Experiment No.: 07

Title: Write a program to implement mathematical package for arithmetic, statistical and trigonometric operations.

Objectives:

1) To learn how to create package in java
2) To learn how to set class path
3) To learn how to use class and methods in another package

Theory:

Definition: A package is a collection of related classes and interfaces that provides access protection and namespace management.
The classes and interfaces that are part of the JDK are members of various packages that bundle classes by function: applet classes are in java.applet, I/O classes are in java.io, and the GUI widget classes are in java.awt. You can put your classes and interfaces in packages, too.

Creating a Package

You can easily create your own packages and put any number of class and interface definitions in them.
To create a package, you simply put a class or interface in it. To do this, you put a package statement at the top of the source file in which the class or interface is defined. For example, the following code appears in the source file Circle.java and puts the Circle class in the graphics package:

```
package graphics;
public class Circle extends Graphic implements Draggable {
. . .
}
```

The Circle class is a public member of the graphics package.
You must include a package statement at the top of every source file that defines a class or interface that is to be a member of the graphics package. So you would also include the statement in Rectangle.java and so on:

```
package graphics;
public class Rectangle extends Graphic implements Draggable
{
. . .
}
```

The scope of the package statement is the entire source file, so all classes and interfaces defined in Circle.java and Rectangle.java are also members of the graphics package. If you put multiple classes in a single source file, only one may be declared public and it must share the name of the source file's basename. Only public package members are accessible from outside the package.
Note: Some compilers might allow more than one public file per .java file. However, we recommend that you use the one-public-class-per-file convention, since it makes public classes easier to find and works for all compilers.
If you do not use a package statement, your class or interface ends up in the default package, which is a package that has no name. Generally speaking, the default package is only for small or temporary applications or when you are just beginning development. Otherwise, classes and interfaces belong in named packages.

Using Package Members

To use the classes and interfaces defined in one package from within another package, you need to import the package. The classes and interfaces that you import must be declared public. Only public package members are accessible outside the package in which they are defined. To use a public package member from outside its package, you must either
• refer to the member by its fully qualified (disambiguated) name,
• import the package member, or
• import the member's entire package.
Each is appropriate for different situations, as explained in the following sections.

Referring to a Package Member by fully qualified Name

So far, the examples in this book have referred to classes and interfaces by the name specified in their declaration (such as Rectangle, AlarmClock, and Sleeper). Such names are called short names. You can use a package member's short name if the code you are writing is in the same package as that member or if the member's package has been imported.
However, if you are trying to use a member from a different package and that package has not been imported, then you must use the member's long name, which includes the package name. This is the fully qualified name for the Rectangle class declared in the graphics package in the previous example:
graphics.Rectangle

You could use this qualified name to create an instance of graphics.Rectangle:
graphics.Rectangle

myRect = new graphics.Rectangle();

You'll find that using long names is okay for one-shot uses. But you'd likely get annoyed if you had to write graphics.Rectangle again and again. Also, your code would get very messy and difficult to read. In such cases, you can just import the member instead.

Importing a Package Member

To import a specific member into the current file, put an import statement at the beginning of your file before any class or interface definitions (but after the package statement, if there is one). Here's how you would import the Circle class from the graphics package created in the previous section:

import graphics.Circle;

Now you can refer to the Circle class by its short name:

Circle myCircle = new Circle();

This approach works just fine if you use just a few members from the graphics package. But if you use many classes and interfaces from a package, you really just want to import the whole package and forget about it.

Importing an Entire Package

To import all of the classes and interfaces contained in a particular package, use the import statement with the asterisk * wildcard character:

import graphics.*;

Now you can refer to any class or interface in the graphics package by its short name: Circle
myCircle = new Circle();
Rectangle myRectangle = new Rectangle();

The asterisk in the import statement can be used only to specify all of the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all of the classes in the graphics package that begin with "A":

```
import graphics.A*; // does not work
```

Instead, it generates a compiler error. With the import statement, you can import only a single package member or an entire package.
For your convenience, the Java runtime system automatically imports three packages for you:
• The default package (the package with no name)
• The java.lang package
• The current package

Disambiguating a Name
If by some chance a member in one package shares the same name with a member in another package and both packages are imported, you must refer to the member by its fully qualified name. For example, the previous example defined a class named Rectangle in the graphics package. The java.awt package also contains a Rectangle class. If both graphics and java.awt have been imported, then the following is ambiguous:

```
Rectangle rect;
```

In such a situation, you have to be more specific and indicate exactly which Rectangle class
you want by using the member's qualified name:

```
 graphics.Rectangle rect;
```

Setting the Class Path
If you must, you can change your class path. This can be done in either of two ways:
1. Set the CLASSPATH environment variable (not recommended).
2. Use the -classpath runtime option when you invoke the compiler or the interpreter.
We don't recommend setting the CLASSPATH environment variable because it can be long-lived (particularly if you set it in a login or startup script). It's also easy to forget about, and then one day, your programs won't work because the compiler or interpreter loads a crusty old class file instead of the one you want. An old, out-of-date CLASSPATH variable is a fruitful source of confusing problems.
The second option, setting the class path with the runtime option, is preferable because it sets the class path only for the current invocation of the compiler or the interpreter. Here's how to use the runtime option -classpath to set your class path.
Platform-specific Details: Using the -classpath Runtime Option:

UNIX:

```
javac -classpath .:~/classes:/JDK/lib/classes.zip
```

DOS shell (Windows):

```
javac -classpath
.;C:\classes;C:\JDK\lib\classes.zip
```

When you specify a class path in this manner, you completely override the current class path. Thus you must include the classes.zip file from the JDK in the class path. It's a good idea to include the current directory as well.
The order of the entries in the class path is important. When the Java interpreter is looking for a class, it searches the entries in your class path, in order, until it finds a class with the correct name.

The Java interpreter runs the first class with the correct name that it encounters and does not search the remaining entries.

Algorithm:

Algorithm:

- Create a package named MyMath.
- Create a class named Trig having data member representing angle in degree.
    - Define method to get sine of angle given in degrees.
    - Define method to get cosine of angle given in degrees.
    - Define method to get tangent of angle given in degrees.
    - Define method to get secant of angle given in degrees.
    - Define method to get cosecant of angle given in degrees.
    - Define method to get cotangent of angle given in degrees.
- Create a class named Arithmetic in same package.
    - Define methods for arithmetic operations like addition, subtraction, multiplication and division of float values in this class.
- Create a class named Stat in same package.
    - Define methods for Statistical operations like min, max, count, sum and average.
- Create a class PackDemo outside of MyMath package.
    - Write main method to demonstrate operations.
    - In main method create object of Trig and call methods to get sine, cosine, tangent, secant, cosecant and cotangent of given angle in radians.
    - In main method create object of Arithmetic class and calls its methods.
    - In main method create object of Stat and call methods to perform operations. Like min, max, count etc.

Key concepts: package, classpath, import.

Note: Please follow the naming conventions while writing the program.