CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Java Packages

Packages

- A package is a grouping of related **types** providing access protection and name space management
- Types refer to classes, interfaces, enumerations, and annotation types
- Types are often referred to as classes and interfaces since enumerations and annotation types are special kinds of classes and interfaces

Advantages of using a package in Java

- **Reusability**: While developing a project in java, we often feel that there are few things that we are writing again and again in our code. Using packages, you can create such things in form of classes inside a package and whenever you need to perform that same task, just import that package and use the class.
- **Better Organization**: in large projects where we have several hundreds of classes, it is always required to group the similar types of classes in a meaningful package name so that you can organize your project better and when you need something you can quickly locate it and use it, which improves the efficiency.
- **Name Conflicts**: We can define two classes with the same name in different packages so to avoid name collision, we can use packages

Types of packages

- User defined package: The package we create is called user-defined package.
- Built-in package: The already defined packages
  - java.io.*, java.lang.*, java.util.*

Creating a package

- To create a package, you choose a name for the package 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
- If you do not use a package statement, your type ends up in an unnamed package
- Use an unnamed package only for small or temporary applications

Placing a Class in a Package

- To place a class in a package, we write the following as the first line of the code (except comments)

  package <packageName>;

  package myownpackage;

Example package

- Suppose you write a group of classes that represent graphic objects, such as circles, rectangles, lines, and points
- You also write an interface, Draggable, that classes implement if they can be dragged with the mouse

```
//in the Draggable.java file
public interface Draggable {}

//in the Graphic.java file
public abstract class Graphic {}

//in the Circle.java file
public class Circle extends Graphic implements Draggable {}

//in the Rectangle.java file
public class Rectangle extends Graphic implements Draggable { }

//in the Point.java file
public class Point extends Graphic implements Draggable {}

//in the Line.java file
public class Line extends Graphic implements Draggable {}
```

Placing a class to a package

```
package SchoolClasses;

public class StudentRecord {
    private String   name;
    private String   address;
    private int      age;
    :
```

the StudentRecord.class file must be placed under the directory named SchoolClasses.

In case of an unnamed package, the current directory is implied.

Using Classes from Other Packages

- To use a public package member (classes and interfaces) from outside its package, you must do one of the following
  - Import the package member using import statement
  - Import the member's entire package using import statement
  - Refer to the member by its fully qualified name (without using import statement)

Importing Packages

- To be able to use classes outside of the package you are currently working in, you need to import the package of those classes.
- By default, all your Java programs import the java.lang.* package, that is why you can use classes like String and Integers inside the program even though you haven't imported any packages.
- The syntax for importing packages is as follows:
- **import <*nameOfPackage*>;**

Importing a Class or a Package

```
// Importing a class
import java.util.Date;

// Importing all classes in the
// java.util package
import java.util.*;
```

"*" stands for classes / interfaces only, not sub-packages if any.

Using Classes of other packages via fully qualified path

```
public static void main(String[] args) {
    java.util.Date x = new java.util.Date();
}
```

- ... if java.util.Date is not imported.

Package & Directory Structure

- Packages can also be nested.
- Java interpreter expects the directory structure containing the executable classes to match the package hierarchy.
- There should have same directory structure, ./myowndir/myownsubdir/myownpackage directory for the following package statement

package myowndir.myownsubdir.myownpackage;

Example

- import java.util.Scanner
- Here:
  → **java** is a top level package
  → **util** is a sub package
  → and **Scanner** is a class which is present in the sub package **util**.

Subpackages with *

- The wild card import like package.* should be used carefully when working with subpackages.
- For example:
  - we have a package **abc** and inside that package we have another package **foo**, now **foo** is a subpackage.
  - classes inside abc are: Example1, Example 2, Example 3
    classes inside foo are: Demo1, Demo2
- import abc.*;
  - will only import classes Example1, Example2 and Example3 but it will not import the classes of sub package.
- To import the classes of subpackage you need to
  - import abc.foo.*;
  - will import Demo1 and Demo2 but it will not import the Example1, Example2 and Example3.
- To import all the classes present in package and subpackage, we need to use two import statements like this:
  - import abc.*;
  - import abc.foo.*;

Packages and imports

- A class can have only one package declaration but it can have more than one package import statements.

```
package abcpackage; //This should be one
import xyzpackage;
import anotherpackage;
import anything;
```

Class name conflict

- Lets say we have two packages **abcpackage** and **xyzpackage** and both the packages have a class with the same name: JavaExample.java.
- Now suppose a class import both these packages like this:
  - import abcpackage.*;
    import xyzpackage.*;
  - This will throw compilation error.
- To avoid such errors you need to use the fully qualified name method
  - abcpackage.JavaExample obj = new abcpackage.JavaExample();
    xyzpackage.JavaExample obj2 = new xyzpackage.JavaExample();
  - This way you can avoid the import package statements and avoid that name conflict error.

Java static import

- Static import allows you to access the static member of a class directly without using the fully qualified name.

```
import static java.lang.System.out;
import static java.lang.Math.*;
class Demo2{
   public static void main(String args[])
   {
      //instead of Math.sqrt need to use only sqrt
      double var1= sqrt(5.0);
      //instead of Math.tan need to use only tan
      double var2= tan(30);
      //need not to use System in both the below statements
      out.println("Square of 5 is:"+var1);
      out.println("Tan of 30 is:"+var2);
   }
}
```

When to use?

- If you are going to use static variables and methods a lot then it's fine to use static imports.
  - for a code with lot of mathematical calculations, you may want to use static import.
- **Drawback:** It makes the code confusing and less readable so if you are going to use static members very few times in your code then probably you should avoid using it
- You can also use wildcard(*) imports.

Managing Source and Class Files

- Many implementations of the Java platform rely on hierarchical file systems to manage source and class files, although *The Java Language Specification* does not require this.
- Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is .java. For example:

```
//in the Rectangle.java file
package graphics;
public class Rectangle { ... }
```

Managing Source and Class Files

- Then, put the source file in a directory whose name reflects the name of the package to which the type belongs:

  .....\graphics\Rectangle.java

- **class name** – graphics.Rectangle
- **pathname to file** – graphics\Rectangle.java
- By convention a company uses its reversed Internet domain name for its package names.
- The Example company, whose Internet domain name is example.com, would precede all its package names with com.example.
- Each component of the package name corresponds to a subdirectory.
- So, if the Example company had a com.example.graphics package that contained a Rectangle.java source file, it would be contained in a series of subdirectories like this:
  - ....\com\example\graphics\Rectangle.java

Managing Source and Class Files

- When you compile a source file, the compiler creates a different output file for each type defined in it.
- The base name of the output file is the name of the type, and its extension is .class.
- The compiled files in the example will be located at:

<path to the parent directory of the output files>\com\example\graphics\Rectangle.class

- Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name.
- However, the path to the .class files does not have to be the same as the path to the .java source files.

Managing Source and Class Files

- By doing this, you can give the classes directory to other programmers without revealing your sources.
- The full path to the classes directory,
  - <path_two>\classes, is called the *class path*, and is set with the CLASSPATH system variable.
  - Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.
- For example, if <path_two>\classes is your class path, and the package name is com.example.graphics, then the compiler and JVM look for .class files in
  - <path_two>\classes\com\example\graphics.

Setting the CLASSPATH System Variable

- A class path may include several paths, separated by a semicolon (Windows) or colon (UNIX). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in your class path.

To display the current CLASSPATH variable, use these commands in Windows and UNIX (Bourne shell):

```
In Windows:    C:\> set CLASSPATH
In UNIX:       % echo $CLASSPATH
```

To delete the current contents of the CLASSPATH variable, use these commands:

```
In Windows:    C:\> set CLASSPATH=
In UNIX:       % unset CLASSPATH; export CLASSPATH
```

To set the CLASSPATH variable, use these commands (for example):

```
In Windows:    C:\> set CLASSPATH=C:\users\george\java\classes
In UNIX:       % CLASSPATH=/home/george/java/classes; export CLASSPATH
```