19-July-2017 / Weekend 12/08/2017

PACKAGES

Definition

Packages are collecⁿ of similar type of classes & ~~data~~ interfaces.

Need

If we gather similar type of APIs in one place it would be easier for programmer to search and use ~~for~~ a fⁿ.

Imp. Rule

Imp. No class can exist without any package in Java.

- So, if you have not given any package making command in a '.java' file, then compiler will create a 'default' Package in the same folder and then put the '.class' file into that default package.
- 'Default' packages don't have any name.
- The classes of 'default' package can't be used outside that package.
- Also, Packages are never OverRiden in Java.

90% of Java packages are present in these two packages.

1. Java
   Child Packages
   (i) awt
   (ii) lang
   (iii) net
   (iv) IO

2. Javax
   (i) swing
   (ii) Security
   etc -

- -

Linking Command in java → import

- Single import statement is used to link only one (1) package at a time with your program.
- All the import statements must be written above the class keyword.

```
import java.awt.* ;
import java.util.* ;
```
} * to link all classes of the package.

- Packages are accessed via Association.

Now,
Example:-

```
    import  java.awt.*;
    import  java.util.*;

    class Temp
    {
     psvm()
     {
     Frame f = new Frame();
     Date  d = new Date();
        Sop(d);
     Button b = new  Button();
     }
    }
```

- Here, By linking all classes of 'awt' & 'util' packages we have degraded performance of program.

So, instead now we will link only those classes that we use in our program.
Example:-

```
  import  java.awt.Frame;
  import  java.awt.Button;
  import  java.util.Date;
```

if we don't write here prog. gives compil^n error.

```
  Class Temp
  {
   psvm()
   {
   Frame  f = new Frame();
   Date   d = new Date();
                              Sop(d);
      Button b = new Button();
                            }
                            }
```

• We can also use a class without impo-
-rting package

Escample:-
```
class Temp
{
    psv m().
    {
        java.awt.Frame f=new java.awt.Frame
                                        ();
    }
}
```
Done

⇒ This is a long method & hence is considered as bad programming.

Imp. • 'lang' package is imported to each java file by the compiler, that's why we use 'String' & 'System' class without importing. 'lang' package. 'System' & 'String' are classes of 'lang' package.

Que → Why only 'lang' is imported?
Ans → Becoz it contains general purpose Classes. All program will use one or other class of "lang" package.
Thats why 'lang' is ~~im~~ implicitly included.

Package Making Command.

Rule
Package making command must be the first line in a java file.

D:\f1> Temp.java

Example:-

```
package p1;

class Temp
{
void show()
{
Sop ("pkg p1");
}
~~psv~~
 psvm()
 {
  Temp t1 = new Temp ();
    t1. show();
  }
}
```

Done

- ~~Aft~~ After compilaⁿ O/S will treats packages as folder but its not.
- D:\f1 > javac Temp.java → compiles ~~b/w~~ but not creates package
- D:\f1 > java Temp → Will gives compilⁿ error

- If we have given any package making command in any '.java' file, we can't compile it normally.
- We have to use a switch of 'javac' tool.

```
D:/f1 >javac -d e:/f2 Temp.java
```

$\underset{\text{destination}}{\text{-d}}$

(i) This will make a package
(ii) And Ask for its destination
↳ package's

Note:-
'-d' can also be used without package making command. It will just go to the given destination 4 create a 'default' package and put '.class' file in this 'default' package.

Now, To Access package that we created
→ e:\f2 >p1
↳ o/s treats as folder

→ ~~e:\3f2/p1 > java Temp1~~ → But we can't use it as folder.

So, To Access/Execute

```
e:\f2 > java p1.Temp1
```

Now, if we want to make package elsewhere

Done ✓ • D:\f1 > javac -d D:\f1 Temp.java
↳ New desired directory.

Done • D:\f1 > javac -d . Temp.java
current dir. in DOS Represent ⟷ Makes in current directory i.e. Here → 'D:\f1'.

~~Other Scenar~~

OTHER SCENARIOS

• Keeping More than One class into a single package

There are two ways for achiving this.

(i) Keep all the classes in single java file

Example:-
```
class Temp1
{
}
Class Temp2
{
}
```

V.Imp ✓. Packages are also made to use a class Outside a folder/package.

- So, we use 'public' in all class to access all ^any^ classes outside the package

Now, Above program is Rewritten as :-

```
public class Temp1
{
}

public class Temp2
{
}

Public class Temp3
{
}
```

- But we can't Keep more than One (1) public class in single '.java' file.

Imp:- **Rule**

You can't have more than one 'public class' in single java file and the name of that '.java' file will be the same as your 'public class' name

x ——— x ——— x

~~Reason for above Rule.~~

~~1. Becoz of implicit compiling~~

20-July-2017

- We need to make 'class' & 'fn' public to use them outside their package.

▶ Reason for Rule (on Left H$ page)

To achive

- Because of implicit compiling. in java

Example :-

```
class Temp
{
psvm()
{
new Temp1();
new Temp2();
new Temp3();
- - - - - - -
 -  -  -  -
new Temp10();
}
}
```

Now create All Below classes in new separate file.

```
class Temp1
{
}
Class Temp2
{
}
= = = =
class Temp 10
{
}
```

All these '.java' files are in same folder.

- Before 'implicit compiling', we had to compile 'Temp1', 'Temp2', ... 'Temp10' before using in & compiling 'Temp'.

- But in & After concept of 'implicit compiling' we only have to compile 'Temp' directly & others will compile automatically.

- But 'implicit compiling' only works in same package.

- 'Implicit compiling' only works if name of '.java' file is same as public 'class' name. ~~of~~

Example

1. A.java to Hathi.java

```
public Class A
{
}
```

B.java

```
class B
{
    psvm()
  {
    new A(); → Object of class A.
  }
}
```

Done

- Here, directly compiling 'B' ~~we ca~~ will also have 'A.java' compiled successfully, as name of '.java' file is same as 'public class' name.

- If instead we name 'A.java' as 'xyz.java' & directly compile B, compiler will not be able to search for 'class A' as it ~~requires~~ searches only in ditto named '.java' file. And gives error.

Done

- Hence, As we see Above to impliment 'implicit compiling' we name '.java' file as 'public class' name

- Also if we would include more than one ~~java~~ 'public class' in single java file we will not be able to give ~~any~~ exact name to '.java' file.

Steps for Keeping more than one ~~public~~ java file ~~for each~~ public class. in single ~~class~~ package

(i) Step #1 → Make a ~~parat~~ separate .java file for each public class. & give same package making command in all of them.

D:\f1> Temp1.java	D:\f1> Temp2.java	D:\f1> Temp3.java
package p1	package p1;	package p1;
public class Temp1	public class Temp2	public class Temp3
{	{	{
}	}	}

(ii) Step#2 → Keep a same package making com-~~ke~~ -mand in each java file.

Now, Compiling.

```
D:\f1>javac -d . *.java
```

↳ compiles all 3 in one go.

- All the ~~part~~ '.class' files will go in package 'p1'

And

As in java packages are not OverRiden ~~it~~ compile will just keep adding new separate '.class' files to same existing package 'p1' or create p1 if its not there.

Now,

Linking a Self-made package To another Self-made package in Another drive.

D:/f1 > Temp1.java

```
package p1;

During c: [public] class Temp1
{
During c : [public] void show()
  {
   Sop ("pkg p1");
  }
public static void main()
{
 Temp1 = new Temp1();
   t1. show();
OR
 new Temp1().show();
 }
}
```

Compiling

```
D:\f1 >javac -d . *.java
```

This successfully creates package p1 4 compiles Temp1. java. in it.

Now, linking 'p1' to another Self created package 'p2' in 'E' drive.

```
E:\f2> Temp2.java

package p2;

import p1.*;

class Temp2
{
  psvm()
 {
Temp t1 = new Temp1();     } OR  new Temp1().show()
       t1.show();
  }
}
```

Done

Compiling #1

```
E:\f2> javac -d . *.java
```

- It will not compile & give error, as we didn't tell compiler where to look for p1 while importing it.

- So, we have to define a 'classpath' by.

```
E:\f2 > set classpath=D:\f1;
```

No space in b/w

Compiler will search for 'p1' only in 'class path'.

To check, current class path

```
E:\f2 > set classpath
```

It will again give error as we have not made class 'Temp1' public.

Compiling #2

- Making class 'Temp1' public
- Compiling Again.

It will again give error. as we have not ~~made~~ made 'show()' fn public.

Compiling #3

- Making 'show()' fn public
- This ~~typ~~ time, we ~~ss~~ sucessfully compile the java file.

✓ As we have set classpath, we can even execute 'Temp1' from any location.

```
E:\f2> java p1.Temp1
```

Rule:
For all those packages which are not found in rt.jar file we have to set the class-path. The classpath of 'rt.jar' is implicitly set.