Topic: Creation of package through the package directive.
OOP concepts involved: Classes, Interfaces, Inheritance, Polymorphism, Java Packages, Java CLASSPATH, Java PATH, Class Modifiers, Static Methods, Objects.
Programming generic concepts involved: Functions, Data Types, Variables, Access Modifiers.

➢ Theoric introduction

JAVA PACKAGES

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

BUILT-IN PACKAGES

These packages consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:

1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) **java.io:** Contains classed for supporting input / output operations.
3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4) **java.applet:** Contains classes for creating Applets.
5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like buttons, menus, canvas, etc).
6) **java.net:** Contain classes for supporting networking operations.

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

HOW TO CREATE A PACKAGE?

To create a package, you choose a name for the package (using lowercase letters is part of the naming convention) and put a *package* statement with that name at the top of *every source* file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, *package graphics*;) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
> javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Note: If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file. For example, you can define public class Circle in the file

Circle.java, define public interface Draggable in the file Draggable.java, define public enum Day in the file Day.java, and so forth.

You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be *package private*.

DIRECTORY STRUCTURE

The package name is closely associated with the directory structure used to store the classes. The classes (and other entities) belonging to a specific package are stored together in the same directory. Furthermore, they are stored in a sub-directory structure specified by its package name. For example, a class named *Circle* of package *com.figure* is stored as "*$BASE_DIR\com\figure\Circle.class*", where *$BASE_DIR* denotes the base directory of the package. Clearly, the "dot" in the package name (*com.figure*) corresponds to a sub-directory of the file system.

The base directory (*$BASE_DIR*) could be located anywhere in the file system. Hence, the Java compiler and runtime must be informed about the location of the *$BASE_DIR* so as to locate the classes. This is accomplished by an environment variable called CLASSPATH. CLASSPATH is similar to another environment variable PATH, which is used by the command shell to search for the executable programs.

Once we compiled our java files and created our package(s), its needed to set correctly the CLASSPATH for a correct execution of the program.

SETTING *CLASSPATH* FOR THE PROGRAM EXECUTION IN WINDOWS

CLASSPATH can be set by any of the following ways:

1. **Creating an environment variable:**

CLASSPATH can be set permanently in the environment:

In Windows, choose control panel **->** System **->** Advanced **->** Environment Variables **->** choose "System Variables" (for all the users) or "User Variables" (only the currently login user) **->** choose "Edit" (if CLASSPATH already exists) or "New" **->** Enter "CLASSPATH" as the variable name **->** Enter the required directories and/or JAR files (separated by semicolons) as the value (e.g., "`.;C:\javaproject\classes;D:\tomcat\lib\servlet-api.jar`"). Take note that you need to include the current working directory (denoted by '.') in the CLASSPATH.

To check the current setting of the CLASSPATH, issue the following command:

```
> SET CLASSPATH
```

2. **Setting it temporarily just for a particular command line session**

CLASSPATH can be set temporarily for a particular shell session by issuing the following command:

```
> SET CLASSPATH=.;C:\javaproject\classes;
```

And then we could do the following execution, with the CLASSPATH already temporarily configured:

```
> java com.figure.Circle
```

3. **Using javac command-line option**

Instead of using the CLASSPATH environment variable, we can also use the command-line option *-classpath* or *-cp* of the javac and java commands, for example:

```
> java -classpath C:\javaproject\classes com\figure\Circle
```

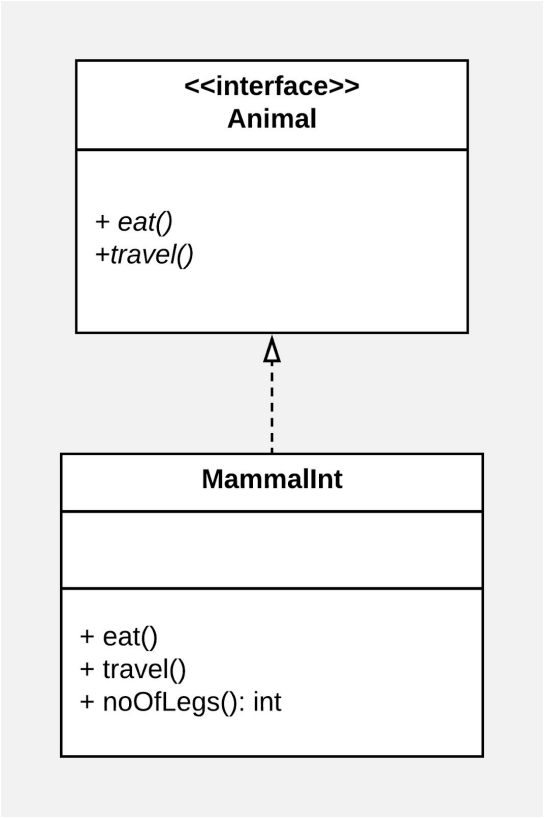
➢ **Statement**

Create an Interface called *Animal*, it needs to contain 2 abstract methods (eat and travel). After that make a *Class* named *MammalInt* that implements the *Animal* interface. *MammalInt* needs to have its own implementation of the *Animal* interface methods, and also it needs to add a third method named *noOfLegs()* that return an int data type.

Assign to both previous java files a package named *animals*. Compile the java file using the command line and use the javac command (`-d Destination_folder`) to create the package.

For the execution process, make use of the third option for setting the CLASSPATH.

➢ **Class design (UML)**



➢ **Program Code**

Animal.java

```
package animals;
// File path: C:\javaCode\
interface Animal {
       // Abstract methods
       public void eat();
       public void travel();
}
```

MammalInt.java

```
package animals;
// File path: C:\javaCode\
public class MammalInt implements Animal {

   public void eat() {
      System.out.println("Mammal eats");
   }

   public void travel() {
      System.out.println("Mammal travels");
   }

   public int noOfLegs() {
      return 0;
   }

   public static void main(String args[]) {
      // Declaring and Instantiating a MammalInt Object
      MammalInt m = new MammalInt();

      // Calling methods
      m.eat();
      m.travel();
   }

}
```

➢ Command Line Process

COMPILATION PROCESS

- *Checking directory before compiling*

```
C:\javaCode> dir
```

```
 Directory of C:\javaCode

09/23/2018  03:21 PM    <DIR>          .
09/23/2018  03:21 PM    <DIR>          ..
09/23/2018  03:17 PM               108 Animal.java
09/23/2018  03:21 PM               496 MammalInt.java
               2 File(s)            604 bytes
               2 Dir(s)  615,970,095,104 bytes free
```

- *Compiling java files*

```
C:\javaproject\classes> javac -d . Circle.java
```

- *Checking the after directory:*

```
C:\javaproject\classes> dir
```

```
 Directory of C:\javaCode

09/23/2018  03:21 PM    <DIR>          .
09/23/2018  03:21 PM    <DIR>          ..
09/23/2018  03:17 PM               108 Animal.java
09/23/2018  03:21 PM    <DIR>          animals
09/23/2018  03:21 PM               496 MammalInt.java
               2 File(s)            604 bytes
               3 Dir(s)  615,966,048,256 bytes free
```

EXECUTION PROCESS

- **SETTING CLASSPATH WITH THE THIRD WAY (***CLASSPATH JAVA COMMAND)*

```
C:\javaCode>java -cp C:\javacode animals.MammalInt
```

```
Mammal eats
Mammal travels
```

➢ Conclusions

Packages are very useful for grouping classes, interfaces, enumerations and annotations of our own creation that are about related topics, because they provide us access protection and namespace management.

As we know, the Java Api manages its own built-in packages where within every package there are several class implementations. The most common Java Api built-in packages are: **java.lang**, **java.io**, **java.util**, **java.swing** and **java.net**.

A package always needs to be at the first line of the java file we are creating. Using lowercase letters is a naming convention for the packages.

If you do not use a package statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.