SCHOOL OF COMPUTER SCIENCE

PRIFYSGOL BANGOR UNIVERSITY

Java Technologies Exercises - Notes

Creating and Using JAR Files

Dave Perkins

Introduction

In this document we investigate the creation and use of JAR files and some JAR related APIs. During the course of this semester you will undoubtedly have seen this technology mentioned in the course of your reading: but what exactly is a JAR file? Oracle provides this definition:

> JAR (Java Archive) is a platform-independent file format that aggregates many files into one. Multiple Java applets and their requisite components (.class files, images and sounds) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction, greatly improving the download speed. The JAR format also supports compression, which reduces the file size, further improving the download time. In addition, the applet author can digitally sign individual entries in a JAR file to authenticate their origin. It is fully extensible.[1]

Well, that may be a start but obviously we are going to need more information if anything useful is to be done with this file format. For the purposes of these laboratory exercises a good place to start is by visiting the Oracle Tutorial site below

http://docs.oracle.com/javase/tutorial/deployment/jar/index.html

Another useful site is provided by Nanyang Technological University, Singapore

https://www3.ntu.edu.sg/home/ehchua/programming/java/J9d_Jar.htm

Having looked at this material you should be in a position to start and complete the first few exercises in these notes; they are all relatively straightforward. Some of the later exercises, however, will require more effort and perhaps some independent research.

Some other useful sources of information are listed below:

- Horstmann's Appendix G Tool Summary in *Big Java*
- The Oracle site has a page describing in detail the use of the Java Archive Tool
  http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jar.html

In the following exercises you will have to use the **jar** tool with appropriate options and via the command line to create and manage JAR files. This can be quite tedious and it is not my mention to recommend this way of working. Instead the purpose of the following exercises is to help you understand some of the magic which is going on when an IDE such NetBeans handles the management of JAR files for you.

[1] http://docs.oracle.com/javase/6/docs/technotes/guides/jar/index.html

Exercise 1: Creating a JAR File

In this first exercise you are asked to construct a JAR file using the Windows command line. When this file is executed you should get the following:

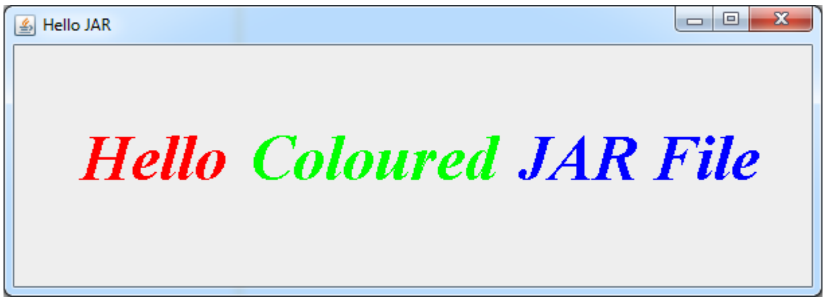


Figure 1 Running a Hello JAR

The JAR file should be built using the class files listed below:

- **HelloComponent.class** **(See Lab 1 ICP 2151 for a similar program)**
- **HelloViewer.class**

After the JAR has been created all of the files below should be visible:

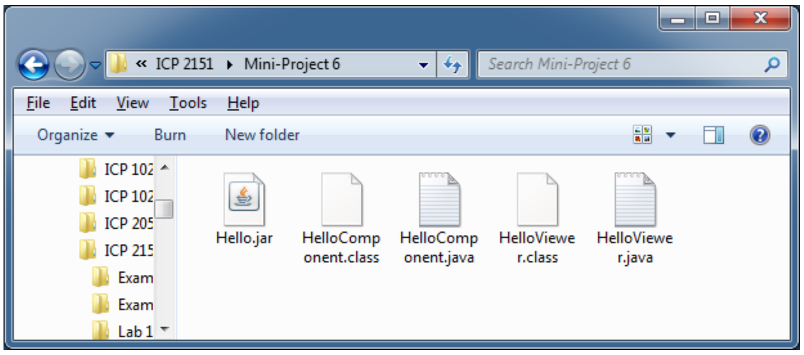


Figure 2 JAR File Created

In order to execute a JAR file an *entry point* must be defined in the associated manifest file. If you do not know what this means look in the documentation listed on the previous page. The entry point can be defined manually, but for the purposes of this exercise you are required to use the **-e** option when creating the archive file.

During the laboratory demonstration you will be required to issue a *single* command to create a JAR file with the correct entry point. Note that you may be asked to create a JAR file *without* compression.

You are advised to refer to the site below for further information about command options.

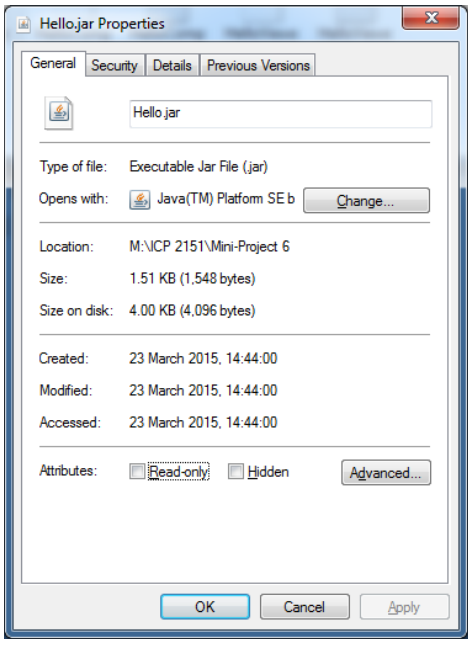
http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jar.html

Exercise 2: Executing JAR Files

To complete this exercise you must do two things:

- demonstrate that the you know how to execute the JAR files from the command line;
- demonstrate that you know how to execute the JAR file using the mouse pointer.

Note that to complete the second activity above you must configure Windows so that the JAR will execute *automatically* if you double click the mouse whilst the pointer is on the JAR icon. To do this you will need to associate the icon with a specific program, namely the Java Platform ™ SE Binary. If done correctly the Properties window for the JAR file should look like this:


Hello.jar Properties

General | Security | Details | Previous Versions

Hello.jar

Type of file: Executable Jar File (.jar)
Opens with: Java(TM) Platform SE b Change...

Location: M:\ICP 2151\Mini-Project 6
Size: 1.51 KB (1,548 bytes)
Size on disk: 4.00 KB (4,096 bytes)

Created: 23 March 2015, 14:44:00
Modified: 23 March 2015, 14:44:00
Accessed: 23 March 2015, 14:44:00

Attributes: Read-only Hidden Advanced...

OK Cancel Apply

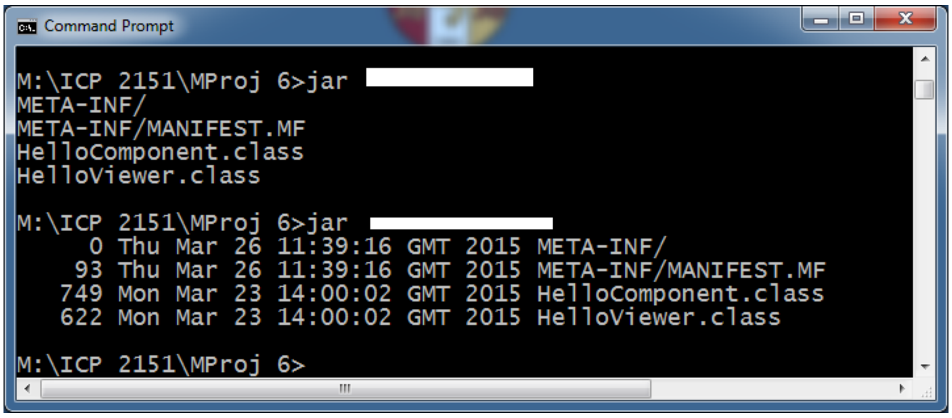
Figure 3 Hello.jar Properties

Having got your JAR file to execute you may wonder if it is possible to turn the file into a Windows executable, that is a Windows program with the standard **.exe** extension. The answer to this question is in the affirmative and involves the use of a program known as a *Windows wrapper utility*. Two reliable open source products are *JSmooth* and *Launch4J* both of which provide wrapper services. Use a Windows wrapper to convert your JAR to a Windows **.exe**. Test that the resulting executable can be started using either the command line or the mouse pointer.

Note that you are free to use any suitable equivalent for other platforms (e.g. Macs) but you must be able to demonstrate understanding of what it is that you are doing.

Exercise 3: Listing and Extracting JAR Files

Use the **jar** command with appropriate *command options* so as to generate the two listings below:



```
M:\ICP 2151\MProj 6>jar [redacted]
META-INF/
META-INF/MANIFEST.MF
HelloComponent.class
HelloViewer.class

M:\ICP 2151\MProj 6>jar [redacted]
     0 Thu Mar 26 11:39:16 GMT 2015 META-INF/
    93 Thu Mar 26 11:39:16 GMT 2015 META-INF/MANIFEST.MF
   749 Mon Mar 23 14:00:02 GMT 2015 HelloComponent.class
   622 Mon Mar 23 14:00:02 GMT 2015 HelloViewer.class

M:\ICP 2151\MProj 6>
```

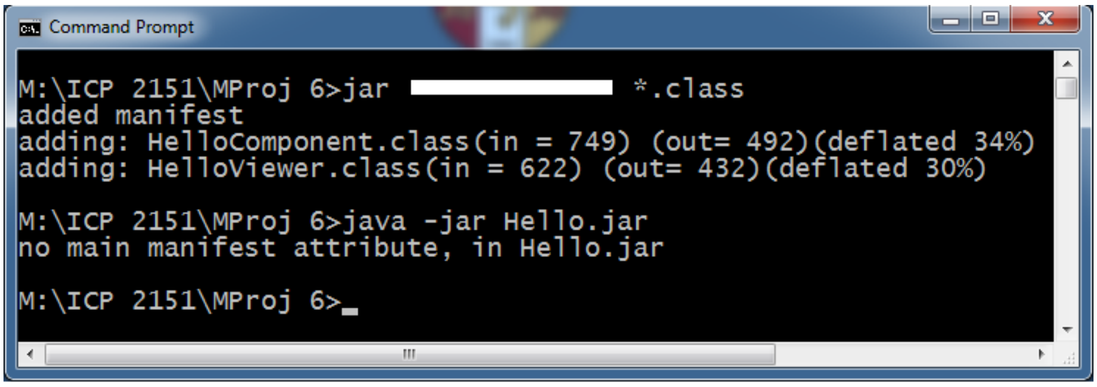
Figure 4 Listing Contents of a JAR File

Create a directory called **Xtract1** in the root of **M:** drive. Then use an apppropriate option with the **jar** tool to *extract* all of the files and directories from **Hello.jar** and place them inside **Xtract1**.

Repeat this exercsie but use a file archiver utility such as 7-Zip to extract the contents of **Hello.jar** and place them in a directory called **Xtract2**.

Exercise 4: Working with Manifest Files

To begin this exercise you should delete the file **Hello.jar**. You should then rebuild the JAR file but do *not* use the **-e** option instead you should generate a JAR file that cannot be immediately executed. Notice below that the command option used does generate a manifest and archives the appropriate class files but the JAR cannot be executed.

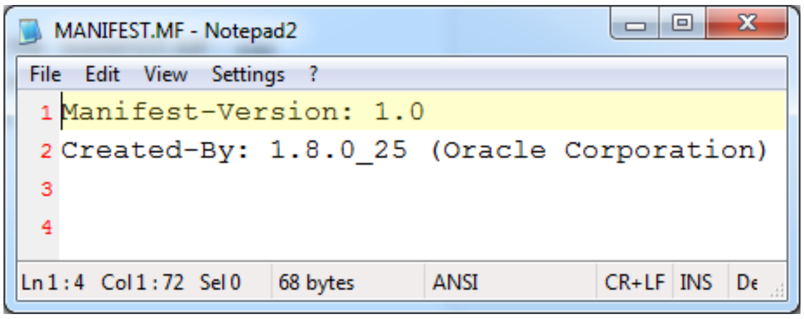


```
M:\ICP 2151\MProj 6>jar [redacted] *.class
added manifest
adding: HelloComponent.class(in = 749) (out= 492)(deflated 34%)
adding: HelloViewer.class(in = 622) (out= 432)(deflated 30%)

M:\ICP 2151\MProj 6>java -jar Hello.jar
no main manifest attribute, in Hello.jar

M:\ICP 2151\MProj 6>_
```

Figure 5 Missing Main Manifest Attribute

The reason is that when the JAR file was created a *default* manifest file - **MANIFEST.MF** - was generated which does not specify the main manifest attribute. To confirm that this is the case extract the default manifest file and inspect its contents using a text editor.

As you can see this manifest file contains version and creation 'header:value' pairs but no such pair to indicate the entry point.



```
Manifest-Version: 1.0
Created-By: 1.8.0_25 (Oracle Corporation)
```

Figure 6 Contents of Default Manifest

To solve this problem you should use the **jar** tool with appropriate options to *update* the manifest of the JAR file **Hello.jar**. Note that the *update* option uses a text file containing the following line:

```
Main-Class: HelloViewer
```

The text file should be called **manifest-additions.mf**. Once the manifest has been updated the JAR file should be executable. Check that this is the case.

Finally delete **Hello.jar** and then see if you can create an *executable* JAR using the following command:

```
jar cfm Hello.jar . . .
```

To get this to work you will also need to create a manifest file containg three 'header:value' pairs.

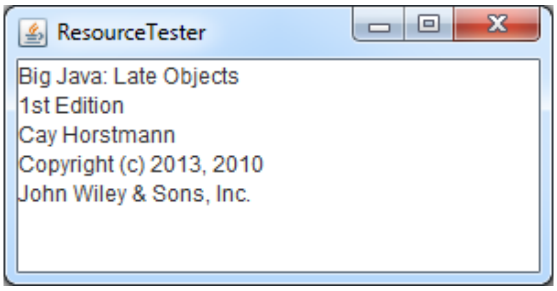
```
Manifest-Version:
Created-By:
Main-Class:
```

Exercise 5: Providing Resources

Java applications distributed as JARs will often need to access associated data files such as:

- Text files
- Image and audio files
- Binary files

In this exercise you will explore how a *resource* such as a text file can be associated with an application that has been bundled as a JAR. The example used is concerned with the display of information relating to books stored in a library, more particularly displaying messages about the various editions of *Big Java*. The idea is illustrated below:



```
Big Java: Late Objects
1st Edition
Cay Horstmann
Copyright (c) 2013, 2010
John Wiley & Sons, Inc.
```

Figure 7 Displaying a Text File Resource from a JAR

Over time the books in the library will change and so the information held about the titles will also need to change. To make this as easy as possible we avoid hardcoding title data into an application but instead store it in a text file called **Big_Java_Late_Objects_1.txt**. Where should this file be stored? Well, the best place to put it would be the other program files stored inside a JAR. To better understand this technique work your way througgh the following activities:

- create a text file to hold the above details about the *Big Java* text;
- create a class called ResourceTester (see Appendix 1);
- create a directory called **data** in the same place as the text file and the Java source file;
- create a JAR file which contains both the text and Java source file;
- execute the JAR file.

Exercise 6: JAR Files and Maven

Create a Maven project containing an application to generate the standard *Hello World* message.

```
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

The Project structure should resemble following:

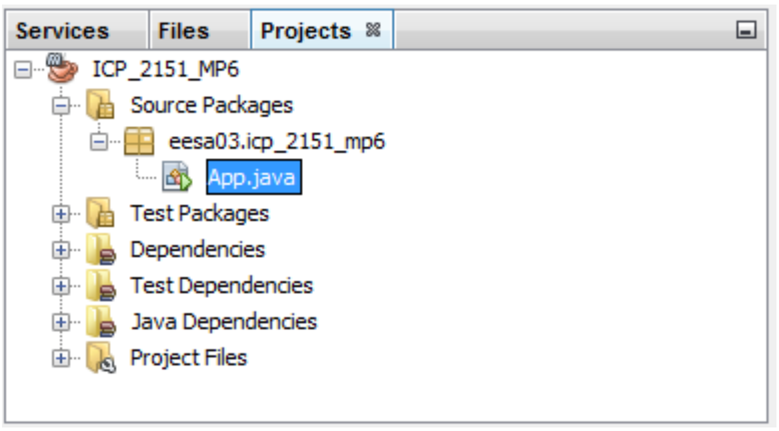


Figure 8 Maven Project Structure

Having created the application run it from within Maven and test that the appropriate message is generated inside the Output Window. If succesful use Maven to create a JAR containg this application.

The resulting file structure should look like this:

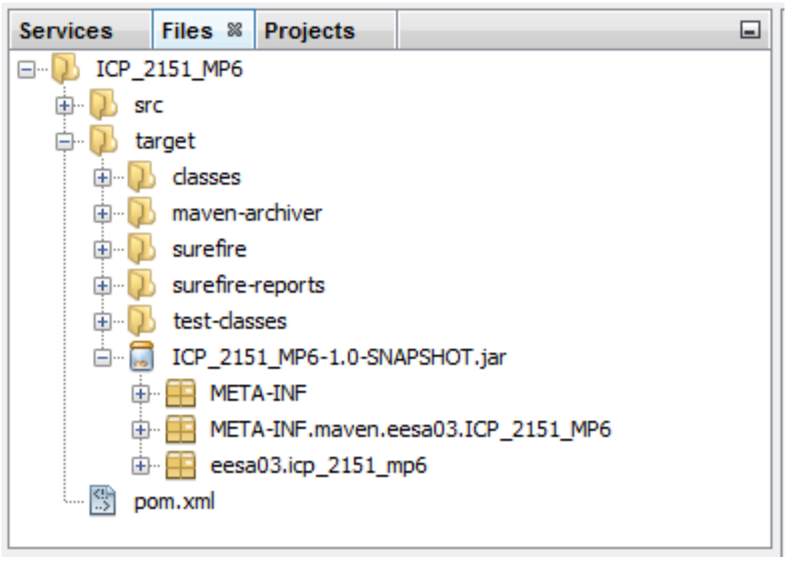


Figure 9 Folder Containing SNAPSHOT JAR File

Copy this JAR file to your M: drive and try to run it. You will find that the JAR cannot be executed. To resolve this problem use the Maven Assembly Plugin. For details use the link below:

http://maven.apache.org/plugins/maven-assembly-plugin/usage.html

Be prepared to demonstrate that the Maven generated JAR can be executed from either the command line or by using the mouse pointer.

Exercise 7: JARs of Beans

This exercise introduces the topic of Java Beans. As this term may be unfamiliar to some of you we shall begin with the official definition of the concept taken from the JavaBeans specification document:

> *A bean is a reusable software component based on Sun's JavaBeans specification that can be manipulated visually in a builder tool.*[2]

As you see the above definition makes mention of a builder tool, a piece of software which enables the direct manipulation (i.e. drag and drop, use of drop down menus) of software components including the familiar widget toolkit used to build GUIs. In this exercise you will the Matisse builder tool supplied with the NetBeans IDE.

An interesting example of a bean with rich behaviour is the **`CalendarBean`** written by Kai Tödter. This can be freely downloaded from

http://www.toedter.com/en/jcalendar

In this exercise you are required to first download the bean and place it within a **`JFrame`** so that the calendar can be displayed. When you run your program you should get something like this:

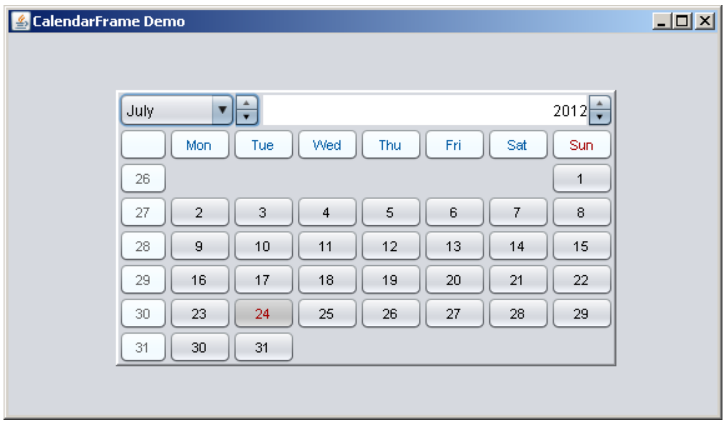


Figure 8 Using CalendarBean

To add a bean you must first place it in the palette of the builder tool. To do this use

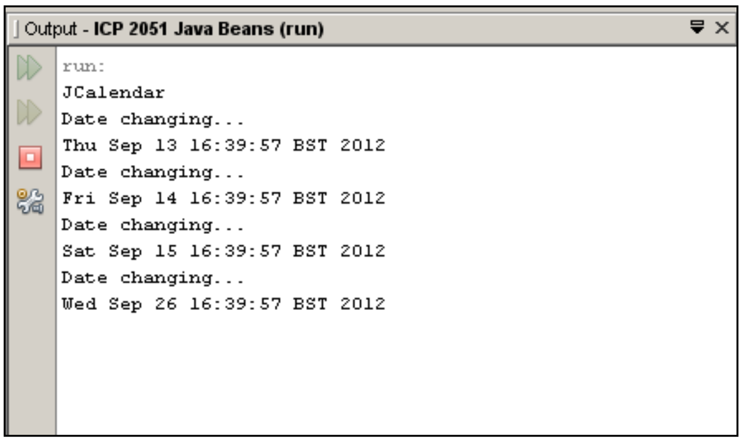
- Tools-> Palette -> Swing\AWT Components
- Add the bean (or beans) from the **JAR** file
- Place the bean(s) into the Beans Palette

If everything has been done correctly you shoud be able to simply drag your bean onto the **JFrame**.

Remember the essential thing that you must do in this exercise is to locate the file **jcalendar-1.4.jar** and use the **Palette** option in **Tools** to place the various beans in this **JAR** file in the Beans collection.

[2] JavaBeans API 1.01 Specification, p.9. To obtain this specification visit the site below maintained as part of the Java Community Process http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/

Once the **`CalendarBean`** is up and working use the **Design** view to modify the source code so that whenever the user changes the date on the calendar tool the date is displayed as a string in the output window of NetBeans. Your output should resemble the following:



```
run:
JCalendar
Date changing...
Thu Sep 13 16:39:57 BST 2012
Date changing...
Fri Sep 14 16:39:57 BST 2012
Date changing...
Sat Sep 15 16:39:57 BST 2012
Date changing...
Wed Sep 26 16:39:57 BST 2012
```

Figure 9 Recovering Date Information from CalendarBean

Hint. Use a method belonging to the **`JCalendar`** class to extract date information.

Exercise 8: Creating and Distributing Packages

In this final exercise you will learn how to create a Java package and distribute it as a JAR file. As you already know a Java package is a convenient mechanism for bundling software components together. This idea is reflected in the official definition of the term package

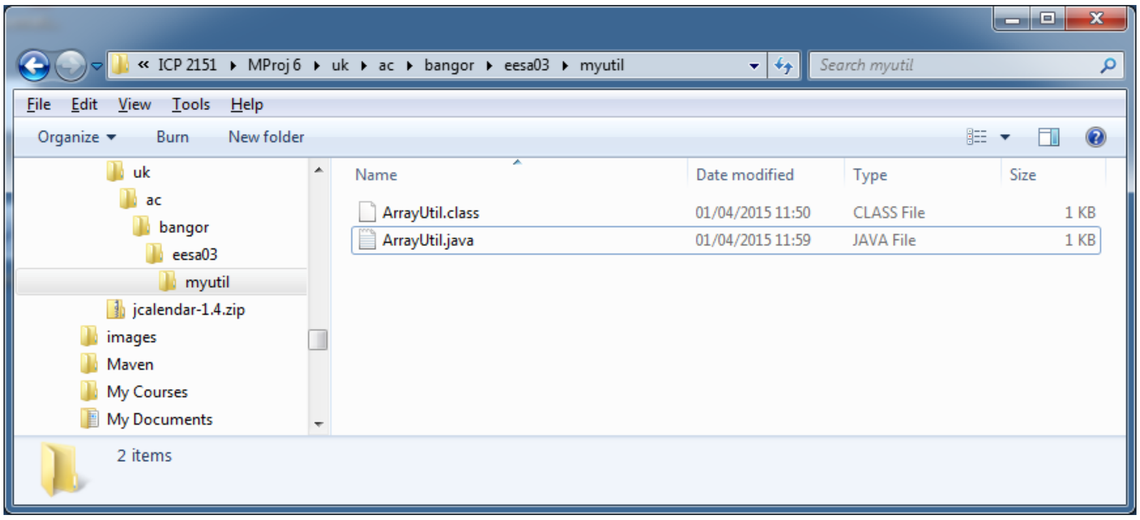
> A *package* is a grouping of related types providing access protection and name space management. Note that *types* refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so *types* are often referred to in this lesson simply as *classes and interfaces*.[3]

Note that In this exercise the package you create will contain only classes but the exercise could be repeated for packages containg the other types listed in the above definition.

To get started download the file **`ArrayUtil.java`** which is supplied through Blackboard. Your first task will be to create a package called **`myutil`** which contains the compiled source code for the class **`ArrayUtil`**. Before doing this you should read *Big Java*, Section 12.4 on how to name a package. As you will see the important thing is to come up with a name that has a high probability of being unique. Commercial developers frequently follow the convention of reversing their *registered domain name* and prefixing that name to the package they are producing. For example, Horstmann would use the name **`com.horstmann.myutil`** to complete this exercise. This is all very well if you are the lucky owner of a domain name, but if not what should you do? One approach is to use your email-address backwards. So, for instance, my email address is **`eesa03@bangor.ac.uk`** and

[3] See the Java™ Tutorials. https://docs.oracle.com/javase/tutorial/java/package/packages.html

this can be used to create the package name **`uk.ac.bangor.eesa03.myutil`**. The use of this name will involve creating the directory structure illustrated below.


« ICP 2151 ▸ MProj 6 ▸ uk ▸ ac ▸ bangor ▸ eesa03 ▸ myutil
Search myutil
File Edit View Tools Help
Organize ▾ Burn New folder
uk
ac
bangor
eesa03
myutil
jcalendar-1.4.zip
images
Maven
My Courses
My Documents
2 items

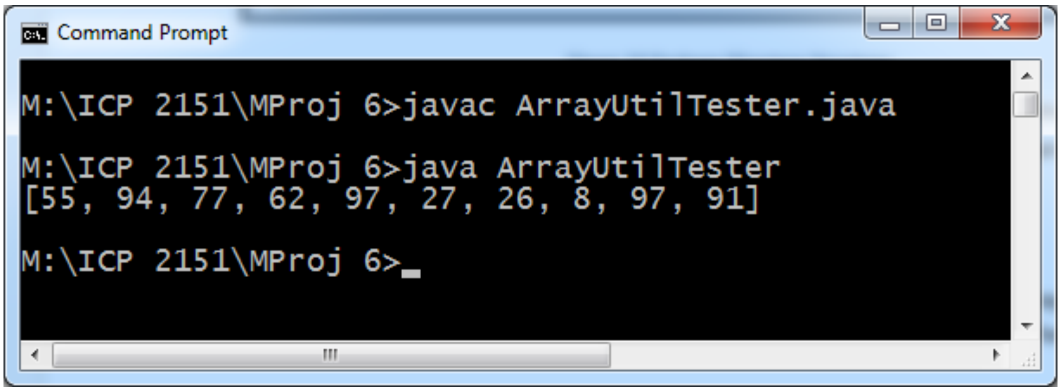
Name	Date modified	Type	Size
ArrayUtil.class	01/04/2015 11:50	CLASS File	1 KB
ArrayUtil.java	01/04/2015 11:59	JAVA File	1 KB

Figure 10 Package Directory Structure

Having created the package write a program to test whether the array utility method **`randomIntArray`** can be successfully accessed. Your test program must contain an import statement similar to the one below:

```
import uk.ac.bangor.eesa03.myutil;
```

To test your package I suggest that the test program should create an integer array of length 10 to hold random values between 0 and 99 inclusive. The array should then be printed to the console screen.



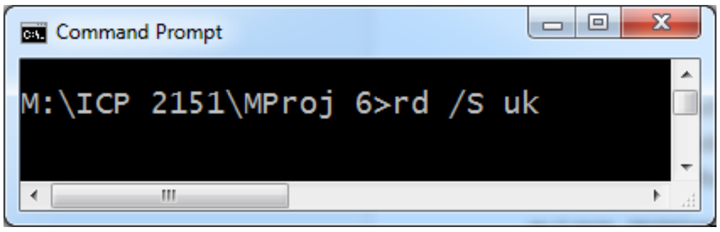
```
M:\ICP 2151\MProj 6>javac ArrayUtilTester.java

M:\ICP 2151\MProj 6>java ArrayUtilTester
[55, 94, 77, 62, 97, 27, 26, 8, 97, 91]

M:\ICP 2151\MProj 6>_
```

Figure 11 Testing Package myutil

Once you are confident that your array utility class is functioning you should use the jar tool to place your package inside a JAR. The JAR file you create must maintain the directory structure of the package. Check that you get something like the following:

```
M:\ICP 2151\MProj 6\MyUtil.jar\uk\ac\bangor\eesa03\myutil\ArrayUtil.class
```

You should now create a “safe-keeping” directory called **`mylib`** in the root directory of the **M:** drive and copy the JAR file over to this new directory. Also remove any other copy of **MyUtil.jar** and the directory structure for the package. On my computer I used the command below to delete the myutil package:

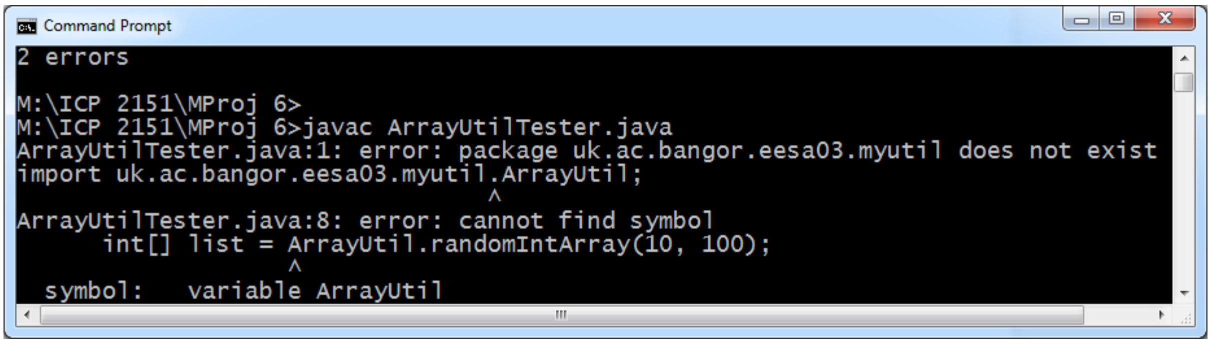


```
M:\ICP 2151\MProj 6>rd /S uk
```

Figure 12 Removing the Package Structure

Having done this bit of housekeeping you are now in the position of having a JAR file stored in a local directory. Our problem is to learn how to use the components bundled in the JAR to develop an application. This is exactly the situation programmers face when they download a third-party JAR to assist in the development of an application.

To explore this issue go back to the directory in which you have the source file for the test class **`ArrayUtilTester`**. If you try to compile this class you will get the following error:



```
2 errors

M:\ICP 2151\MProj 6>
M:\ICP 2151\MProj 6>javac ArrayUtilTester.java
ArrayUtilTester.java:1: error: package uk.ac.bangor.eesa03.myutil does not exist
import uk.ac.bangor.eesa03.myutil.ArrayUtil;
                                 ^
ArrayUtilTester.java:8: error: cannot find symbol
      int[] list = ArrayUtil.randomIntArray(10, 100);
                   ^
  symbol:   variable ArrayUtil
```

The problem, of course, is that the compiler can no longer find the package in which you have stored the class **`ArrayUtil`**. To provide this information you will need to make use of the classpath option (-cp) when using the command javac. A similar issue will arise when you try to execute the test program.

To complete this exercise you need to get **`ArrayUtilTester`** running and accessing the code stored in the JAR file which has been placed in **`mylib`** for safekeeping.