Fundamentals of Java

Session: 11

Interfaces and Nested Classes

Objectives

- Describe Interface

- Explain the purpose of interfaces

- Explain implementation of multiple interfaces

- Describe Abstraction

- Explain Nested class

- Explain Member class

- Explain Local class

- Explain Anonymous class

- Describe Static nested class

Introduction

- Java does not support multiple inheritance.

- However, there are several cases when it becomes mandatory for an object to inherit properties from multiple classes to avoid redundancy and complexity in code.

- For this purpose, Java provides a workaround in the form of interfaces.

- Also, Java provides the concept of nested classes to make certain types of programs easy to manage, more secure, and less complex.

An interface in Java is a contract that specifies the standards to be followed by the types that implement it.

◆ The classes that accept the contract must abide by it.

◆ An interface and a class are similar in the following ways:
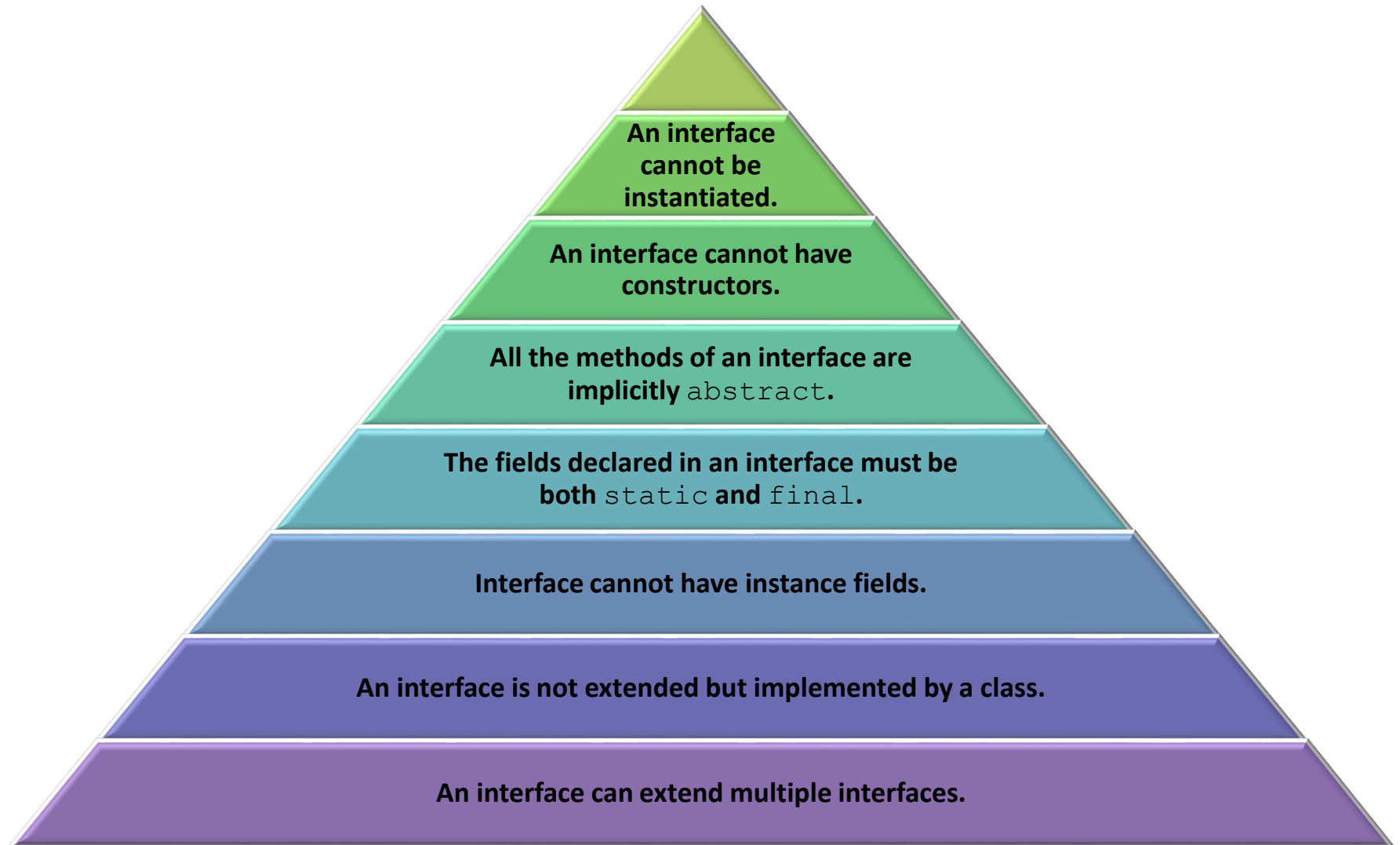
An interface can contain multiple methods.

An interface is saved with a .java extension and the name of the file must match with the name of the interface just as a Java class.

The bytecode of an interface is also saved in a .class file.

Interfaces are stored in packages and the bytecode file is stored in a directory structure that matches the package name.

- ◆ An interface and a class differ in several ways as follows:

An interface cannot be instantiated.

An interface cannot have constructors.

All the methods of an interface are implicitly `abstract`.

The fields declared in an interface must be both `static` and `final`.

Interface cannot have instance fields.

An interface is not extended but implemented by a class.

An interface can extend multiple interfaces.

Purpose of Interfaces 1-11

In several situations, it becomes necessary for different groups of developers to agree to a 'contract' that specifies how their software interacts.

Each group should have the liberty to write their code in their desired manner without having the knowledge of how the other groups are writing their code.

Java interfaces can be used for defining such contracts.

Interfaces do not belong to any class hierarchy, even though they work in conjunction with classes.

Java does not permit multiple inheritance for which interfaces provide an alternative.

In Java, a class can inherit from only one class but it can implement multiple interfaces.

Therefore, objects of a class can have multiple types such as the type of their own class as well as the types of the interfaces that the class implements.

◆ The syntax for declaring an interface is as follows:

Syntax

```
<visibility> interface <interface-name> extends <other-interfaces, … >
{
// declare constants
// declare abstract methods
}
```

where,

<visibility>: Indicates the access rights to the interface. Visibility of an interface is always public.

<interface-name>: Indicates the name of the interface.

<other-interfaces>: List of interfaces that the current interface inherits from.

◆ For example,

```
public interface Sample extends Interface1{
static final int someInteger;
public void someMethod();
}
```

- Consider the hierarchy of vehicles where **`IVehicle`** is the interface that declares the methods which the implementing classes such as **`TwoWheeler`**, **`FourWheeler`**, and so on can define.

- To create a new interface in NetBeans IDE, right-click the package name and select **New → Java** Interface as shown in the following figure:
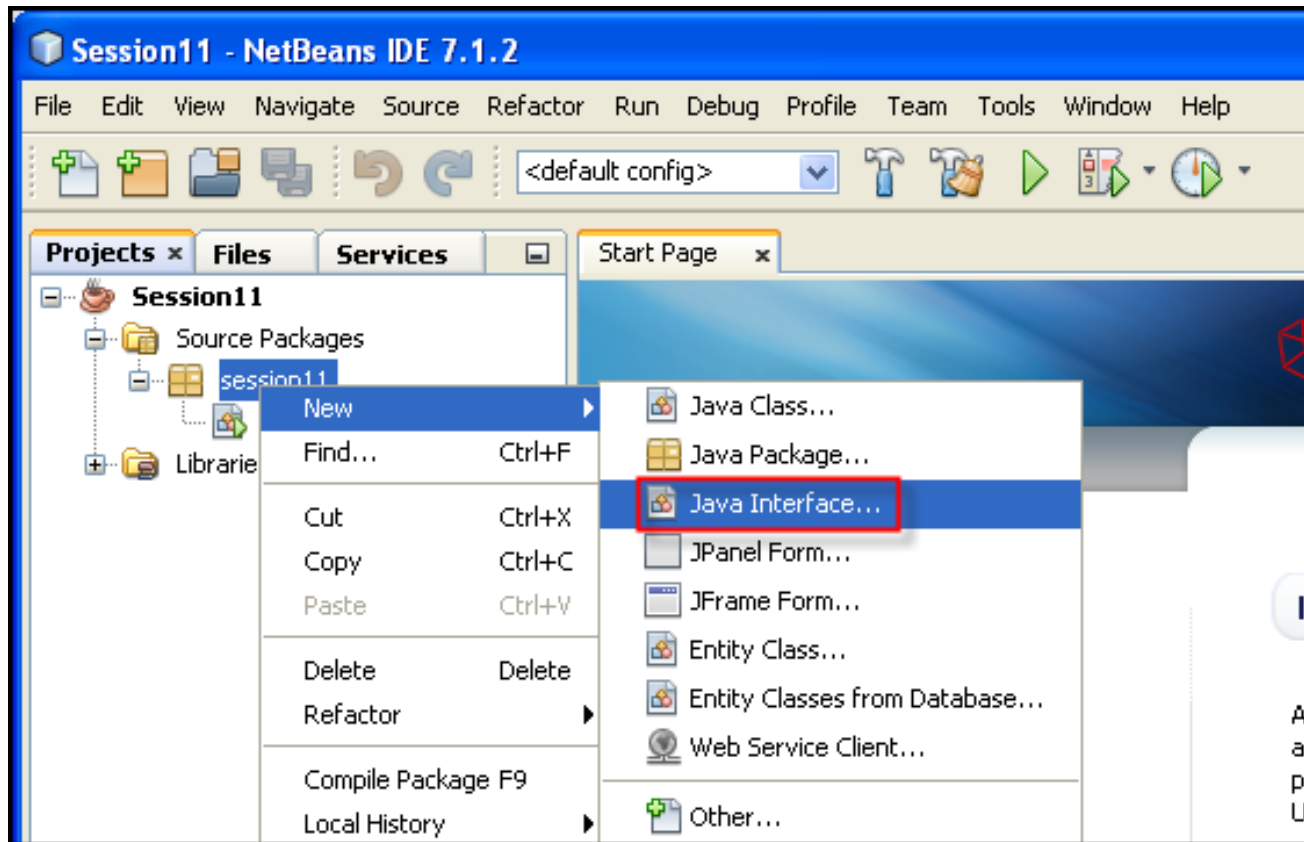
- A dialog box appears where the user must provide a name for the interface and then, click **OK**. This will create an interface with the specified name.

- Following code snippet defines the interface, **IVehicle**:

```
package session11;
public interface IVehicle {

  // Declare and initialize constant
  static final String STATEID="LA-09"; // variable to store state ID

  /**
   * Abstract method to start a vehicle
   * @return void
   */
  public void start();

  /**
   * Abstract method to accelerate a vehicle
   * @param speed an integer variable storing speed
   * @return void
   */
  public void accelerate(int speed);
```

```
   /**
    * Abstract method to apply a brake
    * @return void
    */
  public void brake();


  /**
    * Abstract method to stop a vehicle
    * @return void
    */
  public void stop();
}
```

◆ The syntax to implement an interface is as follows:

Syntax

```
class <class-name> implements <Interface1>,…
{
  // class members
  // overridden abstract methods of the interface(s)
}
```

- Following code snippet defines the class **TwoWheeler** that implements the **IVehicle** interface:

```java
package session11;
class TwoWheeler implements IVehicle {

  String ID; // variable to store vehicle ID
  String type; // variable to store vehicle type

  /**
  * Parameterized constructor to initialize values based on user input
  *
  * @param ID a String variable storing vehicle ID
  * @param type a String variable storing vehicle type
  */
  public TwoWheeler(String ID, String type){
    this.ID = ID;
    this.type = type;
  }

  /**
   * Overridden method, starts a vehicle
   *
```

```
 * @return void
 */
@Override
public void start() {
  System.out.println("Starting the "+ type);
}


/**
 * Overridden method, accelerates a vehicle
 * @param speed an integer storing the speed
 * @return void
 */
@Override
public void accelerate(int speed) {
  System.out.println("Accelerating at speed:"+speed+ " kmph");
}


/**
 * Overridden method, applies brake to a vehicle
 *
 * @return void
```

```java
*/
@Override
public void brake() {
  System.out.println("Applying brakes");
}

/**
 * Overridden method, stops a vehicle
 *
 * @return void
 */
@Override
public void stop() {
  System.out.println("Stopping the "+ type);
}

/**
 * Displays vehicle details
 *
 * @return void
 */
public void displayDetails(){
```

```
      System.out.println("Vehicle No.: "+ STATEID+ " "+ ID);
      System.out.println("Vehicle Type.: "+ type);
   }
}


/**
 * Define the class TestVehicle.java
 *
 */
public class TestVehicle {

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args){

    // Verify the number of command line arguments
    if(args.length==3) {

      // Instantiate the TwoWheeler class
      TwoWheeler objBike = new TwoWheeler(args[0], args[1]);
```
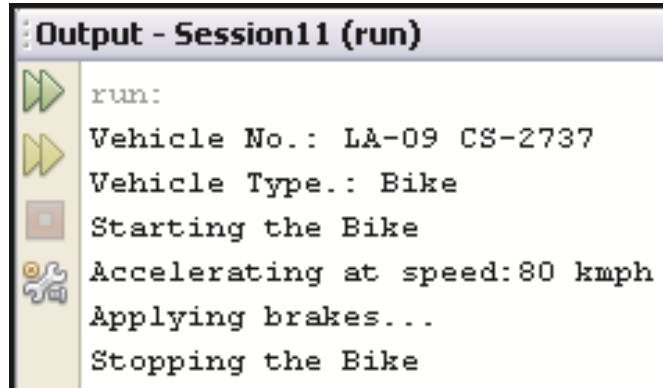
```
        // Invoke the class methods
        objBike.displayDetails();
        objBike.start();
        objBike.accelerate(Integer.parseInt(args[2]));
        objBike.brake();
        objBike.stop();
    }
    else {
        System.out.println("Usage: java TwoWheeler <ID> <Type> <Speed>");
    }
  }
}
```

◆ Following figure shows the output of the code when the user passes **CS-2723 Bike 80** as command line arguments:


```
Output - Session11 (run)
run:
Vehicle No.: LA-09 CS-2737
Vehicle Type.: Bike
Starting the Bike
Accelerating at speed:80 kmph
Applying brakes...
Stopping the Bike
```

- Java does not support multiple inheritance of classes but allows implementing multiple interfaces to simulate multiple inheritance.

- To implement multiple interfaces, write the interface names after the implements keyword separated by a comma.

- For example,

```
public class Sample implements Interface1, Interaface2{
}
```

- Following code snippet defines the interface **IManufacturer**:

```
package session11;
public interface IManufacturer {

  /**
   * Abstract method to add contact details
   * @param detail a String variable storing manufacturer detail
   * @return void
   */
  public void addContact(String detail);

  /**
   * Abstract method to call the manufacturer
```

```
 * @param phone a String variable storing phone number
 * @return void
 */
public void callManufacturer(String phone);

/**
 * Abstract method to make payment
 * @param amount a float variable storing amount
 * @return void
 */
public void makePayment(float amount);
}
```

- The modified class, **TwoWheeler** implementing both the **IVehicle** and **IManufacturer** interfaces is displayed in the following code snippet:

```
package session11;
class TwoWheeler implements IVehicle, IManufacturer {

    String ID; // variable to store vehicle ID
    String type; // variable to store vehicle type
```

```
/**
 * Parameterized constructor to initialize values based on user input
 *
 * @param ID a String variable storing vehicle ID
 * @param type a String variable storing vehicle type
 */
public TwoWheeler(String ID, String type){
  this.ID = ID;
  this.type = type;
}


/**
 * Overridden method, starts a vehicle
 *
 * @return void
 */
@Override
public void start() {
  System.out.println("Starting the "+ type);
}
```

```
/**
 * Overridden method, accelerates a vehicle
 * @param speed an integer storing the speed
 * @return void
 */
@Override
public void accelerate(int speed) {

  System.out.println("Accelerating at speed:"+speed+ " kmph");
}


/**
 * Overridden method, applies brake to a vehicle
 *
 * @return void
 */
@Override
public void brake() {

  System.out.println("Applying brakes...");
}
```

```
/**
 * Overridden method, stops a vehicle
 *
 * @return void
 */
@Override
public void stop() {

  System.out.println("Stopping the "+ type);

}


/**
 * Displays vehicle details
 *
 * @return void
 */
public void displayDetails()
{
  System.out.println("Vehicle No.: "+ STATEID+ " "+ ID);
  System.out.println("Vehicle Type.: "+ type);
}
```

```
// Implement the IManufacturer interface methods

/**
 * Overridden method, adds manufacturer details
 * @param detail a String variable storing manufacturer detail
 * @return void
 */
@Override
public void addContact(String detail) {
  System.out.println("Manufacturer: "+detail);
}

/**
 * Overridden method, calls the manufacturer
 * @param phone a String variable storing phone number
 * @return void
 */
@Override
public void callManufacturer(String phone) {
  System.out.println("Calling Manufacturer @: "+phone);
}
```

```
    /**
     * Overridden method, makes payment
     * @param amount a String variable storing the amount
     * @return void
     */
    @Override
    public void makePayment(float amount) {
      System.out.println("Payable Amount: $"+amount);
    }
}

/**
 * Define the class TestVehicle.java
 *
 */
public class TestVehicle {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args){
      // Verify number of command line arguments
      if(args.length==6) {
```
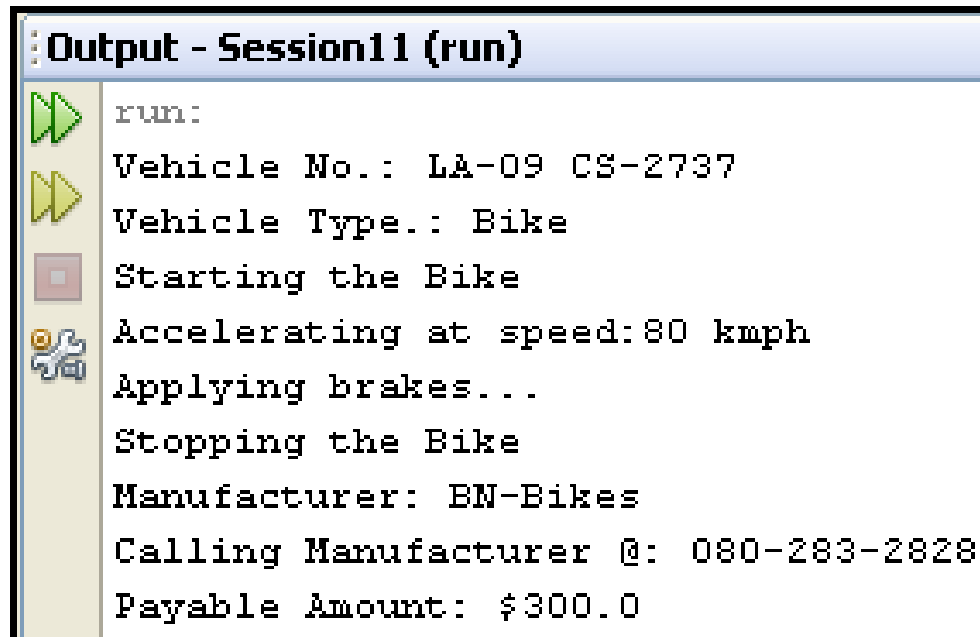
```
            // Instantiate the class
            TwoWheeler objBike = new TwoWheeler(args[0], args[1]);
            objBike.displayDetails();
            objBike.start();
            objBike.accelerate(Integer.parseInt(args[2]));
            objBike.brake();
            objBike.stop();
            objBike.addContact(args[3]);
            objBike.callManufacturer(args[4]);
            objBike.makePayment(Float.parseFloat(args[5]));
        }
        else{

            // Display an error message
            System.out.println("Usage: java TwoWheeler <ID> <Type> <Speed>
             <Manufacturer> <Phone> <Amount>");
        }
    }
}
```

- The class **TwoWheeler** now implements both the interfaces; **IVehicle** and **IManufacturer** as well as all the methods of both the interfaces.

- Following figure shows the output of the modified code.

- The user passes **CS-2737 Bike 80 BN-Bikes 808-283-2828 300** as command line arguments.

```
Output - Session11 (run)
run:
Vehicle No.: LA-09 CS-2737
Vehicle Type.: Bike
Starting the Bike
Accelerating at speed:80 kmph
Applying brakes...
Stopping the Bike
Manufacturer: BN-Bikes
Calling Manufacturer @: 080-283-2828
Payable Amount: $300.0
```

- The interface IManufacturer can also be implemented by other classes such as **FourWheeler**, **Furniture**, **Jewelry**, and so on, that require manufacturer information.

Understanding the Concept of Abstraction 1-3

Abstraction is defined as the process of hiding the unnecessary details and revealing only the essential features of an object to the user.

Abstraction is a concept that is used by classes that consist of attributes and methods that perform operations on these attributes.

Abstraction can also be achieved through composition. For example, a class Vehicle is composed of an engine, tyres, ignition key, and many other components.

In Java, `abstract` classes and interfaces are used to implement the concept of abstraction.

To use an `abstract` class or interface one needs to extend or implement `abstract` methods with concrete behavior.

Abstraction is used to define an object based on its attributes, functionality, and interface.

◆ The differences between an abstract class and an interface are listed in the following table:

Abstract Class	Interface
An `abstract` class can have `abstract` as well as concrete methods that are methods with a body.	An interface can have only `abstract` methods.
An `abstract` class may have non-final variables.	Variables declared in an interface are implicitly `final`.
An `abstract` class may have members with different access specifiers such as `private`, `protected`, and so on.	Members of an interface are `public` by default.
An `abstract` class is inherited using the `extends` keyword.	An interface is implemented using the `implements` keyword.
An `abstract` class can inherit from another class and implement multiple interfaces.	An interface can extend from one or more interfaces.

◆ Java allows defining a class within another class.

◆ Such a class is called a nested class as shown in the following figure:
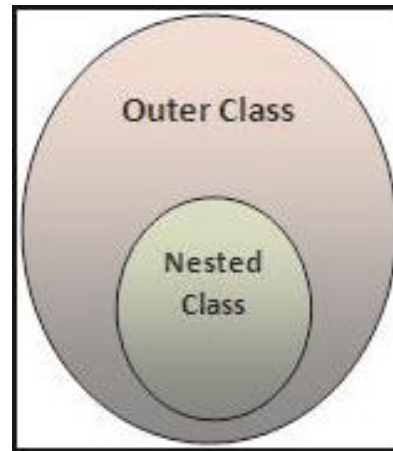
◆ Following code snippet defines a nested class:

```
class Outer{
   ...
   class Nested{
     ...
   }
}
```

◆ The class **Outer** is the external enclosing class and the class **Nested** is the class defined within the class **Outer**.
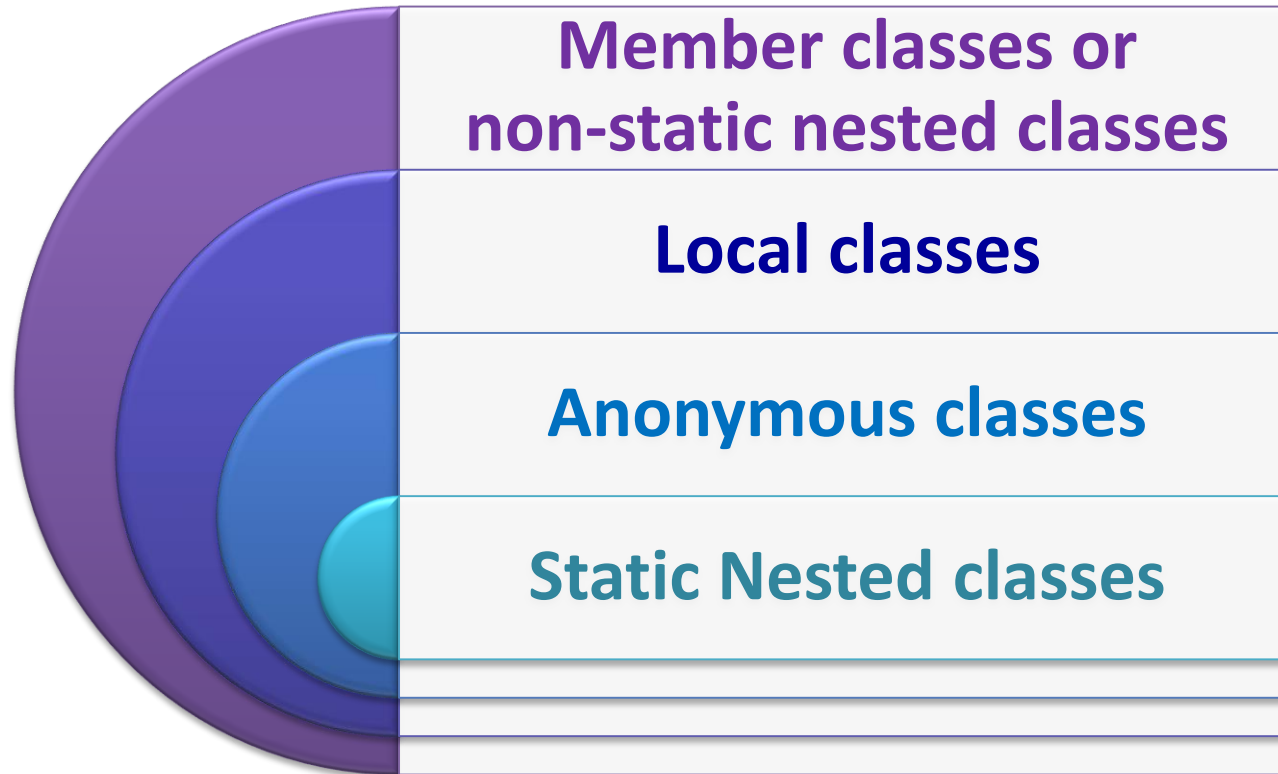
- Nested classes are classified as static and non-static.

- Nested classes that are declared `static` are simply termed as `static` nested classes whereas non-static nested classes are termed as inner classes.

- This has been demonstrated in the following code snippet:

```
class Outer{
  ...
  static class StaticNested{
    ...
  }
  class Inner{
  ...
  }
}
```

- The class **StaticNested** is a nested class that has been declared as `static` whereas the non-static nested class, **Inner**, is declared without the keyword `static`.

- A nested class is a member of its enclosing class.

- Non-static nested classes or inner classes can access the members of the enclosing class even when they are declared as private.

- `Static` nested classes cannot access any other member of the enclosing class.

◆ The different types of nested classes are as follows:

**Member classes or
non-static nested classes**

Local classes

Anonymous classes

Static Nested classes

A member class is a non-static inner class.

It is declared as a member of the outer or enclosing class.

The member class cannot have `static` modifier since it is associated with instances of the outer class.

An inner class can directly access all members that is, fields and methods of the outer class including the private ones.

However, the outer class cannot access the members of the inner class directly even if they are declared as `public`.

This is because members of an inner class are declared within the scope of inner class.

An inner class can be declared as `public`, `private`, `protected`, `abstract`, or `final`.

Instances of an inner class exist within an instance of the outer class.

To instantiate an inner class, one must create an instance of the outer class.

- One can access the inner class object within the outer class object using the statement defined in the following code snippet:

```
// accessing inner class using outer class object
Outer.Inner objInner = objOuter.new Inner();
```

- Following code snippet describes an example of non-static inner class:

```java
package session11;
class Server {
  String port; // variable to store port number

  /**
   * Connects to specified server
   *
   * @param IP a String variable storing IP address of server
   * @param port a String variable storing port number of server
   * @return void
   */
  public void connectServer(String IP, String port) {

  System.out.println("Connecting to Server at:"+ IP + ":" + port);
}
```

```java
    /**
     * Define an inner class
     *
     */
    class IPAddress
    {
        /**
         * Returns the IP address of a server
         *
         * @return String
         */
        String getIP() {
            return "101.232.28.12";
        }
    }
}


/**
 * Define the class TestConnection.java
 *
 */
public class TestConnection {
```
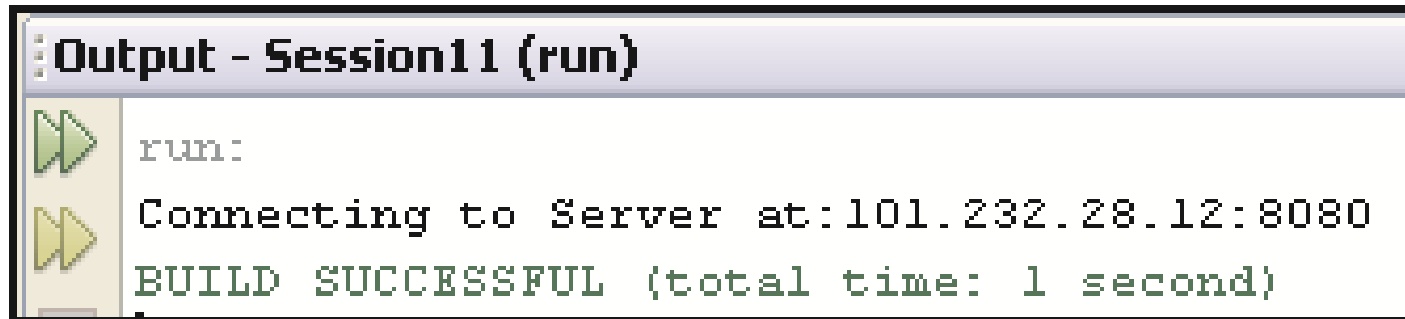
```
/**
 * @param args the command line arguments
 */
public static void main(String[] args){
/**
 * @param args the command line arguments
 */
public static void main(String[] args)
{
  // Check the number of command line arguments
  if(args.length==1) {
    // Instantiate the outer class
    Server objServer1 = new Server();
    // Instantiate the inner class using outer class object
    Server.IPAddress objIP = objServer1.new IPAddress();
    // Invoke the connectServer() method with the IP returned from
    // the getIP() method of the inner class
    objServer1.connectServer(objIP.getIP(),args[0]);
  }
  else {
    System.out.println("Usage: java Server <port-no>"); }
}}
```

◆ The class **Server** is an outer class that consists of a variable port that represents the port at which the server will be connected.

◆ Also, the **connectServer(String, String)** method accepts the IP address and port number as a parameter.

◆ The inner class **IPAddress** consists of the **getIP()** method that returns the IP address of the server.

◆ Following figure shows the output of the code when user provides '**8080**' as the port number at command line:

```
Output - Session11 (run)

 run:

 Connecting to Server at:101.232.28.12:8080

 BUILD SUCCESSFUL (total time: 1 second)
```

Summary

- An interface in Java is a contract that specifies the standards to be followed by the types that implement it.

- To implement multiple interfaces, write the interfaces after the implements keyword separated by a comma.

- Abstraction, in Java, is defined as the process of hiding the unnecessary details and revealing only the necessary details of an object.

- Java allows defining a class within another class. Such a class is called a nested class.

- A member class is a non-static inner class. It is declared as a member of the outer or enclosing class.

- An inner class defined within a code block such as the body of a method, constructor, or an initializer, is termed as a local inner class.

- An inner class declared without a name within a code block such as the body of a method is called an anonymous inner class.

- A static nested class cannot directly refer to instance variables or methods of the outer class just like static methods but only through an object reference.