

# Learning Java - A Foundational Journey

## Session: 8

### Interfaces and Nested Classes





- ◆ Describe interfaces
- ◆ Explain the purpose of interfaces
- ◆ Explain implementation of multiple interfaces
- ◆ Describe private methods in interfaces
- ◆ Describe Abstraction
- ◆ Explain Nested class
- ◆ Explain Member class
- ◆ Explain Local class
- ◆ Explain Anonymous class
- ◆ Describe Static nested class



- ◆ 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.

For Apteck Centre Use Only



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

An interface can contain multiple methods.

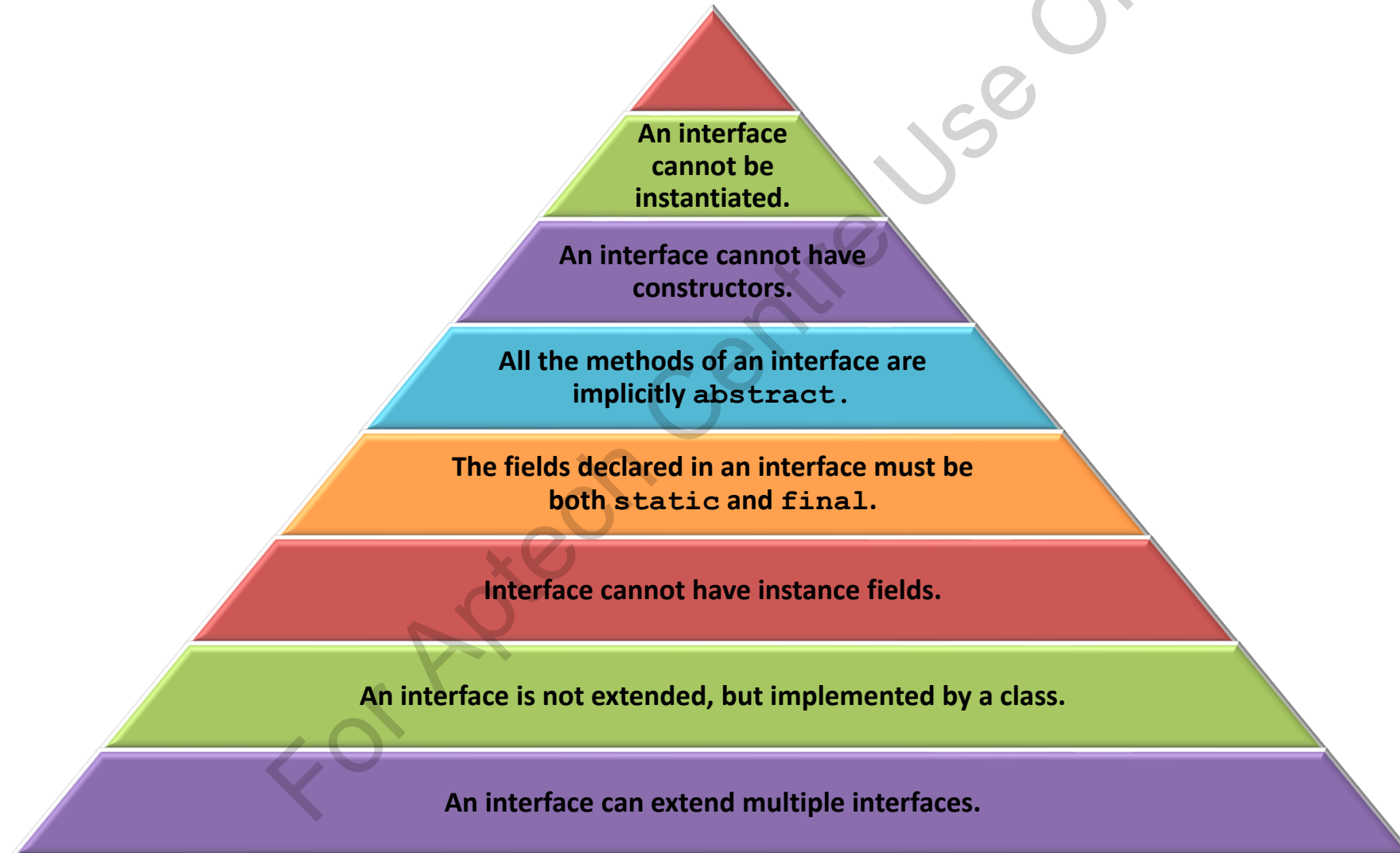
An interface is saved with a **.java** extension and the name of the file must match with the name of the interface just as a Java class.

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:





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.

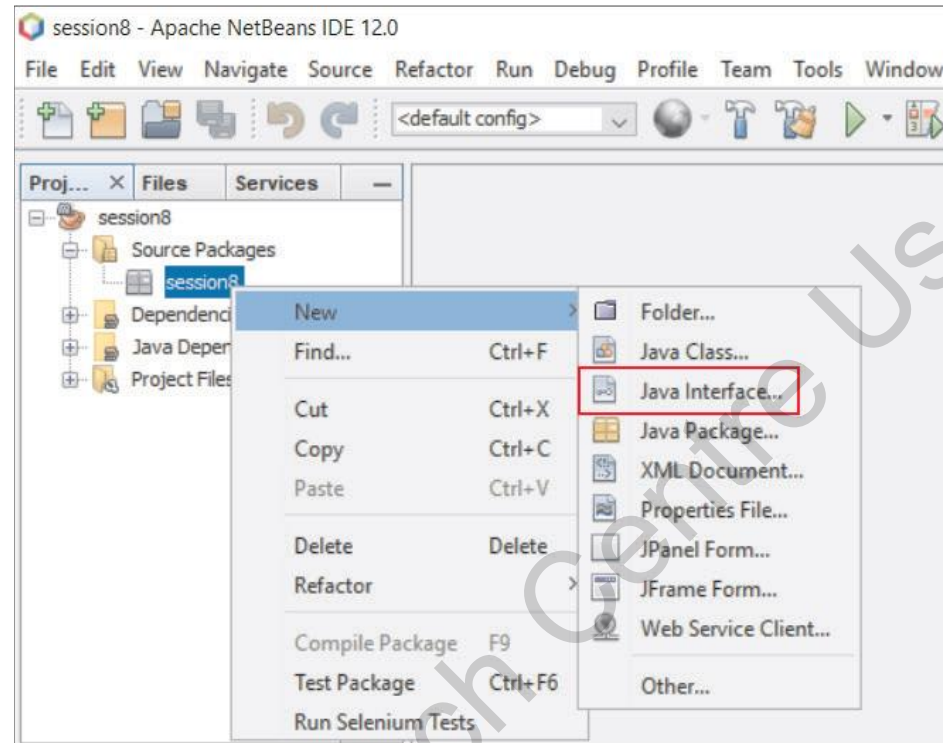
Java does not permit multiple inheritance for which interfaces provide an alternative.

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

## Syntax

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

# Purpose of Interfaces 2-2



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

# Implementing Multiple Interfaces 1-6



- ◆ Java does not support multiple inheritance of classes, but allows implementing multiple interfaces to simulate multiple inheritance.
- ◆ For example,  
`public class Sample implements Interface1, Interface2{ }`
- ◆ Following code snippet defines the interface **IManufacturer**:

```
package session8;
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);
}
```



# Implementing Multiple Interfaces 2-6



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

```
package session8;
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);
    }
}
```

# Implementing Multiple Interfaces 3-6



```
/**
 * 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);
}
```

# Implementing Multiple Interfaces 4-6



```
/**
 * 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);
}
```

# Implementing Multiple Interfaces 5-6



```
/**
 * 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]);
        }
    }
}
```

# Implementing Multiple Interfaces 6-6



```
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.

A screenshot of a Java IDE's console window. It shows the output of a program. The text is as follows:  
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.



Use private and abstract modifiers together

Private non-static methods cannot be used inside private static methods

```
package session8;
interface Person{
    default void display1(String msg){
        msg+=" from display1";
        printMessage(msg);
    }
    default void display2(String msg){
        msg+=" from display2";
        printMessage(msg);
    }
    private void printMessage(String msg){
        System.out.println(msg);
    }
}
public class Employee implements Person {
    public void printInterface(){
        display1("Hello there");
        display2("Hi there");
    }
    public static void main(String[] args){
        Employee objEmployee = new Employee();
        objEmployee.printInterface();
    }
}
```

# Understanding the Concept of Abstraction 1-2



Process of hiding the unnecessary details and revealing only the essential features of an object to the user.

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

# Understanding the Concept of Abstraction 2-2



- ◆ Some of the differences between Abstraction and Encapsulation are as follows:

Abstraction is implemented using an interface in an abstract class whereas Encapsulation is implemented using `private`, `default` or `package-private`, and `protected` access modifier.

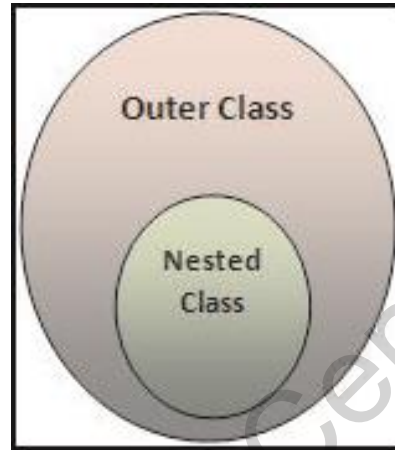
Encapsulation is also referred to as data hiding.

The basis of the design principle 'programming for interface than implementation' is abstraction and that of 'encapsulate whatever changes' is encapsulation.





- ◆ 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{  
        ...  
    }  
}
```

# Benefits of Using Nested Class



## Creates logical grouping of classes

- If a class is of use to only one class, then it can be embedded within that class and the two classes can be kept together.

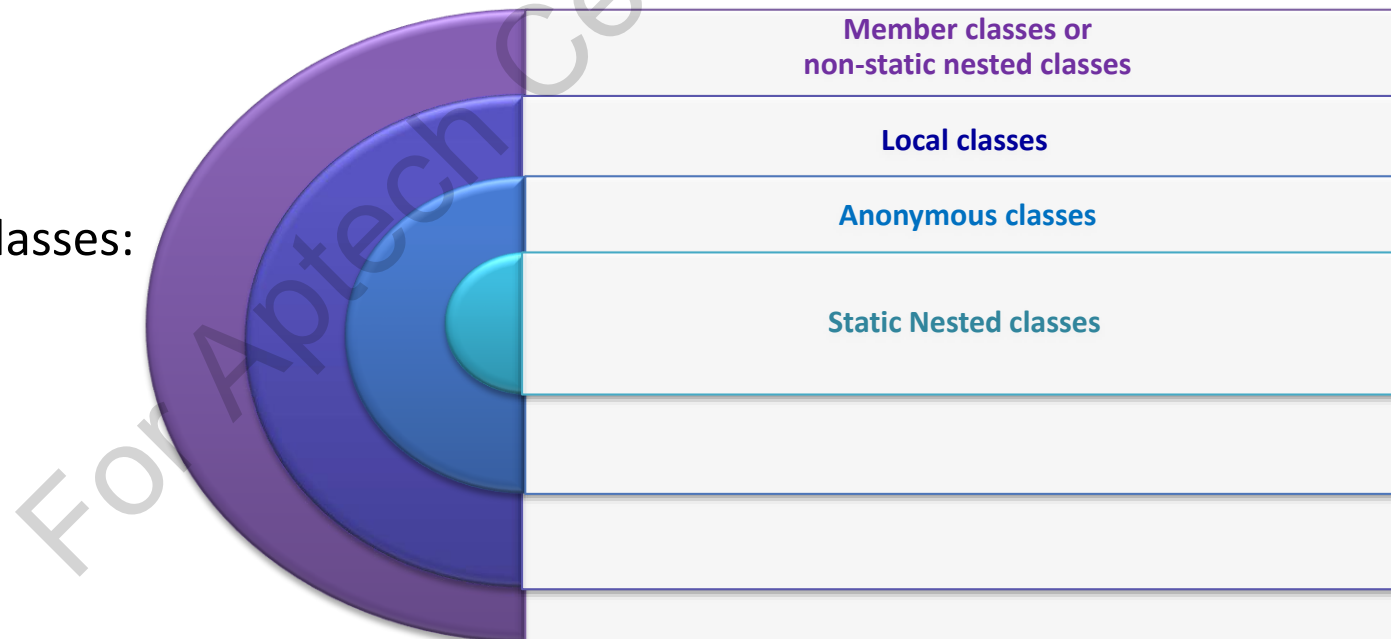
## Increases encapsulation

- In case of two top level classes such as class A and B, when B wants access to members of A that are private, class B can be nested within class A so that B can access the members declared as private.

## Increased readability and maintainability of code

- Nesting of small classes within larger top-level classes helps to place the code closer to where it will be used.

## ◆ Different types of 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.

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.



- ◆ 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:

```
package session8;
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);
    }
    /**
     * 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.

A screenshot of a Java IDE's console window. It shows the output of a program execution. The text is as follows:

```
run:
Connecting to Server at:101.232.28.12:8080
BUILD SUCCESSFUL (total time: 1 second)
```

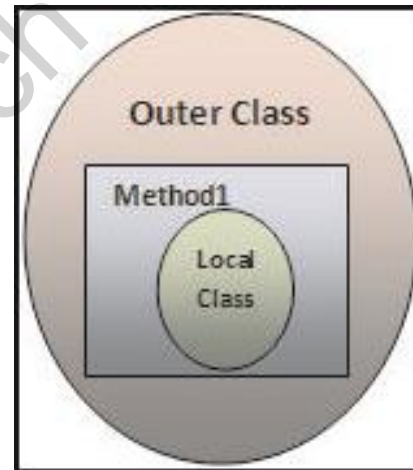
The console window has a standard Windows-style title bar and a vertical scrollbar on the right. The text is in a monospaced font, with the first line in black, the second in black, and the third in green.



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.

The scope of a local inner class is only within that particular block.

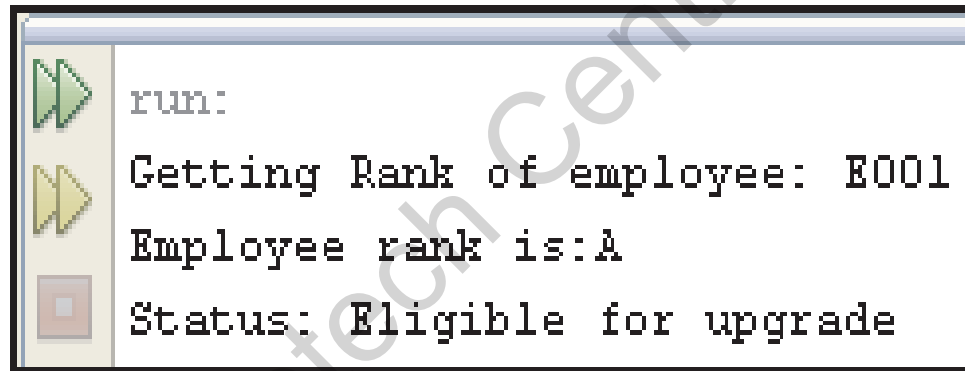
Unlike an inner class, a local inner class is not a member of the outer class and therefore, it cannot have any access specifier.



local inner class

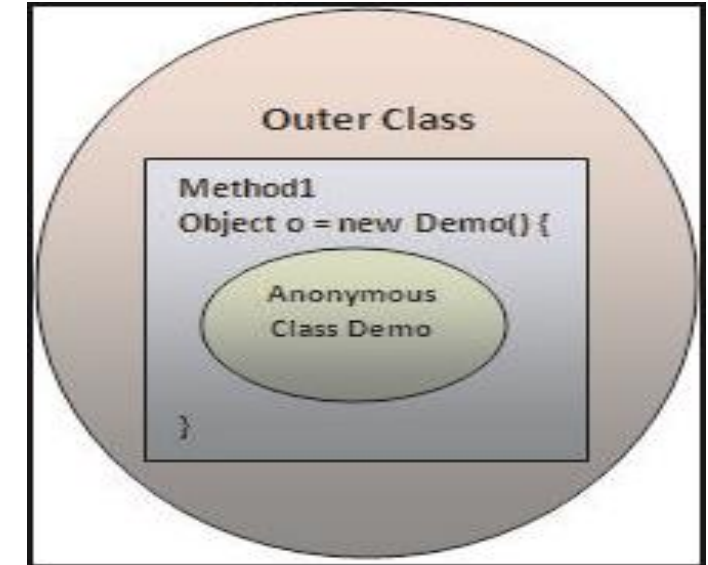
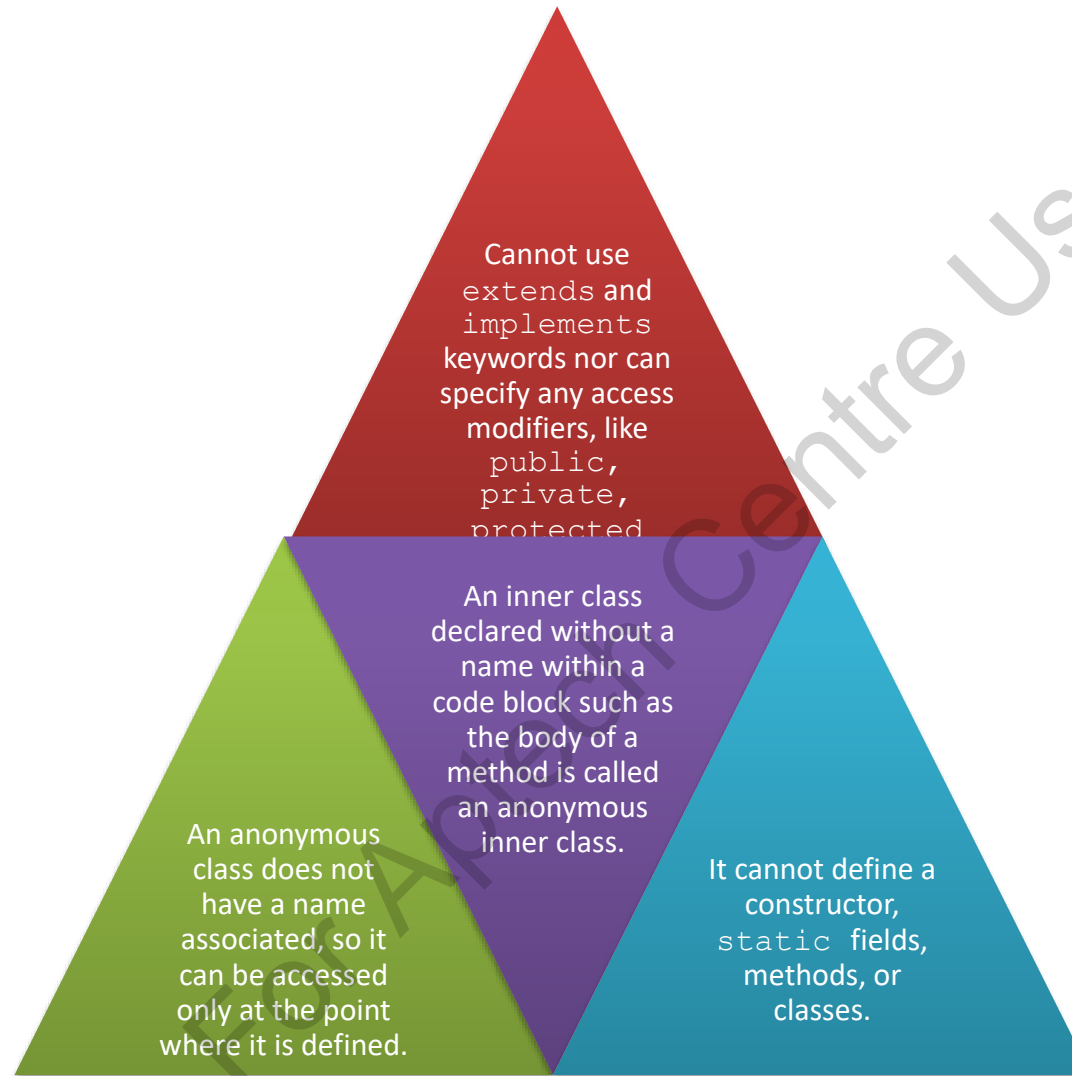


- ◆ Local inner class has the following features:
  - ◆ It is associated with an instance of the enclosing class.
  - ◆ It can access any members, including private members, of the enclosing class.
  - ◆ It can access any local variables, method parameters, or exception parameters that are in the scope of the local method definition, provided that these are declared as final.



```
run:
Getting Rank of employee: E001
Employee rank is:A
Status: Eligible for upgrade
```







- ◆ Following code snippet describes an example of anonymous class:

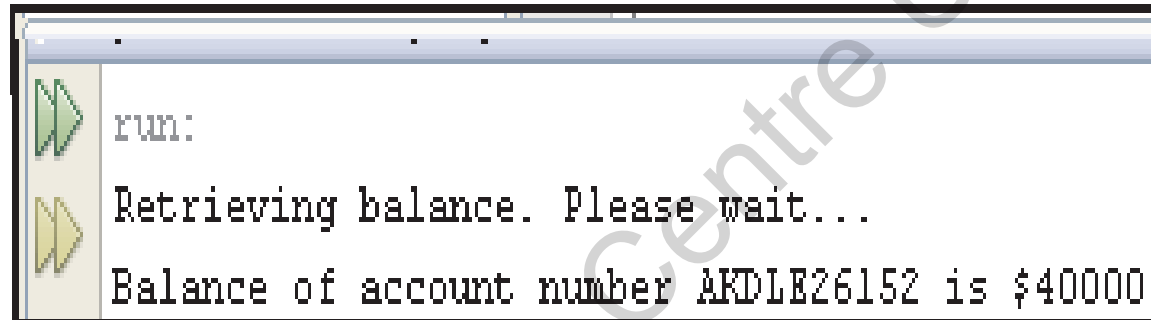
```
package session8;
class Authenticate {
    /**
     * Define an anonymous class
     *
     */
    Account objAcc = new Account() {
        /**
         * Displays balance
         * @param accNo a String variable storing balance
         * @return void
         */
        @Override
        public void displayBalance(String accNo) {
            System.out.println("Retrieving balance. Please wait...");
            // Assume that the server returns 40000
            System.out.println("Balance of account number " + accNo.toUpperCase() + " is $40000");
        }
    }; // End of anonymous class
}
/**
 * Define the Account class
 */
class Account {
    /**
     * Displays balance
     * @param accNo a String variable storing balance
     * @return void
     */
}
```



```
        public void displayBalance(String accNo) {
        }
    }
    /**
     * Define the TestAuthentication class
     */
    public class TestAuthentication {
        /**
         * @param args the command line arguments
         */
        public static void main(String[] args) {
            // Instantiate the Authenticate class
            Authenticate objUser = new Authenticate()
            // Check the number of command line arguments
            if (args.length == 3) {
                if (args[0].equals("admin") && args[1].equals("abc@123")){
                    // Invoke the displayBalance() method
                    objUser.objAcc.displayBalance(args[2]);
                }
                else{
                    System.out.println("Unauthorized user");
                }
            }
            else {
                System.out.println("Usage: java Authenticate <user-name> <password> <account-no>");
            }
        }
    }
}
```

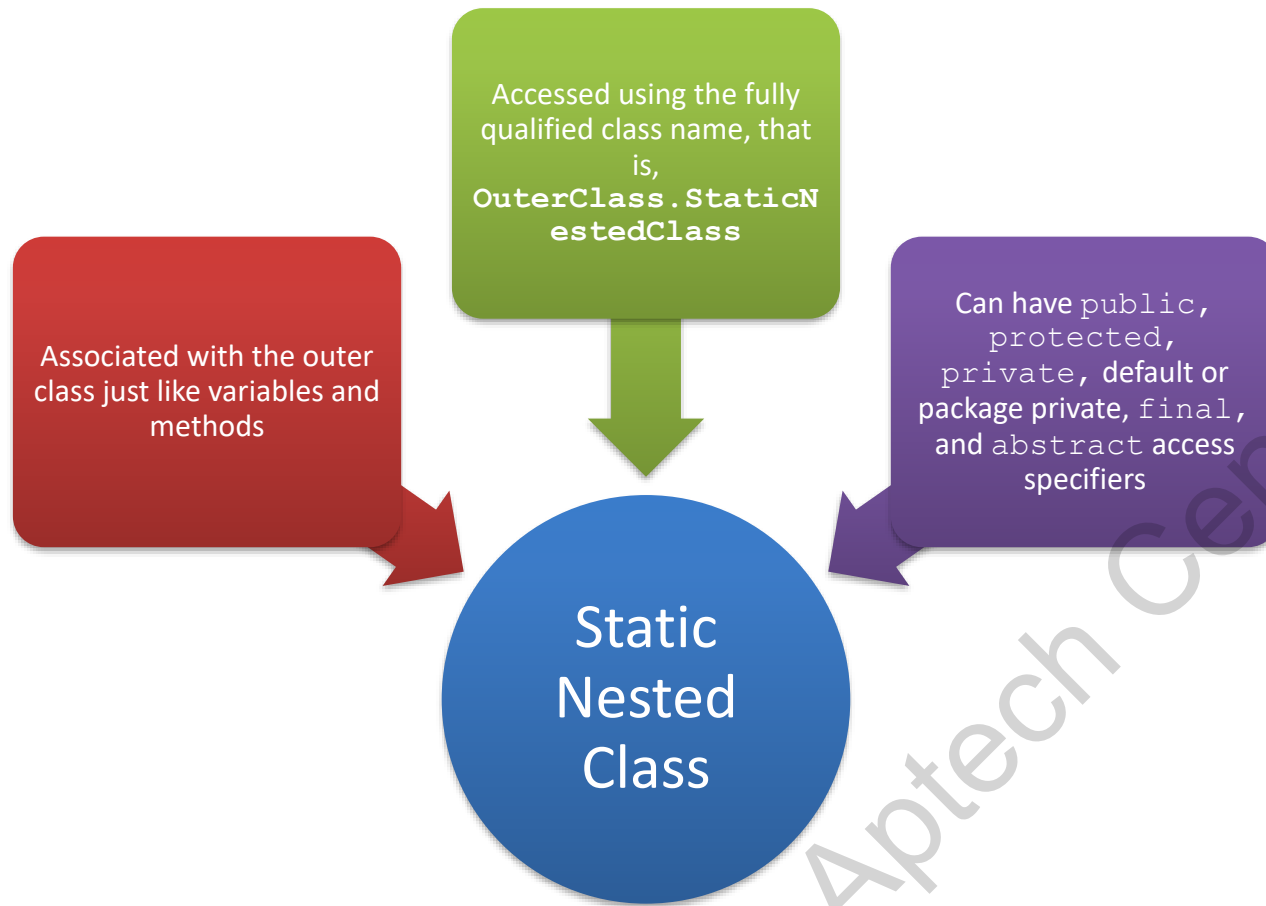


- ◆ The class **Authenticate** consists of an anonymous object of type **Account**.
- ◆ The **displayBalance(String)** method is used to retrieve and display the balance of the specified account number.
- ◆ Following figure shows the output of the code when user passes 'admin', 'abc@123', and 'akdle26152', as arguments:



```
run:  
Retrieving balance. Please wait...  
Balance of account number AKDLE26152 is $40000
```

# Static Nested Class 1-2



```
run:
State Bank of America
Branch: New York
Code: K3983LK3IE
Date-Time:Thu Dec 27 11:50:38 GMT+05:30 2012
Balance of account number AKDLB26152 is $200000
```



- ◆ In the code, the pattern **dd/MM/yyyy HH:mm:ss** uses several symbols that are pattern letters recognized by the `SimpleDateFormat` class.

Pattern Letter	Description
d	Day of the month
M	Month of the year
Y	Year
H	Hour of a day (0-23)
m	Minute of an hour
s	Second of a minute

- ◆ Following figure shows the output of the modified code:

A screenshot of a Java IDE's output window. It shows the following text:

```
run:
State Bank of America
Branch: New York
Code: K3983LKSIE
Date-Time:27/12/2012 11:51:27
Balance of account number AKDLE26152 is $200000
```



- ◆ 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.

For Aptech Centre Use Only