

# Lecture overview

- Integrated Development Environments (IDE's)
- Eclipse
- Using the keyword 'package' in Java
- Compiling and executing programs in packages
- Creating Java Archive files (JAR files)
- Executing a JAR file and editing the Manifest
- Making two or more classes work together in Eclipse

# Integrated Development Environment

- Programming in the large-scale is difficult
- In most instances teams of developers must work together to produce some product to specification
- Setting up environments for large-scale projects is also challenging

# IDE's

- In some software houses there are individuals employed specifically for deployment, i.e. deployment and setting up work environments can be a major challenge to any project
- IDE's aid in the development and deployment of software as the development is “integrated” into a common tool

# IDE's

- Creating huge software projects in Java requires the use of many classes and packages and APIs (Application Programming Interfaces)
- IDE's provide simple ways to import packages and API's and also provide simple mechanisms for building, testing and deploying software
- Eclipse is an example of a tool that can be used as an IDE...this lecture shows some of the 'made easy' activities carried out using eclipse!!!

# Java Packages

# Java ‘package’ keyword

- The keyword ‘package’ is used to define a group to which a particular class belongs
- In windows a package translates into a particular directory
- Packages are a good way of modularising code

# Java ‘package’ keyword

- The Java standard API is distributed into packages e.g. `java.lang`, `java.util`, `java.net` etc.
- Packages can be used to organise software into a hierarchical structure of classes, e.g. standard Java classes are all contained within the `java` package
- Packages can be used as a mechanism to avoid reuse of classes names also, e.g. `mypackage.MyClass` is distinguishable from `yourpackage.MyClass` even though the classes have the same names

# Java ‘package’ keyword

- Companies often use their Internet domain name in reverse order to start their package naming, this almost guarantees that naming conflicts will be avoided, e.g. `com.ourcorp.greatnewsoftwaresystem.gui`
- When you compile a program that has no package defined, the compiler will assign this program to the default package (usually the current directory ‘.’)



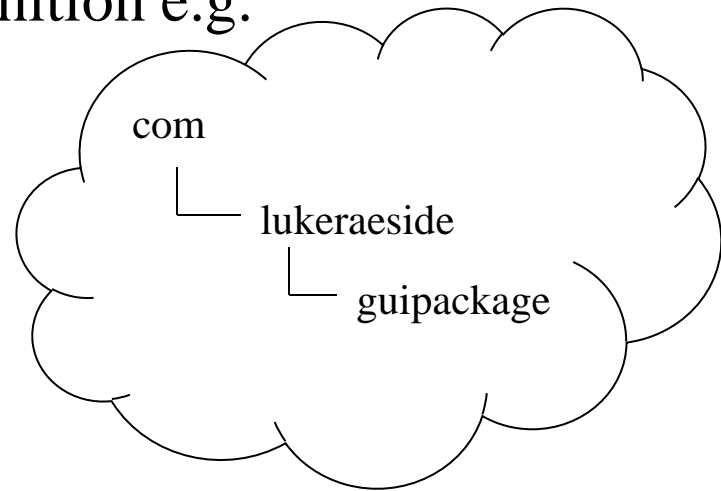
# Java 'package' keyword

- To define a package for a class place the package definition BEFORE the code definition e.g.

```
package com.lukeraeside.guipackage;
```

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

```
public void someMethod() {  
    System.out.println("You have called " + this.getClass().getName());  
}  
}
```



# Java ‘package’ keyword

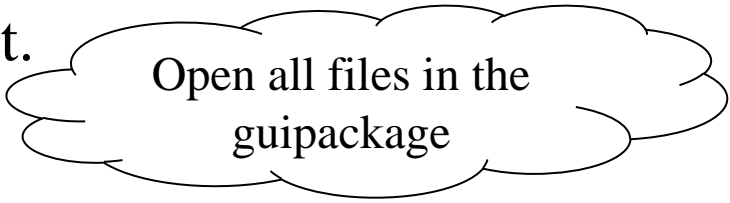
- The package statement must be the first statement after any comments in the class file, i.e. first code statement
- Access to the PackagedClass program from another class (not in the same package) can be achieved using two different techniques
  - Firstly using the full package name e.g.  
`com.lukeraeside.guipackage.PackagedClass.someMethod()`
  - Secondly by importing the `com.lukeraeside.guipackage` using the ‘import statement’ (to import all classes in the package use ‘\*’, use the class name `PackagedClass` for import of one class only)

# Compiling with packages

- When you compile packaged classes, in general, they are stored in corresponding directory structures
- All files in a package must be located in a subdirectory that matched the full package name, e.g. the `PackagedClass` used in the previous slide was stored using the `com/lukeraeside/guipackage` directory structure
- The point at which the hierarchy branches off depends on the classpath setting, i.e. if the classpath environment variable specifies the root directory then all package structures stem from this root path

# Compiling with packages

- In order for the below class to compile (when using Textpad), set the classpath environment variable to the root directory of the package hierarchy, and add '.' to the classpath if not already present.



Open all files in the  
guipackage

```
import com.lukeraeside.guipackage.*;  
public class Importer {  
    public static void main(String[] args) {  
  
        PackagedClass testClass = new PackagedClass();  
        testClass.someMethod();  
    }  
}
```

# Compiling with packages using Eclipse

- Eclipse automatically sets the classpath variables for you so you do not need to worry about editing classpaths manually
- When a project is created the project directory is the working directory, i.e. all files and folders will be referenced from the project directory by default

# The 'package' keyword as access modifier

- The keywords public, private and protected are used to specify access to features e.g.

```
public String username;  
private String password;
```

- The absence of a modifier will by default assign a feature to 'package' access e.g.

```
String username; //this attribute of a class is set to package access
```

- Package access means that other classes may access this feature directly if they are within the same package

# The 'package' keyword as access modifier

- If the PackagedClass class below were changed as below (someMethod() is no longer public), the Importer class would no longer compile!....why?

```
package com.lukeraeside.guipackage;
```

```
public class PackagedClass {
```

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

```
    void someMethod() {  
        System.out.println("You have called " + this.getClass().getName());  
    }  
}
```

# Java Archive Files (JAR Files)



# JAR Files

- JAR files allow you to load all of the classes needed to run a program into a single file (especially useful for multi-class programs)
- The JAR may also be compressed using ZIP format and therefore may help to increase the deployment and download efficiency of large Java programs
- Making JAR files is easy...simply use the **jar** command supplied with the jdk/jsdk...it is made even easier in Eclipse

# JAR Files

- In summary the advantages of JAR files include:
  - Security : JAR files can be signed
  - Decreased download time
  - Compressed – ZIP format
  - Portability – supported by standard Java

# JAR Files - Creating

- The **jar** command is located in the **bin** directory of the JDK
- The general format for use of the **jar** command is:

**jar options JARFileName File1 File2 ...**

- For example to jar all of the java files in the current directory use (c = create; f = command line):

**jar cf MyJar.jar \*.java**

# JAR Files

- JAR files may contain non-java files that are used by the program stored in the JAR
- These other files are termed ‘resources’ and can be accessed from the classes within the JAR file, e.g. there could be several images or sound files contained within the JAR file
- You may retrieve the resources stored in the JAR file from the program class by using the **getResource(name)** method as shown in the internationalisation lectures (later in the semester)!

# JAR Files - Viewing

- To view the contents of a JAR file use:

**jar tf filename**

- The *t* means view table, the *f* specifies the use a command line

# JAR Files - Extracting

- To extract a JAR file use the following command structure:

**jar xf filename**

- The x means extract, the f specifies the command line

(Note: adding the JDK bin directory to the PATH environment variable will allow the jar command be called from any directory)

# JAR Files – Manifest file

- When a JAR is created a manifest file is also stored in the JAR file
- Changes can be made to this manifest file so that extra information can be added
- To change the manifest create a text file that contains the extra information required
- One reason to change the manifest file is to specify the main class to execute when the program runs

# JAR Files – Manifest file

- To specify the main class for execution the following information must be added to the manifest:

**Main-Class: Name\_Of\_Main\_Class**

- To create a JAR file with the new manifest save the above information in a text file and execute the following command:

**jar cmf newManifestInfo.txt MyJar.jar \***



# JAR Files – Execution

- To run a JAR file with the main class specified in the manifest file execute the following command:

**java -jar Filename.jar**

# JAR Files – Using Eclipse

- Creating a JAR file in Eclipse is very straight forward
- Firstly select the Export menu option from the File menu
- Select Java->Jar File
- Select the directory to export the jar file to
- You can also specify the main class for execution of the Jar using the wizard

# Javadoc Commenting

# Javadoc commenting

- When new objects/classes are created in OOP they are usually intended for reuse
- They may be reused by the code implementer or they may be implemented by another programmer that had nothing to do with the coding of the object/class
- If you consider that a user of your class may not have the source code available to them, how can you communicate the functionality that your class provides??
- One way of explaining the functionality of your class is to use javadoc commenting

# Javadoc commenting

- When you visit [java.sun.com](http://java.sun.com) and examine a listing of java class methods using HTML, you are in fact looking at HTML information that has been automatically generated using javadoc comments
- The `/**` symbol is used to indicate the beginning of a javadoc comment
- The `*/` symbol indicates the end of the javadoc commenting
- Javadoc comment should precede the class, interface, method, and field definitions
- Javadoc comments include javadoc tag strings which label particular information into a particular category

# Javadoc commenting

- Below is an example of how to use the javadoc commenting:

```
/**
 * @param number The number to be square by the square method
 * @return The calling code is returned the square of number passed
 */
public double square(double number) {
    return (number * number);
}
```

- For a full listing of javadoc tags see  
<http://www.oracle.com/technetwork/java/index.html>

# Javadoc commenting

- To generate the javadoc HTML page from your code simply execute the javadoc command found in the bin directory of the jdk directory e.g.

```
javadoc MyClass.java
```

- If no destination folder is specified then the javadoc HTML page will be generated in the same directory as the source code

# Javadoc commenting using Eclipse

- Using Javadoc commenting with Eclipse is extremely simple, when you are creating your program simply select that you wish comments to be generated
- Eclipse will place commenting sections into your class and generated methods so that you can simply fill in the blanks with the appropriate information and tags
- Select the generate Javadoc command once your project is ready to be documented



# Using two classes together

- Generally speaking large software projects DO NOT simply contain one class or just one file
- It is expected that classes are split into logical units and these classes will work together to produce a desired effect
- The following example creates a simple utility class called MathHelper, another called my calculator then utilises this class by calling the *doubleInt* method in the MathHelper class

# Using two classes together

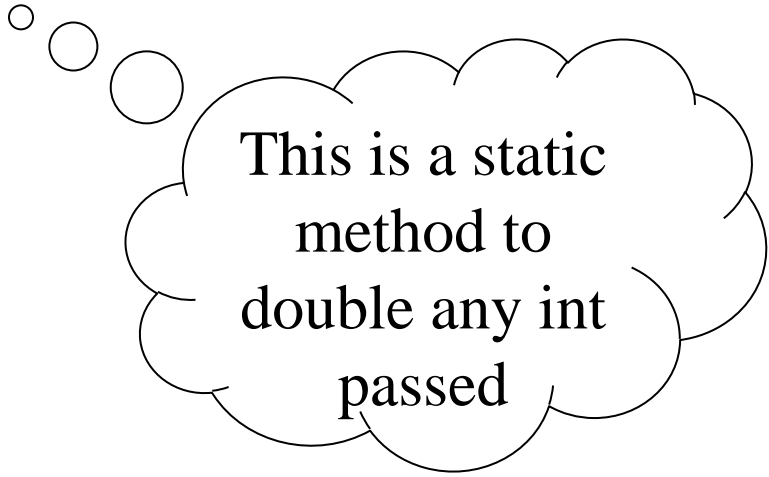
```
public class MathHelper {
```

```
    public static int doubleInt(int inputNum) {
```

```
        return inputNum*2;
```

```
    }
```

```
}
```



This is a static  
method to  
double any int  
passed

# Using two classes together Eclipse

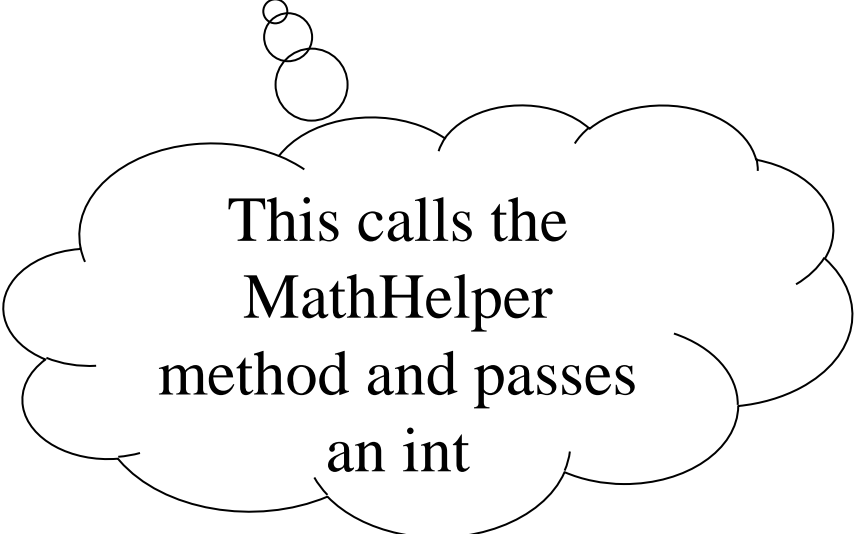
```
public class MyCalculator {
```

```
    public static void main(String[] args) {
```

```
        System.out.println(MathHelper.doubleInt(9));
```

```
    }
```

```
}
```



This calls the  
MathHelper  
method and passes  
an int

# Using two classes together in Eclipse

- Using two classes together in the same project is simple
- Create a new project and create the two classes within the same project
- Build the entire project and select the class that contains the main method

# Labwork for this week – Exercise 1

- Create a new project using the IDE (Eclipse)
- Write a Java program to prove that the working directory is the project directory. (*You can achieve this by creating a simple application with a main method. Call the `System.getProperty("user.dir")` method, this will return a `String` representing the working directory of your current program. Combine this with a `System.out.print` call and you'll see the working directory appear.*)

# Labwork for this week – Exercise 2

- Create a new project using the IDE (Eclipse)
- Add the MathHelper and MyCalculator classes as shown in this lecture
- Run the MyCalculator class

# Labwork for this week – Eclipse 3

- Create a new project using the IDE (Eclipse)
- Create a JFrame application called MyCalculatorGUI, this program must contain TWO JButtons with labelled “Double” and “Triple”. Add TWO JTextFields to the GUI, one text field to input an integer and one textfield to OUTPUT the “Double” or “Triple” of the number entered by the user.
- Create a second class called MathHelper, add two static methods to this class called *doubleInt* and *tripleInt*
- Implement ActionListener(s) for the above buttons so that the correct MathHelper method is called once the “Double” or “Triple” buttons are pressed. The user must enter the number to double and see the output of double and triple in the second TextField.
- Finally JAR this project and run the program from the command line