

# Fundamentals of Java

## Session: 11

## Interfaces and Nested Classes





- ◆ 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



- ◆ 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.

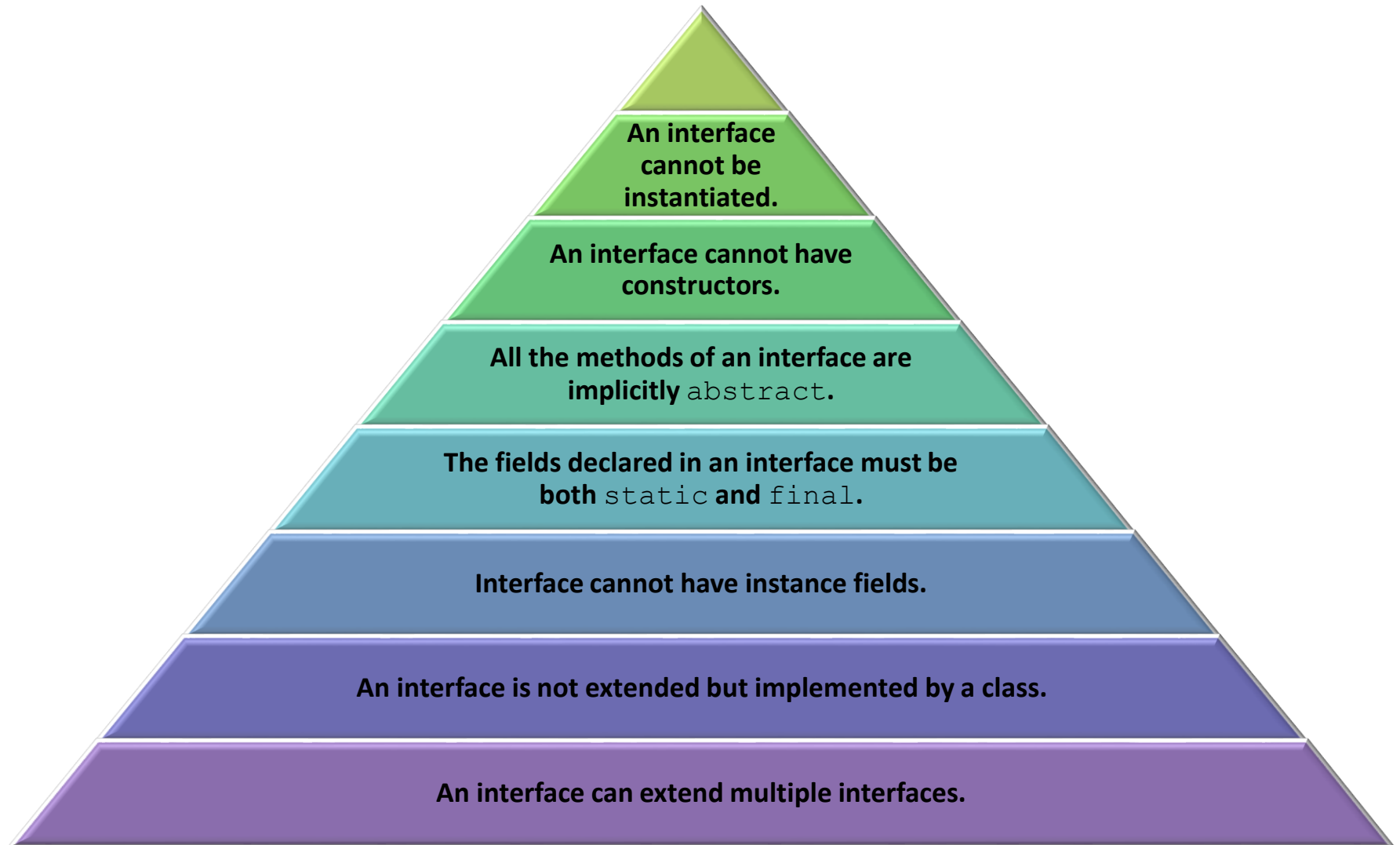
An interface is saved with a .java extension and the name of the file must match with the name of the interface just as a Java class.

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:



# 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.

# Purpose of Interfaces 2-11



- ◆ 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();
}
```

# Purpose of Interfaces 3-11



- ◆ In Java, interface names are written in CamelCase, that is, first letter of each word is capitalized.
- ◆ Also, the name of the interface describes an operation that a class can perform. For example,  

```
interface Enumerable  
interface Comparable
```
- ◆ Some programmers prefix the letter 'I' with the interface name to distinguish interfaces from classes. For example,  

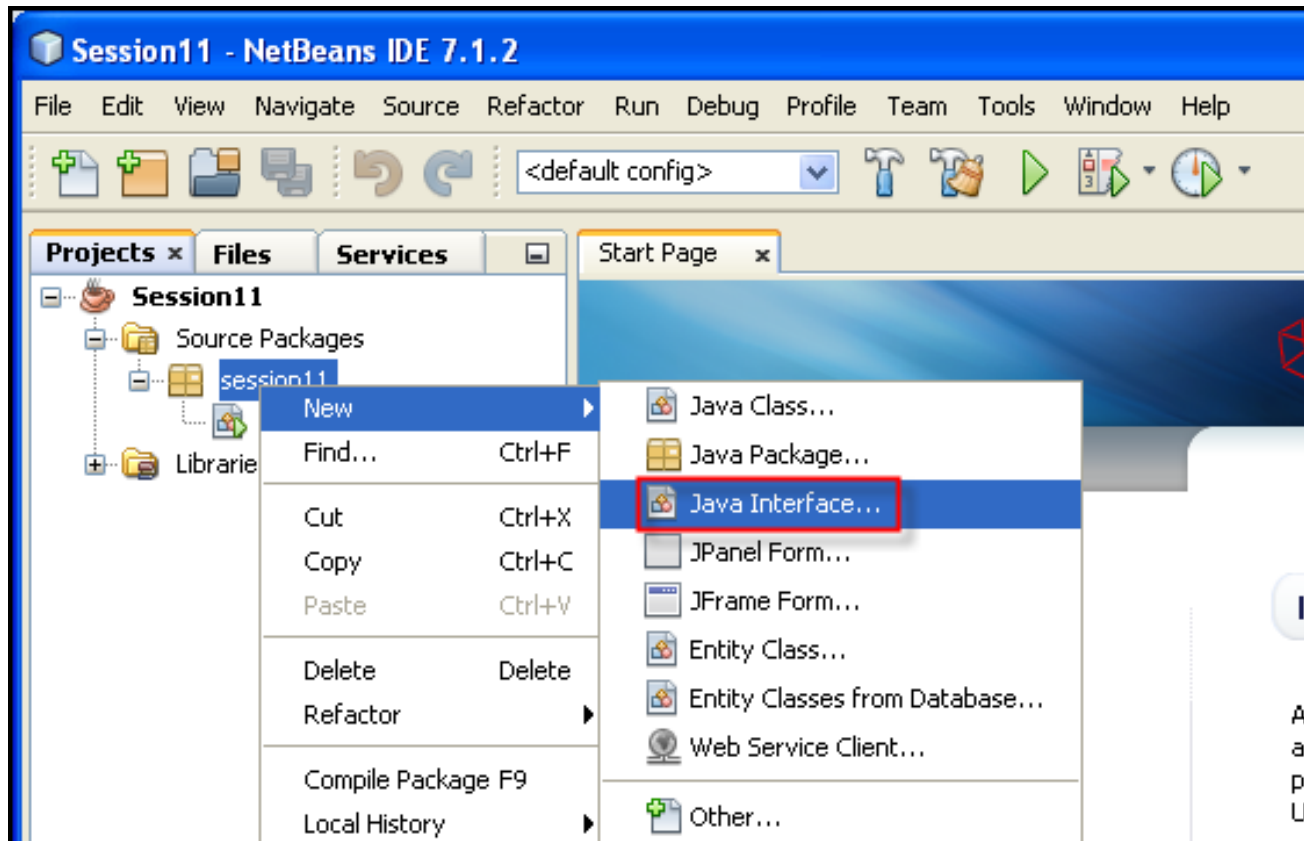
```
interface IEnumerable  
interface Icomparable
```
- ◆ Notice that the method declaration does not have any braces and is terminated with a semicolon.
- ◆ Also, the body of the interface contains only abstract methods and no concrete method.
- ◆ Since all methods in an interface are implicitly `abstract`, the `abstract` keyword is not explicitly specified with the method signature.



# Purpose of Interfaces 4-11



- ◆ 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)
}
```

# Purpose of Interfaces 7-11



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

```
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
     *
     */
}
```

# Purpose of Interfaces 8-11



```
* @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
```

# Purpose of Interfaces 9-11



```
*/
@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(){
```

# Purpose of Interfaces 10-11



```
        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) {

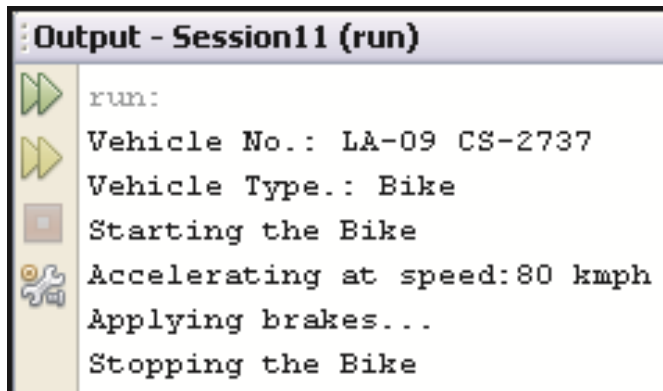
            // Instantiate the TwoWheeler class
            TwoWheeler objBike = new TwoWheeler(args[0], args[1]);
```

# Purpose of Interfaces 11-11



```
// 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
```



# Implementing Multiple Interfaces 1-9



- ♦ 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, Interface2{  
}
```

- ♦ 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
```

# Implementing Multiple Interfaces 2-9



```
* @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
```

# Implementing Multiple Interfaces 3-9



```
/**
 * 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 4-9



```
/**
 * 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...");
}
```

# Implementing Multiple Interfaces 5-9



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

# Implementing Multiple Interfaces 6-9



```
// 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 7-9



```
/**
 * 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) {
```

# Implementing Multiple Interfaces 8-9



```
// 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{

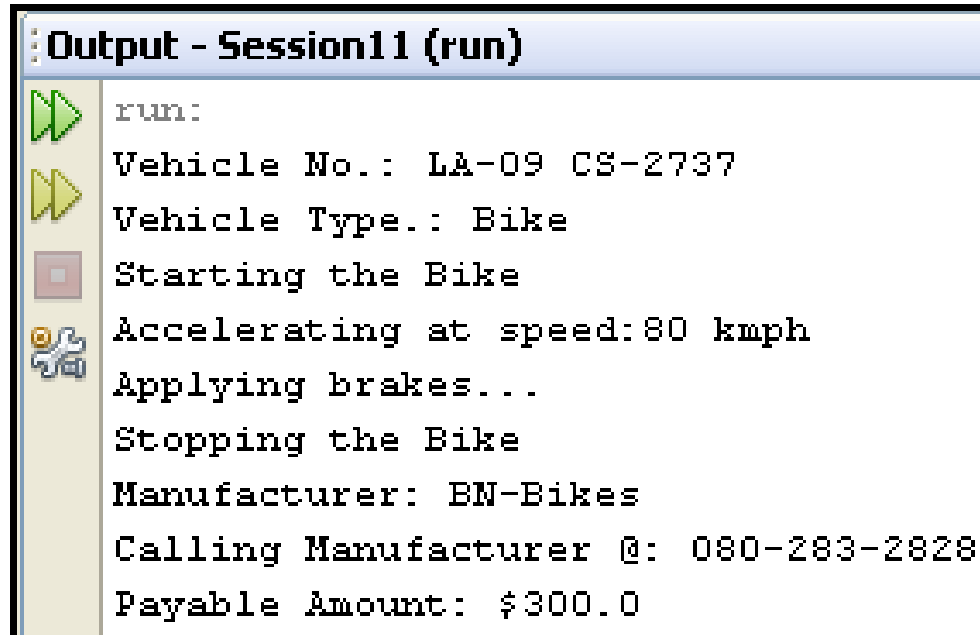
    // Display an error message
    System.out.println("Usage: java TwoWheeler <ID> <Type> <Speed>
        <Manufacturer> <Phone> <Amount>");
}
}
}
```



# Implementing Multiple Interfaces 9-9



- ◆ 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.



```
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.

# Understanding the Concept of Abstraction 2-3



- ◆ 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.

# Understanding the Concept of Abstraction 3-3



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

Abstraction refers to bringing out the behavior from 'How exactly' it is implemented whereas Encapsulation refers to hiding details of implementation from the outside world so as to ensure that any change to a class does not affect the dependent classes.

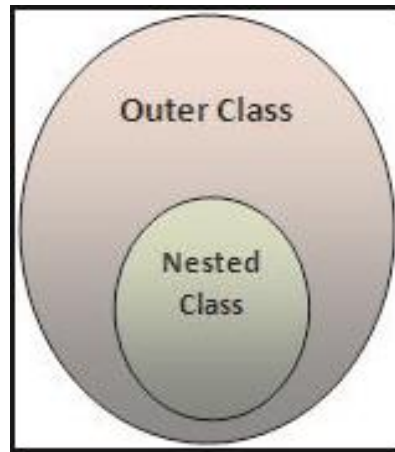
Abstraction is implemented using an interface in an `abstract` class whereas Encapsulation is implemented using `private`, `default` or `package-private`, and `protected` access modifier.

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{  
        ...  
    }  
}
```

- ◆ 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.

# Benefits of Using Nested Class



- ◆ The reasons for introducing this advantageous feature of defining nested class in Java are as follows:

## 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.
- In other words, it helps in grouping the related functionality together.
- Nesting of such 'helper classes' helps to make the package more efficient and streamlined.

## Increases encapsulation

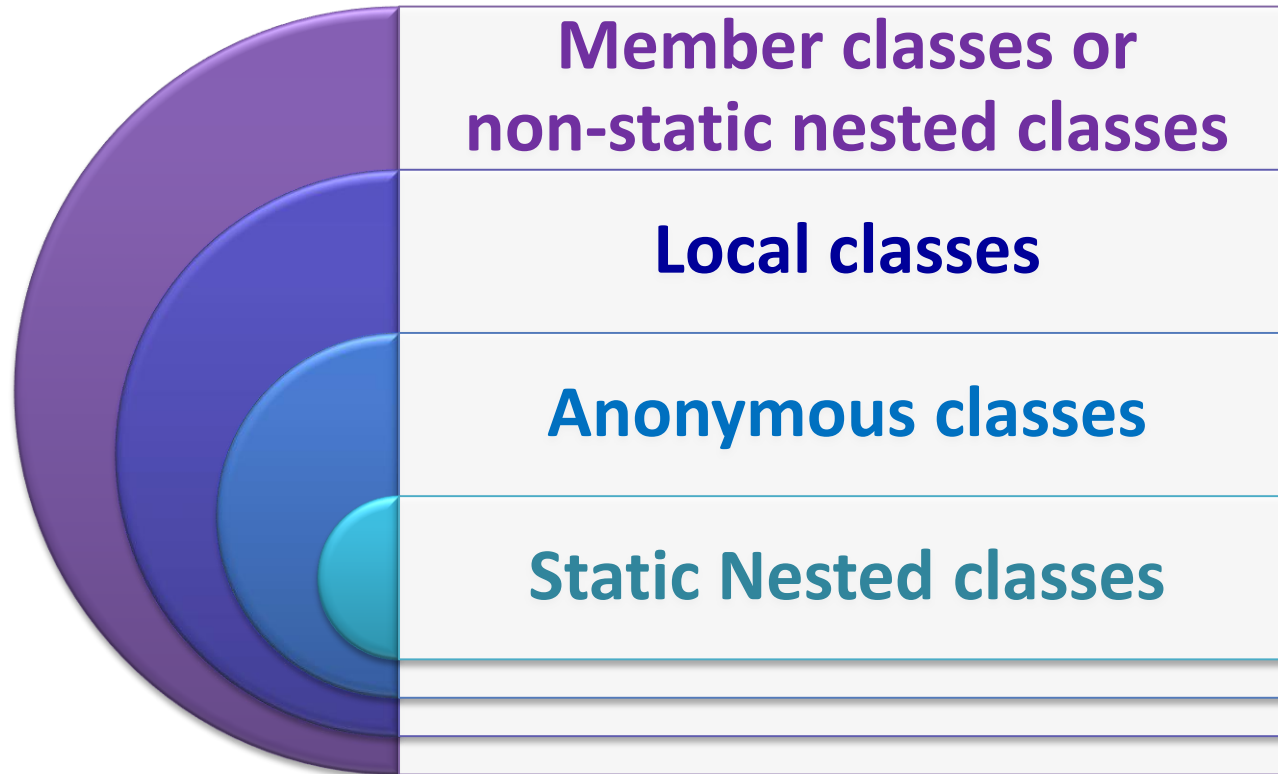
- In case of two top level classes such as class A and B, when B wants access to members of A that are `private`, class B can be nested within class A so that B can access the members declared as `private`.
- Also, this will hide class B from the outside world.
- Thus, it helps to access all the members of the top-level enclosing class even if they are 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.



- ◆ The different types of nested classes are as follows:







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:

```
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);
    }
}
```



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

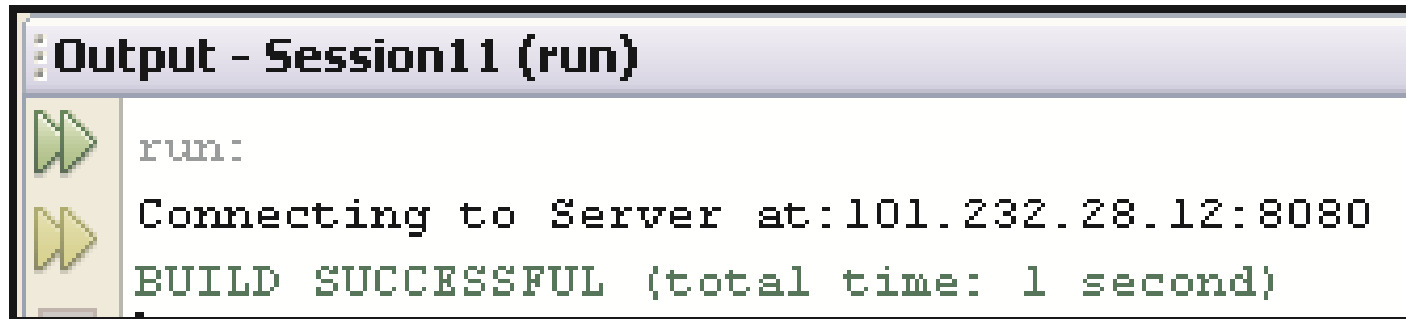
/**
 * Define the class TestConnection.java
 *
 */
public class TestConnection {
```



```
/**
 * @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:

A screenshot of an IDE's output window. The title bar reads "Output - Session11 (run)". The output area contains three lines of text: "run:", "Connecting to Server at:101.232.28.12:8080", and "BUILD SUCCESSFUL (total time: 1 second)". To the left of the text are two green right-pointing arrow icons.

```
Output - Session11 (run)
run:
Connecting to Server at:101.232.28.12:8080
BUILD SUCCESSFUL (total time: 1 second)
```



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.

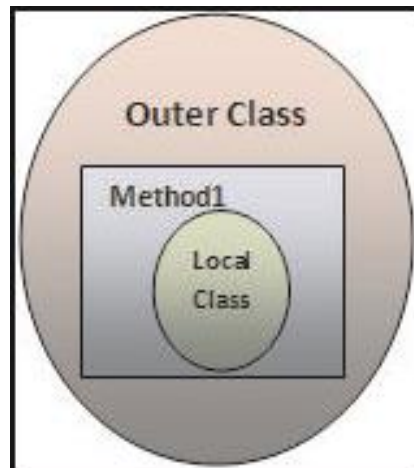
The scope of a local inner class is only within that particular block.

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

That is, it cannot use modifiers such as `public`, `protected`, `private`, or `static`.

However, it can access all members of the outer class as well as `final` variables declared within the scope in which it is defined.

- ◆ Following figure displays a 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`.
- ◆ Following code snippet demonstrates an example of local inner class.

```
package session11;
class Employee {

    /**
     * Evaluates employee status
     *
     * @param empID a String variable storing employee ID
     * @param empAge an integer variable storing employee age
     * @return void
     */
    public void evaluateStatus(String empID, int empAge){
        // local final variable
        final int age=40;
```



```
/**
 * Local inner class Rank
 *
 */
class Rank{

    /**
     * Returns the rank of an employee
     *
     * @param empID a String variable that stores the employee ID
     * @return char
     */
    public char getRank(String empID){
        System.out.println("Getting Rank of employee: "+ empID);
        // assuming that rank 'A' was returned from server
        return 'A';
    }
}

// Check the specified age
if(empAge>=age) {
```





```
// Instantiate the Rank class
Rank objRank = new Rank();

// Retrieve the employee's rank
char rank = objRank.getRank(empID);

// Verify the rank value
if(rank == 'A') {
    System.out.println("Employee rank is:" + rank);
    System.out.println("Status: Eligible for upgrade");
}
else{
    System.out.println("Status: Not Eligible for upgrade");
}
}
else{
    System.out.println("Status: Not Eligible for upgrade");
}
}
}
```

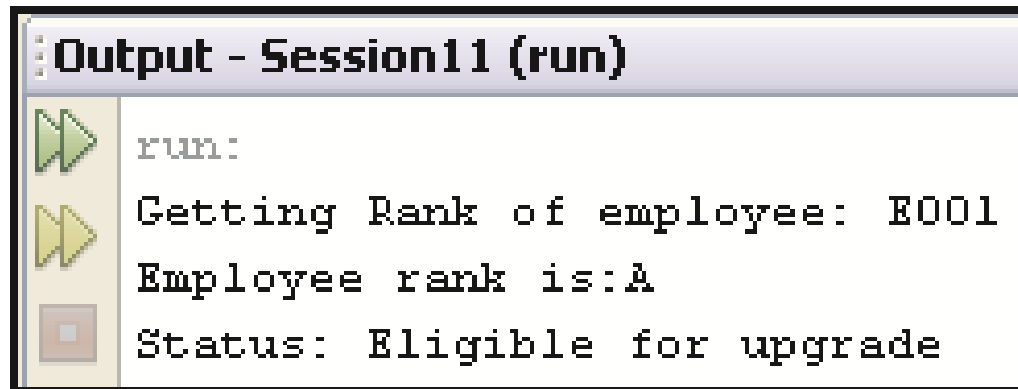


```
/**
 * Define the class TestEmployee.java
 *
 */
public class TestEmployee {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        if(args.length==2) {
            // Object of outer class
            Employee objEmp1 = new Employee();
            // Invoke the evaluateStatus() method
            objEmp1.evaluateStatus(args[0], Integer.parseInt(args[1]));
        }
        else{
            System.out.println("Usage: java Employee <Emp-Id> <Age>");
        }
    }
}
```



- ◆ The class **Employee** is the outer class with a method named **evaluateStatus(String,int)**.
- ◆ The class **Rank** is a local inner class within the method.
- ◆ The **Rank** class consists of one method **getRank()** that returns the rank of the specified employee Id.
- ◆ If the age of the employee is greater than 40, the object of **Rank** class is created and the rank is retrieved.
- ◆ If the rank is equal to 'A' then, the employee is eligible for upgrade otherwise the employee is not eligible.
- ◆ Following figure shows the output of the code when user passes 'E001' as employee Id and 50 for age:



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



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

An anonymous class does not have a name associated, so it can be accessed only at the point where it is defined.

It cannot use the `extends` and `implements` keywords nor can specify any access modifiers, such as `public`, `private`, `protected`, and `static`.

It cannot define a constructor, `static` fields, methods, or classes.

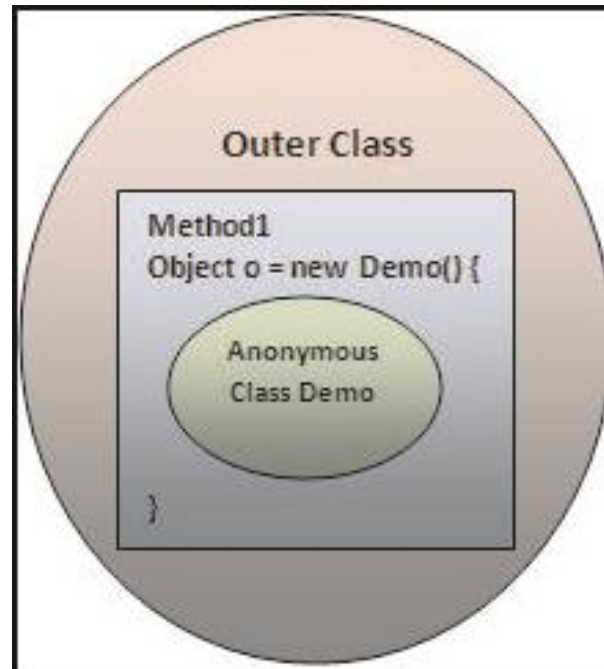
It cannot implement anonymous interfaces because an interface cannot be implemented without a name.

Since, an anonymous class does not have a name, it cannot have a named constructor but it can have an instance initializer.

Rules for accessing an anonymous class are the same as that of local inner class.



- ◆ Usually, anonymous class is an implementation of its super class or interface and contains the implementation of the methods.
- ◆ Anonymous inner classes have a scope limited to the outer class.
- ◆ They can access the internal or `private` members and methods of the outer class.
- ◆ Anonymous class is useful for controlled access to the internal details of another class.
- ◆ Also, it is useful when a user wants only one instance of a special class.
- ◆ Following figure displays an anonymous class:





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

```
package session11;
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 {
```

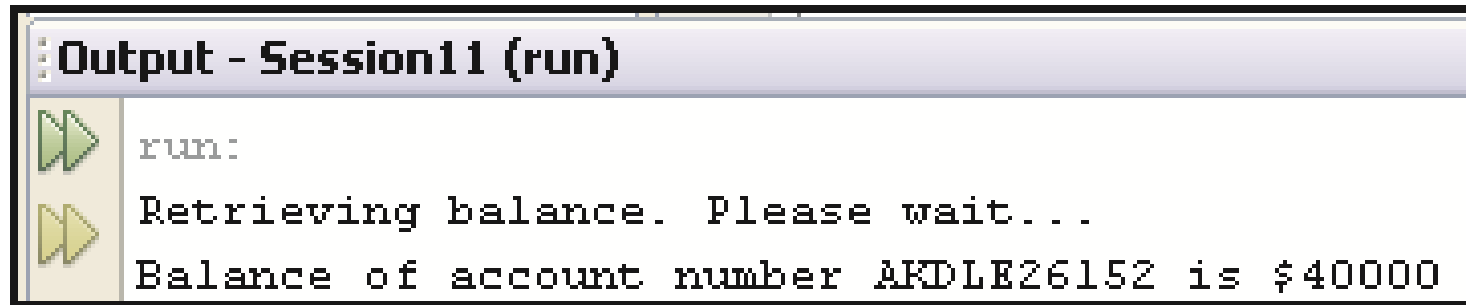


```
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    // Instantiate the Authenticate class
    Authenticate objUser = new Authenticate()
    // Check the number of command line arguments
    if (args.length == 3) {
        if (args[0].equals("admin") && args[1].equals("abc@123")) {
            // 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:

A screenshot of an IDE's output window. The title bar reads "Output - Session11 (run)". The output area contains three lines of text: "run:", "Retrieving balance. Please wait...", and "Balance of account number AKDLE26152 is \$40000". To the left of the text are two green right-pointing arrow icons.

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



A `static` nested class is associated with the outer class just like variables and methods.

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

A `static` nested class, by behavior, is a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the fully qualified class name, that is, **`OuterClass.StaticNestedClass`**.

A `static` nested class can have `public`, `protected`, `private`, default or `package private`, `final`, and `abstract` access specifiers.

- ◆ Following code snippet demonstrates the use of `static` nested class:

```
package session11;
import java.util.Calendar;
class AtmMachine {
    /**
     * Define the static nested class
     */
}
```



```
*/
static class BankDetails
{
    // Instantiate the Calendar class of java.util package
    static Calendar objNow = Calendar.getInstance();

    /**
     * Displays the bank and transaction details
     *
     * @return void
     */
    public static void printDetails(){
        System.out.println("State Bank of America");
        System.out.println("Branch: New York");
        System.out.println("Code: K3983LKSIE");

        // retrieving current date and time using Calendar object
        System.out.println("Date-Time:" + objNow.getTime());
    }
}
```



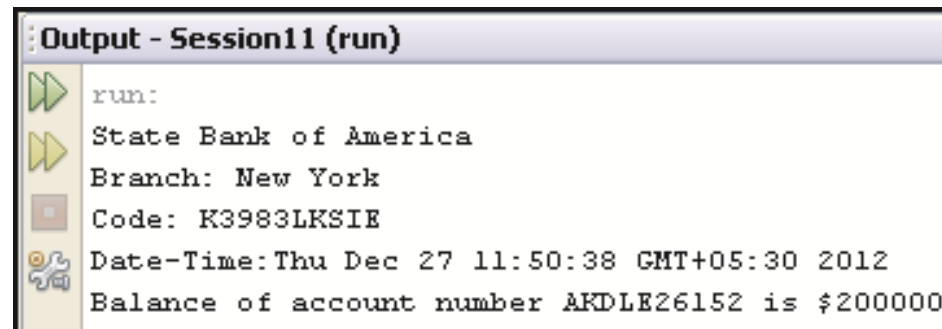
```
/**
 * Displays balance
 * @param accNo a String variable that stores the account number
 * @return void
 */
public void displayBalance(String accNo) {
    // Assume that the server returns 200000
    System.out.println("Balance of account number " + accNo.toUpperCase()
+
    " is $200000");
}
}

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



```
if(args.length ==1) { // verifying number of command line arguments
    // Instantiate the outer class
    AtmMachine objAtm = new AtmMachine();
    // Invoke the static nested class method using outer class object
    AtmMachine.BankDetails.printDetails();
    // Invoke the instance method of outer class
    objAtm.displayBalance(args[0]);
}
else{
    System.out.println("Usage: java AtmMachine <account-no>");
}
}
```

- ◆ Following figure shows the output of the code when user passes 'akdle26152' as account number:



```
Output - Session11 (run)
run:
State Bank of America
Branch: New York
Code: K3983LKSIE
Date-Time: Thu Dec 27 11:50:38 GMT+05:30 2012
Balance of account number AKDLE26152 is $200000
```



- ◆ Notice that the output of date and time shows the default format as specified in the implementation of the `getTime()` method.
- ◆ The format can be modified according to the user requirement using the `SimpleDateFormat` class of `java.text` package.
- ◆ `SimpleDateFormat` is a concrete class used to format and parse dates in a locale-specific manner.
- ◆ `SimpleDateFormat` class allows specifying user-defined patterns for date-time formatting.
- ◆ The modified **BankDetails** class using **SimpleDateFormat** class is displayed in the following code snippet:

```
import java.text.SimpleDateFormat;
import java.util.Calendar;
class AtmMachine {
    /**
     * Define the static nested class
     *
     */
    static class BankDetails {
        // Instantiate the Calendar class of java.util package
        static Calendar objNow = Calendar.getInstance();
```



```
/**
 * Displays the bank and transaction details
 *
 * @return void
 */
public static void printDetails(){
    System.out.println("State Bank of America");
    System.out.println("Branch: New York");
    System.out.println("Code: K3983LKSIE");

    // Format the output of date-time using SimpleDateFormat class
    SimpleDateFormat objFormat = new SimpleDateFormat("dd/MM/yyyy
        HH:mm:ss");

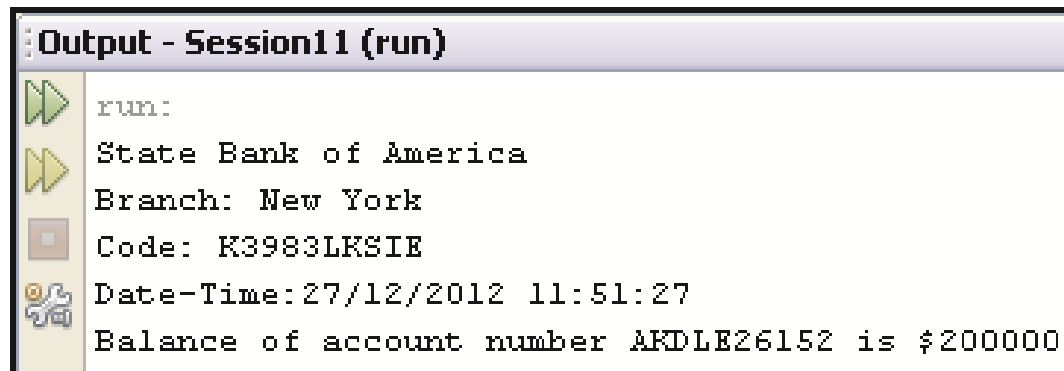
    // Retrieve the current date and time using Calendar object
    System.out.println("Date-Time:" +
        objFormat.format(objNow.getTime()));
}
...
}
```



- ◆ The `SimpleDateFormat` class constructor takes the date pattern as a `String`.
- ◆ In the code, the pattern **`dd/MM/yyyy HH:mm:ss`** uses several symbols that are pattern letters recognized by the `SimpleDateFormat` class.
- ◆ Following table lists the pattern letters used in the code with their description:

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:



```
Output - Session11 (run)
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
```

- ◆ The date and time are now displayed in the specified format that is more understandable to the user.





- ◆ 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.