
Object-Oriented Programming

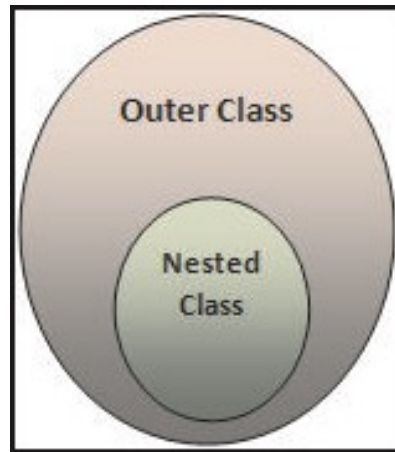
Nested Class

Objectives

- Explain Nested class
- Explain Member class
- Explain Local class
- Explain Anonymous class
- Describe Static nested class

Nested Class 1-2

- 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 Class 2-2

- 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

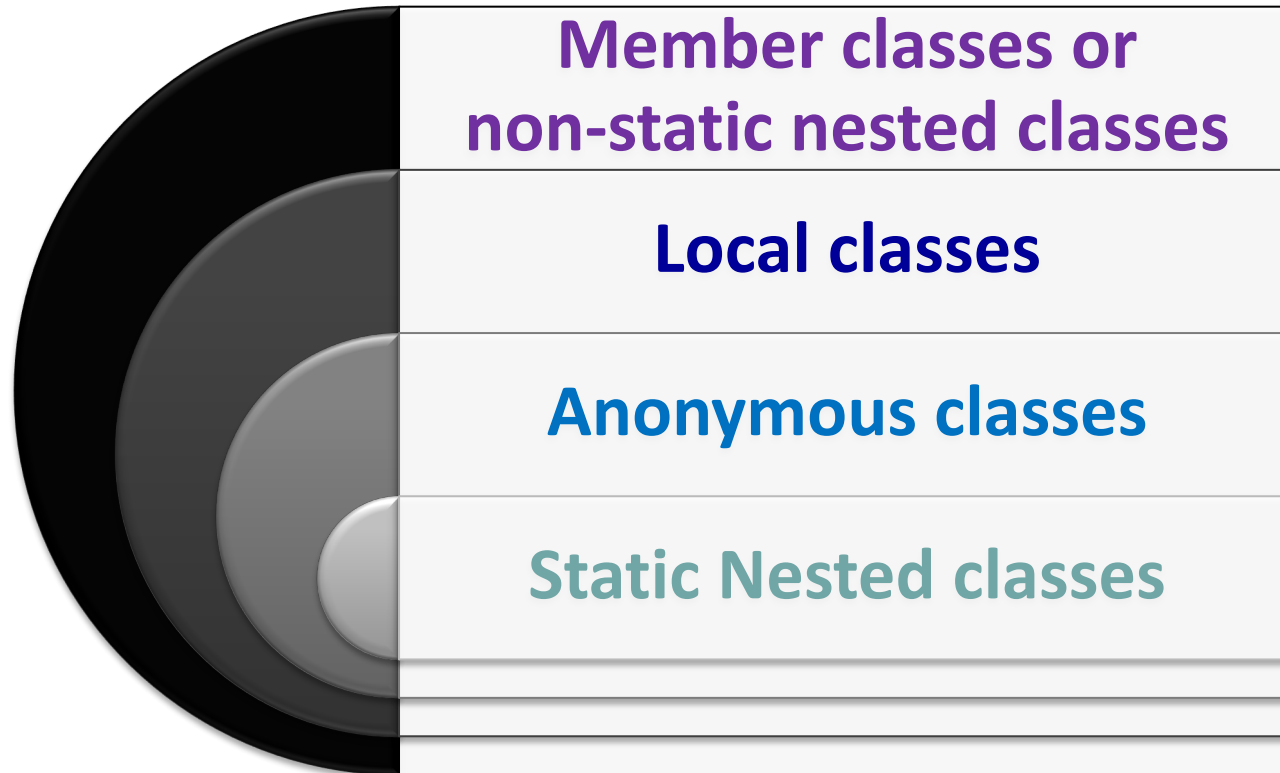
- In case of two top level classes such as class A and B, when B wants access to members of A that are `private`, class B can be nested within class A so that B can access the members declared as `private`.
- Also, this will hide class B from the outside world.
- Thus, it helps to access all the members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

Types of Nested Classes

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



Member Classes 1-5

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.

Member Classes 2-5

- ◆ 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);
    }
}
```


Member Classes 3-5

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

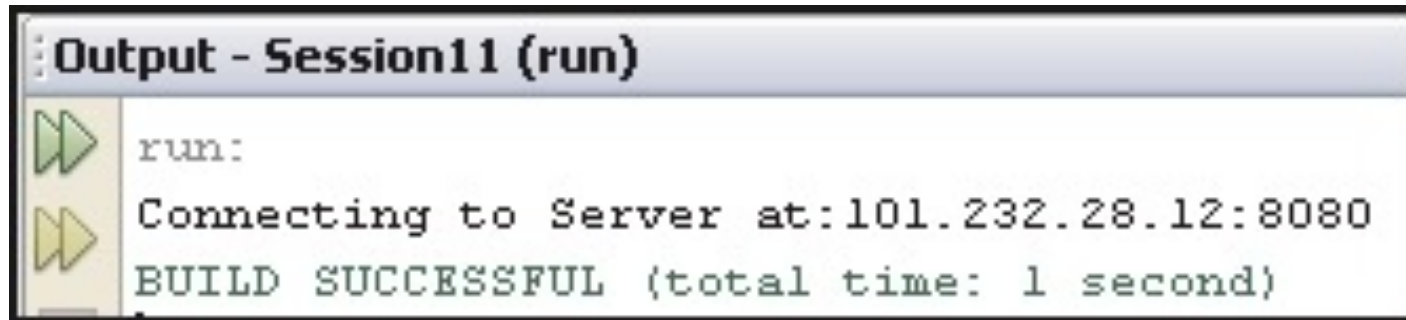
/**
 * Define the class TestConnection.java
 */
public class TestConnection {
```

Member Classes 4-5

```
/**
 * @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>"); }
}}
```

Member Classes 5-5

- ◆ 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)
```

Local Class 1-6

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.

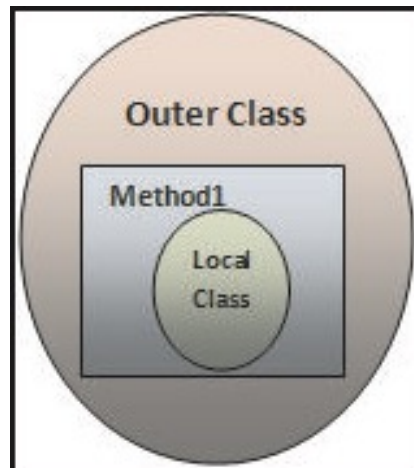
The scope of a local inner class is only within that particular block.

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 Class 2-6

- ◆ 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 Class 3-6

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

Local Class 4-6

```
// 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");
}
}
}
```

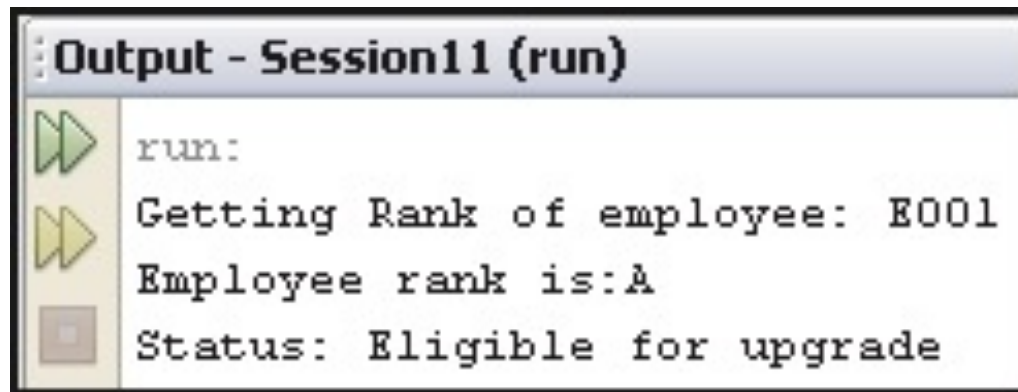
Local Class 5-6

```
/**
 * 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>");
        }
    }
}
```


Local Class 6-6

- ◆ 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
```

Anonymous Class 1-6

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.

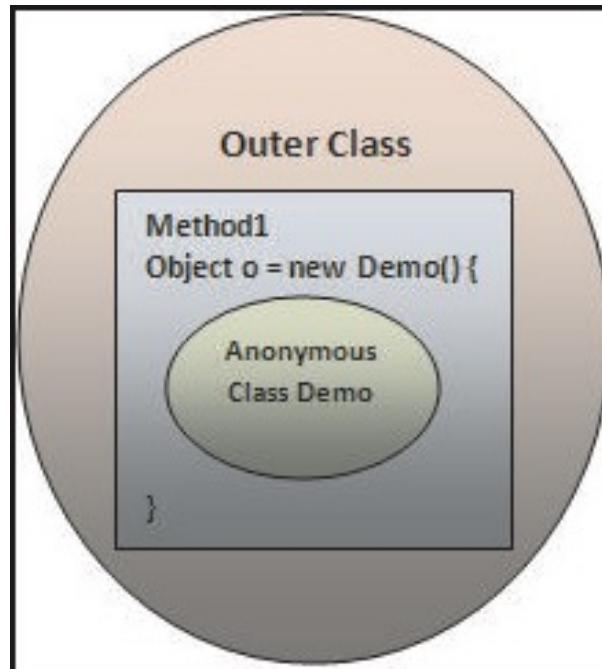
It cannot implement anonymous interfaces because an interface cannot be implemented without a name.

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.

Anonymous Class 2-6

- ◆ 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:



Anonymous Class 3-6

- ◆ 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");
        }
    }
}
```

Anonymous Class 4-6

```
}; // 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) {
    }
}

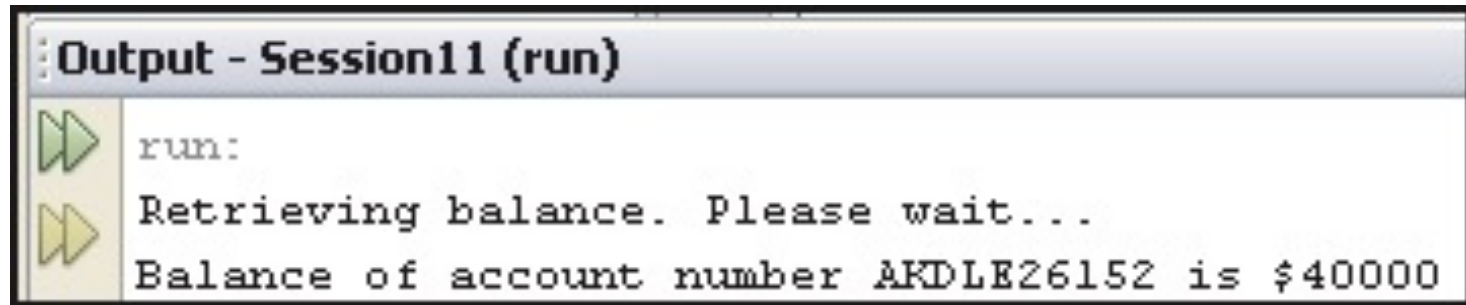
/**
 * Define the TestAuthentication class
 *
 */
public class TestAuthentication {
```

Anonymous Class 5-6

```
/**
 * @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>");
    }
}
}
```

Anonymous Class 6-6

- ◆ 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:



```
Output - Session11 (run)
run:
Retrieving balance. Please wait...
Balance of account number AKDLE26152 is $40000
```

Static Nested Class 1-7

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 Nested Class 2-7

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

Static Nested Class 3-7

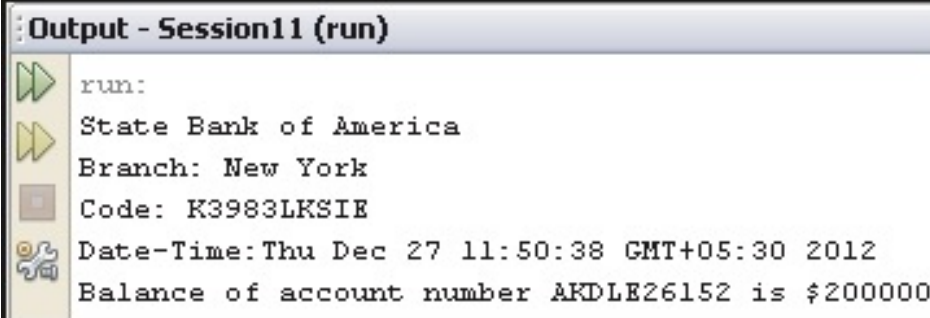
```
/**
 * Displays balance
 * @param accNo a String variable that stores the account number
 * @return void
 */
public void displayBalance(String accNo) {
    // Assume that the server returns 200000
    System.out.println("Balance of account number " + accNo.toUpperCase() +
        " is $200000");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

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

Static Nested Class 4-7

```
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:



The screenshot shows an IDE output window titled "Output - Session11 (run)". It contains the following text:

```
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
```

Static Nested Class 5-7

- ◆ 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();
```

Static Nested Class 6-7

```
/**
 * 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");

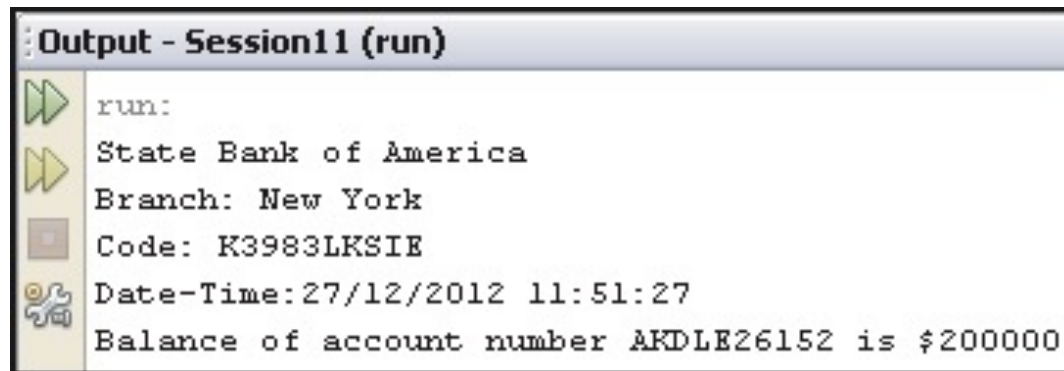
    // Retrieve the current date and time using Calendar object
    System.out.println("Date-Time:" +
    objFormat.format(objNow.getTime()));
}
}
...
...
}
```

Static Nested Class 7-7

- ◆ 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.

Summary

- ◆ 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.