Packages
Inner Classes

Sisoft Technologies Pvt Ltd
SRC E7, Shipra Riviera Bazar, Gyan Khand-3, Indirapuram, Ghaziabad
Website: www.sisoft.in Email:info@sisoft.in
Phone: +91-9999-283-283

Learning - Topics

- Packages
- Import
  - Static Import
- Nested Class
  - Static Nested Class
  - Inner Class
    - Member Inner Class
    - Method Local Inner Class
    - Anonymous Inner Class
- Lambda Functions

Packages

- *Packages are containers for* classes. They are used to keep the class name space compartmentalized
- The package is both a naming and a visibility control mechanism
- The package name is the first statement in any java program
- If you omit the **package statement, the class names are** put into the default package, which has no name
- The general form of the **package** statement:

  package *pkg;*

Finding Packages

How does the Java run-time system know where to look for packages that you create?

- First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
- You can specify a directory path or paths by setting the **CLASSPATH environmental variable.**
- you can use the **-classpath option** with **java and javac to specify the** path to your classes.

Import Statement

- Java includes the **import statement to bring certain** classes, or entire packages, into visibility
- The general form of the **import statement:**

  import *pkg1 [.pkg2].(classname | *);*
- In a Java source file, **import statements** occur immediately following the **package statement (if it exists) and before** any class definitions.
- The basic language functions are stored in a package inside of the **java package called java.lang,** it is implicitly imported by the compiler for all programs.

Static Import Statement

- By following **import** with the keyword **static, an import** statement can be used to import the static members of a class or interface
- When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class. This simplifies and shortens the syntax required to use a static member

Static Import Statement

- There are two general forms of the **import static statement:**
- Brings into view a single name :

import static *pkg.type-name.staticmember-name;*

- Here, *type-name is the name of a class or* interface that contains the desired static member. Its full package name is specified by *pkg. The name of the member is* specified by *static-member-name.*
- The second form of static import imports all static members of a given class or interface. Its general form is shown here:

  import static *pkg.type-name.*;*

Nested Class

- The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass {
... class NestedClass { ...
}
}
```

- A nested class is a member of its enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

Nested Class - Categories

- Nested classes are divided into two categories: static and non-static.
  - Nested classes that are declared static are called *static nested classes*.
  - Non-static nested classes are called *inner classes*

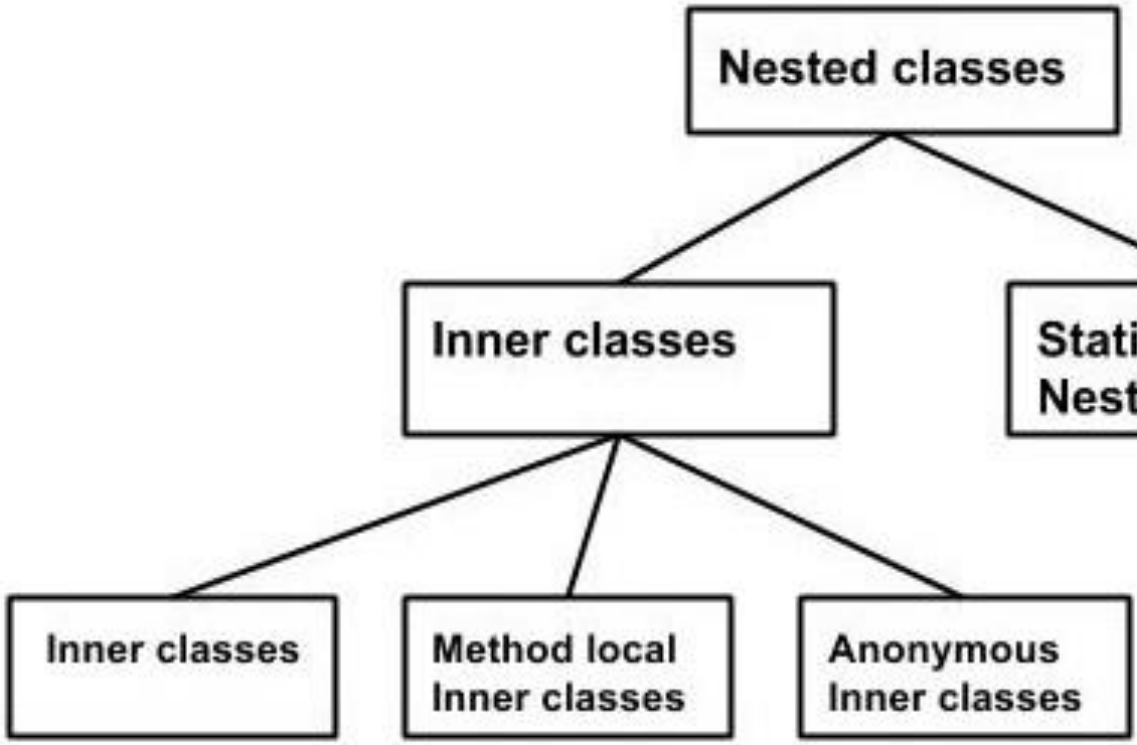
```
class OuterClass { ...
static class StaticNestedClass { ... }
class InnerClass { ... }
}
```

- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class

Inner class

- An inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.
- Also, because an inner class is associated with an instance, it cannot define any static members itself.
- There are three types of inner classes:
  - Member Inner Class
  - Anonymous Inner Class
  - Local Inner Class

Nested Class Types



Member inner class

- A **member class** is defined at the top level of the class. It may have the same access modifiers as variables (public, protected, package, static, final), and is accessed in much the same way as variables of that class.

Instantiating an inner class

- **Outside to outer class**

  Outerclass.Innerclas inRef =new Outerclass().new Innerclas()

- **Within outer class**

Innerclas inRef =new Innerclas();

Method **local inner class**

- A **local inner class** is defined within a method, and the usual scope rules apply to it.
- It is only accessible within that method, therefore access restrictions (**public**, **protected**, **package**) do not apply.
- The objects (and their methods) created from this class may not persist after the method returns, a local inner class *may not refer to parameters or non-**final** local variables of the method*.

anonymous inner class

- An **anonymous inner class** is one that is declared and used to create one object (typically as a parameter to a method), all within a single statement.
- It should be used to override method of class or interface
- An anonymous inner class may extend a class:
  - **new *SuperClass*(*parameters*){ *class body* }**Here, ***SuperClass*** is not the name of the class being defined, but rather the name of the class being extended. The ***parameters*** are the parameters to the constructor for that superclass.
- An anonymous inner class may implement an interface:
  - **new *Interface*(){ *class body* }**

Example Anonymous inner class

```
class Myclass{
 void go()
{
    new Bar(){  //Extending Class
        void doSomething(){SOP("doSomeThings")}
   }.doSomeThing() ;
   // By Implementing interface
   new Foo(){ void foof(){SOP("Foof")}}.foof();
   }
}

interface Foo{ void foof(); }
class Bar{ void dostuff(Foo f){} }
```

Lambda Expression

Introduction

- Provides a clear and concise way to represent one method interface using an expression
- An interface which has only one abstract method is called functional interface
- Lambda expression provides implementation of *functional interface*
- Sort of replacement for java inner anonymous class
- Java lambda expression is treated as a function, so compiler does not create .class file
- Lambda Expression requires less coding

Lambda Expression

There are three components to Lamba Expression

1) **Argument-list:** It can be empty or non-empty as well.
2) **Arrow-token:** It is used to link arguments-list and body of expression.
3) **Body:** It contains expressions and statements for lambda expression.

Lambda Expression Syntax

```
(argument-list) -> {body}
```

Using interface without Lambda

```java
interface Bus{
    public void travel();
}
public class WithoutLambaExample {
    public static void main(String[] args) {
        int capacity = 10;

        //without lambda, Drawable implementation using anonymous class
        Bus b=new Bus(){
            public void travel(){System.out.println("Traveling "+capacity);}
        };
        b.travel();
    }
}
```

Output -

```
Drawing 10
```

Using interface with lambda

```java
@FunctionalInterface  //optional
interface Bus{
    public void travel();
}

public class LambdaExpressionExample {
    public static void main(String[] args) {
        int capacity=10;

        //with lambda
        Bus b=()->{
            System.out.println("Traveling "+ capacity);
        };
        b.travel();
    }
}
```

Output -

```
Drawing 10
```