```
src/uk/ac/cam/your-crsid/fjava/tick2/TestMessageReadWrite.java
src/uk/ac/cam/your-crsid/fjava/tick2/ChatClient.java
src/uk/ac/cam/your-crsid/fjava/tick2/TestMessage.java
src/uk/ac/cam/your-crsid/fjava/tick2/FurtherJavaPreamble.java
src/uk/ac/cam/cl/fjava/messages/Execute.java
src/uk/ac/cam/cl/fjava/messages/DynamicObjectInputStream.java
src/uk/ac/cam/cl/fjava/messages/ChangeNickMessage.java
src/uk/ac/cam/cl/fjava/messages/NewMessageType.java
src/uk/ac/cam/cl/fjava/messages/RelayMessage.java
src/uk/ac/cam/cl/fjava/messages/StatusMessage.java
src/uk/ac/cam/cl/fjava/messages/ChatMessage.java
src/uk/ac/cam/cl/fjava/messages/Message.java
```

When you are satisfied you have completed everything, you should commit all outstanding changes and push these to the Chime server. On the Chime server, check that the latest version of your files are in the repository, and once you are happy schedule your code for testing. 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 a final response you should notify `ticks1b-admin@cl.cam.ac.uk` of the problem. You should submit a version of your code which successfully passes the automated checks by the deadline, so don't leave it to the last minute!