Object Oriented Programming with Java
Practical-5

<u>Introduction to Package and Interface</u>

What is Package in Java?

PACKAGE in Java is a collection of classes, sub-packages, and interfaces. It helps organize your classes into a folder structure and make it easy to locate and use them. More importantly, it helps improve code reusability.

Each package in Java has its unique name and organizes its classes and interfaces into a separate namespace, or name group.

Although interfaces and classes with the same name cannot appear in the same package, they can appear in different packages. This is possible by assigning a separate namespace to each Java package.

Syntax:-

```
package nameOfPackage;
```

The following video takes you through the steps of creating a package.

How to Create a package?

Creating a package is a simple task as follows

- Choose the name of the package
- Include the package command as the first line of code in your Java Source File.
- The Source file contains the classes, interfaces, etc you want to include in the package
- Compile to create the Java packages

Step 1) Consider the following package program in Java:

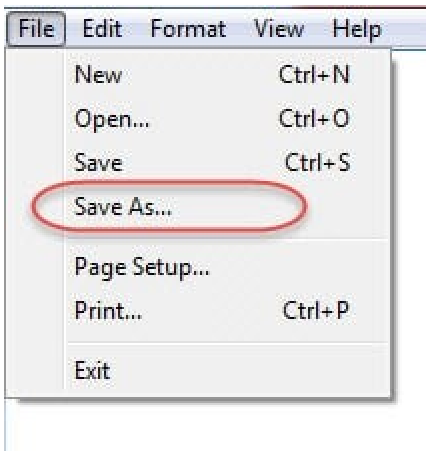
```
package p1;

class c1(){
public void m1(){
System.out.println("m1 of c1");
}
public static void main(string args[]){
c1 obj = new c1();
obj.m1();
}
}
```

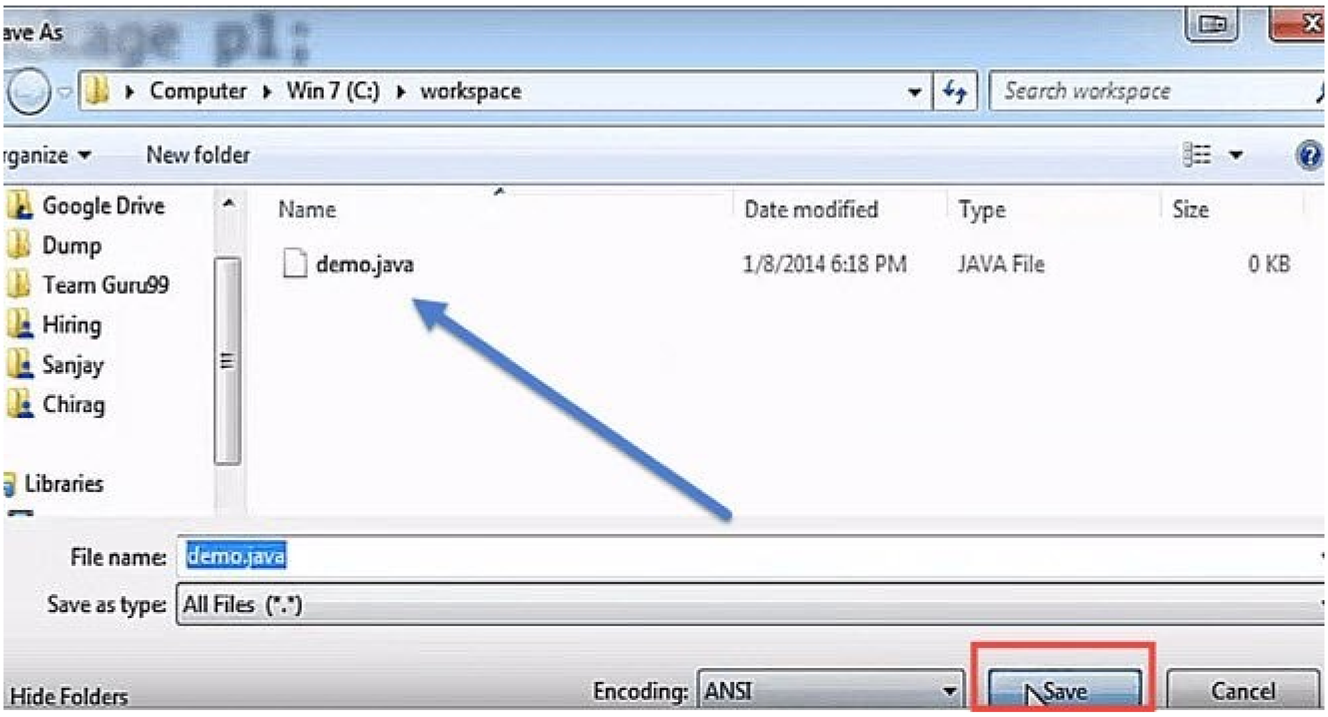
Here,

1. To put a class into a package, at the first line of code define package p1
2. Create a class c1
3. Defining a method m1 which prints a line.

4. Defining the main method
5. Creating an object of class c1
6. Calling method m1

Step 2) In next step, save this file as demo.java

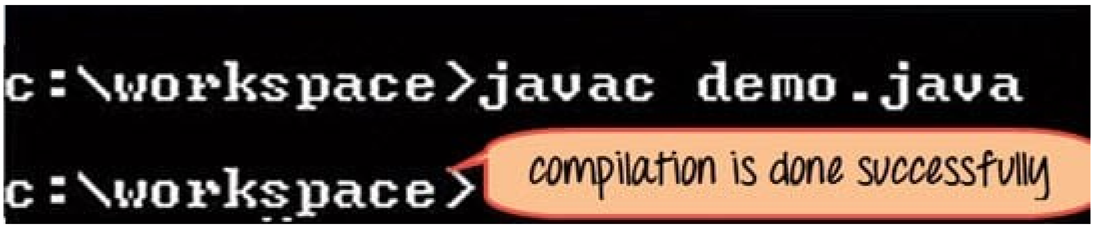




Name | Date modified | Type | Size
demo.java | 1/8/2014 6:18 PM | JAVA File | 0 KB

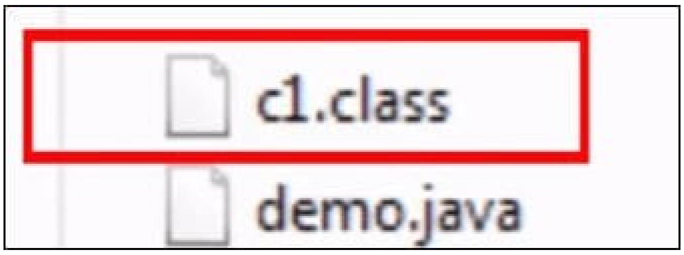
File name: demo.java
Save as type: All Files (*.*)

Step 3) In this step, we compile the file.



```
c:\workspace>javac demo.java

c:\workspace>
```

The compilation is completed. A class file c1 is created. However, no package is created? Next step has the solution

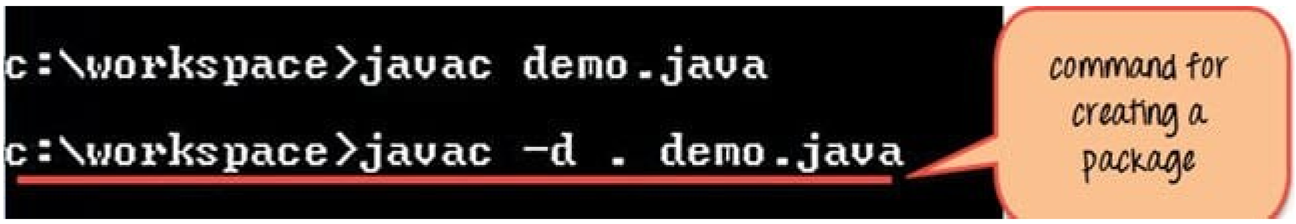


Step 4) Now we have to create a package, use the command

```
javac -d . demo.java
```

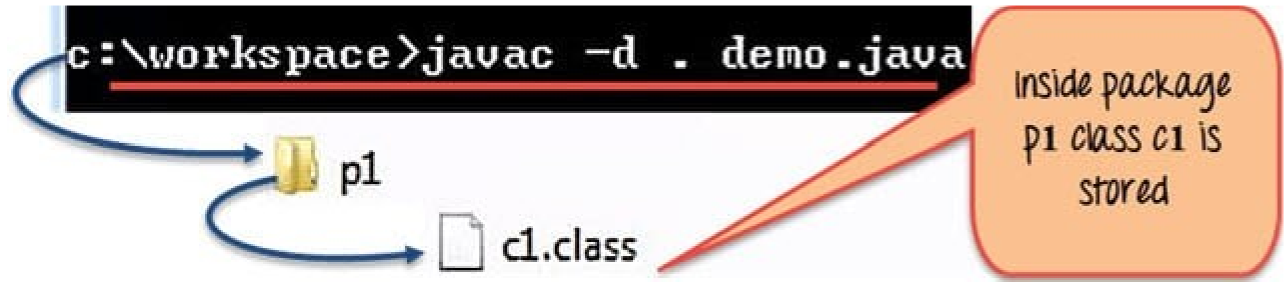
This command forces the compiler to create a package.

The **“.”** operator represents the current working directory.



```
c:\workspace>javac demo.java

c:\workspace>javac -d . demo.java
```

Step 5) When you execute the code, it creates a package p1. When you open the java package p1 inside you will see the c1.class file.

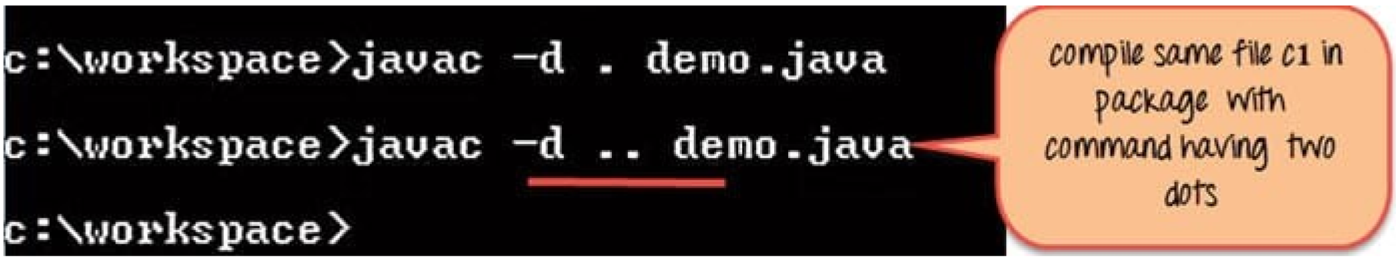


```
c:\workspace>javac -d . demo.java
```

Step 6) Compile the same file using the following code

```
javac -d .. demo.java
```

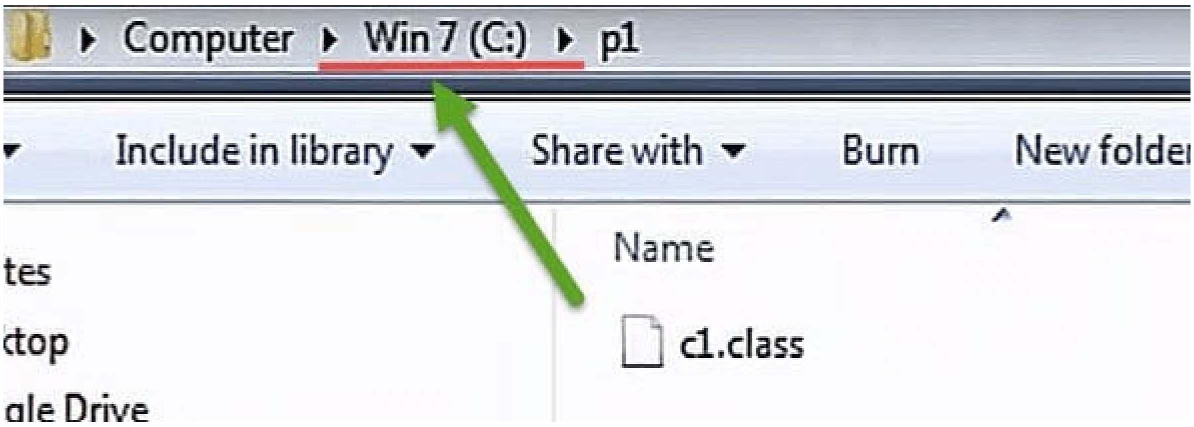
Here “..” indicates the parent directory. In our case file will be saved in parent directory which is C Drive



```
c:\workspace>javac -d . demo.java

c:\workspace>javac -d .. demo.java

c:\workspace>
```

compile same file c1 in package with command having two dots

File saved in parent directory when above code is executed.



Example: To import package
Step 1) Copy the code into an editor.

```
package p3;
import p1.*; //imports classes only in package p1 and NOT  in the sub-package p2
class c3{
  public   void m3(){
     System.out.println("Method m3 of Class c3");
  }
  public static void main(String args[]){
    c1 obj1 = new c1();
    obj1.m1();
  }
}
```

Step 2) Save the file as Demo2.java. Compile the file using the command **javac –d . Demo2.java**

Step 3) Execute the code using the command **java p3.c3**

Interfaces

Another way to achieve abstraction in Java, is with interfaces.

An `interface` is a completely "**abstract class**" that is used to group related methods with empty bodies:

Example

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the `implements` keyword (instead of `extends`). The body of the interface method is provided by the "implement" class:

Example

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}
```

```
// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default `abstract` and `public`
- Interface attributes are by default `public`, `static` and `final`
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example

```
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Java Inner Classes (Nested Classes)

Java inner class or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Need of Java Inner class

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Difference between nested class and inner class in Java

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
    1. Member inner class
    2. Anonymous inner class
    3. Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
Local Inner Class	A class was created within the method.
Static Nested Class	A static class was created within the class.
Nested Interface	An interface created within class or interface.

Java Member Inner class

A non-static class that is created inside a class but outside a method is called **member inner class**. It is also known as a **regular inner class**. It can be declared with access modifiers like public, default, private, and protected.

Example

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

Java Anonymous inner class

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).
2. Interface

Example

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

Java Local inner class

A class i.e., created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them. However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Example

```
public class localInner1{
    private int data=30;//instance variable
    void display(){
     class Local{
      void msg(){System.out.println(data);}
     }
     Local l=new Local();
     l.msg();
    }
    public static void main(String args[]){
     localInner1 obj=new localInner1();
     obj.display();
    }
   }
```

Java static nested class

A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of the outer class, including private.
- The static nested class cannot access non-static (instance) data members

Example

```
class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }
}
```