
# JAVA BASICS

## 4KCJ001 - LECTURE 9

By
Sudha Agarwal
sudha.agarwal@4kitsolutions.com

http://www.4kitsolutions.com/

# INDEX

- Nested Classes
- Functional Interfaces
- Lambda Expressions

# FUNCTIONAL INTERFACE

❖A functional interface is an interface that contains only one abstract method.

❖They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.

❖A functional interface can have any number of default methods. Runnable, ActionListener, Comparable are some of the examples of functional interfaces.

❖Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

# FUNCTIONAL INTERFACE

```
class Test
{
    public static void main(String args[])
    {
        // create anonymous inner class object
        new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                System.out.println("New thread created");
            }
        }).start();
    }
}
```

# LAMBDA EXPRESSIONS

❖Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming, and simplifies the development a lot.

❖lambda expressions implement the only abstract function and therefore implement functional interfaces

# LAMBDA EXPRESSIONS

❖A Lambda expression is a block of code that we can pass around to execute.

❖Passing a block of code to a function is something that we as Java programmers are not used to.

❖All our behavior defining code is encapsulated inside method bodies and executed through object references as it is with this code:

# LAMBDA EXPRESSIONS

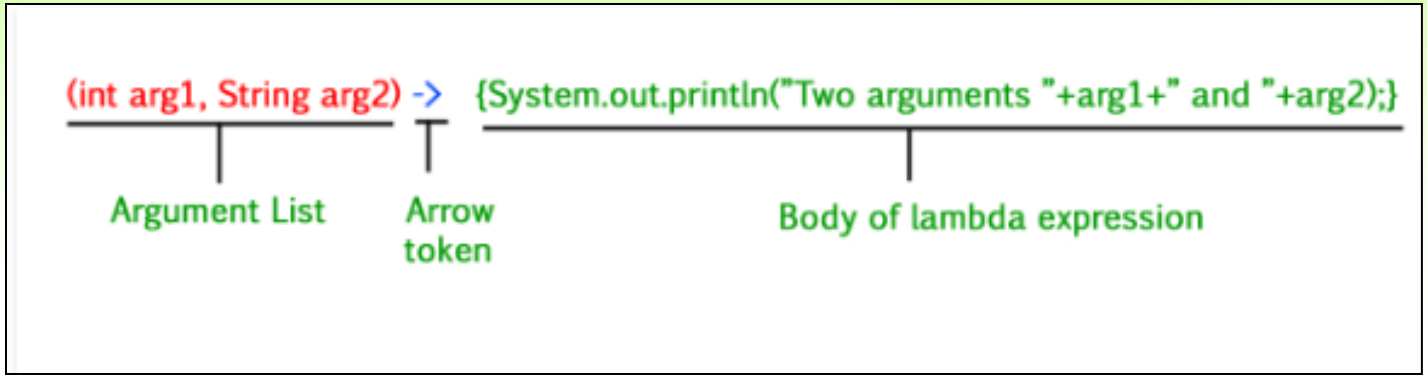
```
public class LambdaDemo {
    public void printSomething(String something) {
        System.out.println(something);
    }

    public static void main(String[] args) {
        LambdaDemo demo = new LambdaDemo();
        String something = "I am learning Lambda";
        demo.printSomething(something);
    }
}
```

# LAMBDA EXPRESSIONS

❖This is classic OOP style of hiding method implementations from the caller. The caller simply passes a variable to the method which then does something with the value of the variable and returns another value or produces a side effect as it is in our case.

❖We are now going to see an equivalent implementation that uses behavior passing other than variable passing. To achieve this, we have to create a functional interface that defines that abstracts the behavior instead of a method.

# LAMBDA EXPRESSIONS




http://www.4kitsolutions.com/

# LAMBDA EXPRESSIONS

- Syntax:
  - Lambda operator -> body

- where lambda operator can be:
- Zero parameter:
- () -> System.out.println("Zero parameter lambda");

- One parameter:–
- (p) -> System.out.println("One parameter: " + p);
- It is not mandatory to use parentheses, if the type of that variable can be inferred from the context

- Multiple parameters :
- (p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

# LAMBDA EXPRESSIONS

❖The body of a lambda expression can contain zero, one or more statements.

❖When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.

❖When there are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

# NESTED CLASSES

❖In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java.

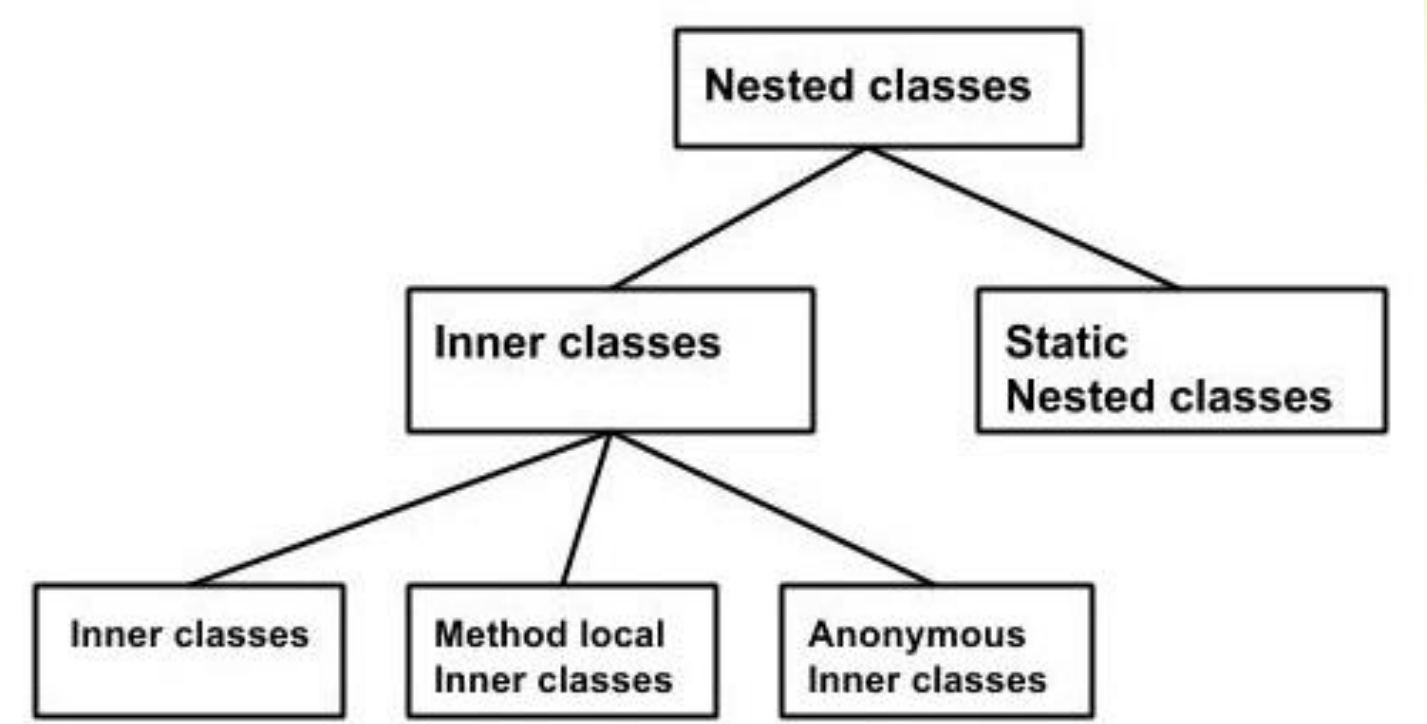
❖The class written within is called the nested class, and the class that holds the inner class is called the outer class.

❖Nested classes are divided into two types:

❖**Non-static nested classes** − These are the non-static members of a class.

❖**Static nested classes** − These are the static members of a class.

# NESTED CLASSES



# STATIC NESTED CLASS

- **static nested class**
  - If the nested class is static, then it's called static nested class. Static nested classes can access only static members of the outer class.
  - Static nested class is same as any other top-level class and is nested for only packaging convenience.

```
class MyOuter {
   static class Nested_Demo {
   }
}
```

# INNER CLASSES

❖Inner classes are a security mechanism in Java.

❖We know a class cannot be associated with the access modifier private, but if we have the class as a member of other class, then the inner class can be made private.

❖And this is also used to access the private members of a class.

❖Inner classes are of three types depending on how and where you define them. They are −

❖**Inner Class**

❖**Method-local Inner Class**

❖**Anonymous Inner Class**

# INNER CLASSES

```java
class Outer_Demo {
   int num;

   // inner class
   private class Inner_Demo {
      public void print() {
         System.out.println("This is an inner class");
      }
   }

   // Accessing the inner class from the method within
   void display_Inner() {
      Inner_Demo inner = new Inner_Demo();
      inner.print();
   }
}

public class My_class {

   public static void main(String args[]) {
      // Instantiating the outer class
      Outer_Demo outer = new Outer_Demo();

      // Accessing the display_Inner() method.
      outer.display_Inner();
   }
}
```

# INNER CLASSES

❖Here you can observe that Outer_Demo is the outer class, Inner_Demo is the inner class, display_Inner() is the method inside which we are instantiating the inner class, and this method is invoked from the main method.

# ACCESSING THE PRIVATE MEMBERS

❖inner classes are also used to access the private members of a class.

❖Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, getValue(), and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.

# ACCESSING THE PRIVATE MEMBERS

❖To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

```
Outer_Demo outer = new Outer_Demo();

Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
```

# METHOD-LOCAL INNER CLASS

❖In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

❖A method-local inner class can be instantiated only within the method where the inner class is defined.

# Anonymous Inner Class

❖An inner class declared without a class name is known as an anonymous inner class.

❖In case of anonymous inner classes, we declare and instantiate them at the same time.

❖Generally, they are used whenever you need to override the method of a class or an interface.

```
AnonymousInner an_inner = new AnonymousInner() {
   public void my_method() {
      ........
      ........
   }
};
```

# ANNOTATIONS

❖Java annotations are used to provide meta data for your Java code. Being meta data, Java annotations do not directly affect the execution of your code, although some types of annotations can actually be used for that purpose.

❖Java annotations were added to Java from Java 5.

❖Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

❖Annotations in java are used to provide additional information, so it is an alternative option for XML and java marker interfaces.

# ANNOTATIONS

- There are several built-in annotations in java. Some annotations are applied to java code and some to other annotations.

  - Built-In Java Annotations used in java code
  - @Override
  - @SuppressWarnings
  - @Deprecated

- Built-In Java Annotations used in other annotations
  - @Target
  - @Retention
  - @Inherited
  - @Documented

# @Override

❖@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

```
public class MySuperClass {

    public void doTheThing() {
        System.out.println("Do the thing");
    }
}


public class MySubClass extends MySuperClass{

    @Override
    public void doTheThing() {
        System.out.println("Do it differently");
    }
}
```

# @SuppressWarnings

❖@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```
@SuppressWarnings
public void methodWithWarning() {


}
```

# @Deprecated

❖@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

```
@Deprecated
public class MyComponent {

}
```

# GENERICS

❖Generics are used to check the type compatibility at the compile time and hence removing the chances of occuring ClassCastException at run time.

❖The syntax for defining generic class is as follows:

```
class Class_Name<T1, T2, T3 ... Tn>
{
    //Generic Type or Parameterized type
}
```

❖Where T1, T2, T3 … Tn (T stands for Type) enclosed within angle brackets (<>) are called type parameters and class ‘Class_Name‘ is called generic type or parameterized type.

# GENERICS

```
class GenericClass<T>
{
    T t;

    public GenericClass(T t)
    {
        this.t = t;
    }

    public void setT(T t)
    {
        this.t = t;
    }

    public T getT()
    {
        return t;
    }
}
```

# GENERICS

❖While creating an instance to the above generic class, you can pass any class type as a type parameter and that class type replaces generic ‘T’ for that object.

❖For example, if you pass String type as a type parameter then String will be the type of variable ‘t’. If you pass Integer as type parameter than Integer will be the type of variable ‘t’.

# GENERICS

❖In the other words, when you pass a type while creating an object to the generic class, that object works only with that type. For example, If you pass

❖String type while creating an object to the above generic class then that object works only with String type. That means setT() method takes String type as an argument and getT() method returns String type.

❖ If you pass any other type to setT() method, it gives compile time error. Hence, strictly checking type casting during compilation.

# GENERICS

❖While creating an instance of generic class, you must pass only derived types. You can’t pass primitive types. If you pass primitive type, it gives compile time error. i.e generics works only with derived type.

```
public class GenericsInJava
{
    public static void main(String[] args)
    {
        GenericClass<int> gen1 = new GenericClass<int>(123);   //Error, can't use primitive

        GenericClass<float> gen2 = new GenericClass<float>(23.56);  //Error, can't use prim
    }
}
```

# GENERICS

❖Objects of same generic class differ depending upon their type parameters. For example, object of above generic class created using String type is not compatible with an object of same class created using Integer type.

```
public class GenericsInJava
{
    public static void main(String[] args)
    {
        GenericClass<String> gen1 = new GenericClass<String>("Value Of t");

        GenericClass<Integer> gen2 = new GenericClass<Integer>(new Integer(20));

        gen1 = gen2;        //Error : Type mismatch

        gen2 = gen1;        //Error : Type mismatch
    }
}
```

# GENERICS

❖Generics are very useful and flexible feature of Java. Generics provide safe type casting to your coding. Along with safe type casting, they also give flexibility to your coding.

❖For example, Once you write a class or interface using generics, you can use any type to create objects to them. In simple words, You can make objects to work with any type using generics.

# GENERICS

❖One more addition to generics is Generic Methods. If you don't want whole class or interface to be generic, you want only some part of class as generic, then generic methods will be solution for this.

❖The syntax for defining generic methods is as follows:

```
<type-Parameters> return_type method_name(parameter list)
{

}
```

❖You can observe that type parameters are mentioned just before the return type. It is a rule you must follow while defining generic methods. The remaining parts are same as in normal method.

❖Generic methods can be static or non-static. There is no restriction for that. Generic class as well as non-generic class can have generic methods.

# GENERICS

❖In this example, ‘genericMethod()’ is a static generic method with ‘T’ as type parameter. Notice that type parameter is mentioned just before the return type.

```
class NonGenericClass
{
    static <T> void genericMethod(T t1)
    {
        T t2 = t1;

        System.out.println(t2);
    }
}
```

❖While calling above generic method, you can pass any type as an argument. This is the best example for generics providing the flexibility.

# GENERICS

```
public class GenericsInJava
{
    public static void main(String[] args)
    {
        NonGenericClass.genericMethod(new Integer(123));      //Passing Integer type as an a

        NonGenericClass.genericMethod("I am string");         //Passing String type as an ar

        NonGenericClass.genericMethod(new Double(25.89));     //Passing Double type as an ar
    }
}
```

# THANK YOU


http://www.4kitsolutions.com/

By
Sudha Agarwal
sudha.agarwal@4kitsolutions.com