CHAPTER

34

LAMBDA EXPRESSIONS

So far, the entire focus in Java language is only on objects. But now, more importance is given to the functional aspects of programming. JavaSoft people realized that doing everything using objects is becoming cumbersome and using functions (or methods) can be more efficient in certain cases. 'Lambda expressions' is the most important feature that brings revolutionary change to the way programming is done in Java.

Lambda Expressions and Functional interfaces

A lambda expression is a method without a name, access specifier, and return value declaration. Such a method is also called an 'anonymous method' or 'closure'. Remember that an anonymous class is a class that does not have a name but for which an object can be created. In the same manner, a lambda expression is a method without a name but can be used to perform a task.

Important Interview Question

What is lambda expression?

A lambda expression is a method without a name, access specifier, and return value declaration. Lambda expressions are also known as 'anonymous methods' or 'closures'.

To understand the concept of lambda expression, let us write a simple method to display a message as:

```
public void message()
{
System.out.println("Hello how are U?");
 }
```

We can convert this method into a lambda expression by removing the access specifier 'public', return type declaration 'void', and the name 'message' and write it as:

```
() -> {
System.out.println("Hello how are U?");
 }
```

Observe the empty parentheses. They represent that the method does not have any parameters. After them, we should write ' -> ' symbol which is called arrow symbol. This -> symbol separates the method header with the method body. Hence, after the -> symbol, we should write the body of the method. The above lambda expression is also written in a single line as:

```
() ->{  System.out.println("Hello how are U?");  }
```

Let us take another example. We have the following method to add two integers:

```
void add(int a, int b)
{
System.out.println("Sum= "+ (a+b));
}
```

The preceding method can be converted into a lambda expression as:

```
(int a, int b) ->{ System.out.println("Sum= " +(a+b)); }
```

Similarly, the method that returns a string is shown here:

```
String display(String str)
{
returnstr;
}
```

The preceding method can be written as a lambda expression as:

```
(String str) ->{ returnstr; }
```

When the type of the parameters can be decided by the compiler automatically, we can omit the type and write the lambda expression. For example, the preceding expression can be rewritten as:

```
( str) ->{ returnstr; }
```

Points regarding construction of the lambda expressions

The following points must be kept in mind while constructing lambda expressions:

- A lambda expression can have zero, one or more parameters. For example,

```
() ->System.out.println("Hello");   //zero parameters
(int a, int b) ->{ returna+b; }  //two parameters
```

- The programmer should mention the type of the parameters. If the type of the parameters can be decided by the compiler, then they need not be mentioned by the programmer. For example,

```
(int a, int b) ->{ System.out.println("Sum= " +(a+b)); }  //type mentioned
(a, b) ->{ System.out.println("Sum= " +(a+b)); }  //type is decided implicitly
```

- The parameters should be separated by commas while mentioned in the parentheses. For example,

```
(int a, int b)
(a, b)
```

- When there are zero parameters, we have to put empty parentheses. For example,

```
() ->System.out.println("Hello");
() -> 10;  //return 10 as result
```

- In case of a single parameter, if its type is known to the compiler, then we need not use parentheses. For example,

```
(int a) ->{ return a+100 ; };  //single parameter
a -> { return a+100 ; };
```

- Just like methods, we can write zero or more statements in the body of the lambda expression. For example,

```
() ->{ System.out.println(x); }  //single statement
() ->{ System.out.println(this.x);
System.out.println( x); };  //two statements
```

- When there is more than one statement in the body, we should enclose them inside the curly brackets. When there is only one statement, curly brackets are not compulsory. For example,

```
() ->System.out.println("Var of the class= " + this.x);  //no curly brackets
```

Once a lambda expression is written, it can be called and executed like a method. The next question is how to call a lambda expression. For this purpose, first we should create a 'functional interface'.

Functional interface

A functional interface is an interface that contains only one abstract method. It is also possible that a functional interface may have more than one static or default method. It can also override methods of the 'Object' class of the java.lang package. Let us look at the following code snippet to understand a functional interface:

```
interface MyInter
{
    //Only one abstract method
void method1(int a, int b);

    //functional interface can have more than one static or default methods
default void method2()
  {
System.out.println("This is default method");
  }

  //it can also override Object class methods
public String toString()
  {   }
}
```

A default method is a method in an interface that is declared as 'default'. This method is available by default to all the implementation classes of that interface. The implementation classes can override this method, if they feel like providing a different implementation for the default method. This is a new feature of Java 8.0.

Some of the functional interfaces that are already available in Java are:

- ActionListener interface (in java.awt.event package) that contains only actionPerformed() method
- Runnable interface (in java.lang package) that contains only run() method
- FileFilter interface (in java.io package) that contains only accept() method
- Callable interface (in java.util.concurrent package) that contains only call() method
- Comparator interface (in java.util package) that contains only compare() method

The main advantage of a functional interface is that it can be used to assign a lambda expression or it can be used as a method reference. Let us understand this more clearly. After writing the lambda expression, we should assign it to the reference of a functional interface as:

```
MyInter mi = () ->{ System.out.println("Hello how are U?"); };
```

Here, 'MyInter' is a functional interface whose reference mi is referencing the lambda expression at the right hand side. This is possible because the functional interface, MyInter, contains an abstract method 'void message()'. This method is in the following form:

```
void message()
{
System.out.println("Hello how are U?");
 }
```

When this method is converted into lambda expression, we will have:

```
() ->{ System.out.println("Hello how are U?"); };
```

This lambda expression should be referenced using the MyInter interface reference as:

```
MyInter mi = () ->{ System.out.println("Hello how are U?"); };
```

Since the reference, mi, is referring to the lambda expression, if we want to call and execute the lambda expression, we should call it using mi as:

```
mi.message();
```

The lambda expression gets executed and the output "Hello how are U?" is displayed. Our first program will clear this total concept.

Program 1: Write a Java program to create a lambda expression that displays a Hello message. Call and execute the expression using functional interface reference.

```
//a lambda expression to display Hello message
class LambdaDemo1
{
   //create a functional interface with a single abstract method
interfaceMyInter
   {
void message();
   }

public static void main(String args[])
   {
      //create functional interface reference that refers to lambda expression
MyInter mi = () ->{ System.out.println("Hello how are U?"); };
      //call the method using reference
mi.message();

   }
}
```

Output:

```
C:\>javac LambdaDemo1.java
C:\>java LambdaDemo1
Hello how are U?
```

Important Interview Question

What is a functional interface?

A functional interface is an interface that contains only one abstract method. The main advantage of a functional interface is that it can be used to refer to a lambda expression or as a method reference. Thus execution of lambda expressions or methods can be possible.

Now, we will try to create a lambda expression to find the sum of two integers. First assume a method like this:

```
void add(int a, int b)
{
System.out.println("Sum= "+ (a+b));
 }
```

This method can be converted into a lambda expression by eliminating the method return type, name, as:

```
(int a, int b) ->{ System.out.println("Sum= " +(a+b)); }
```

To execute this lambda expression, we need to assign it to a functional interface reference as:

```
MyInter mi = (int a, int b) ->{ System.out.println("Sum= " +(a+b)); }
```

It is compulsory that the functional interface, MyInter, should have the abstract method that is contained in the lambda expression. It means void add(int a, int b) should be a method of the MyInter interface. Thus the interface would look like:

```
interfaceMyInter
 {
void add(int a, int b);
 }
```

Finally, to call the method, we have to use the MyInter interface reference as:

```
mi.add(10, 22);  //call the method and pass 10 and 22
```

Program 2: Let us write a program to create and execute a lambda expression that adds two integers.

```
//create a lambda expression to find sum of two integers
class LambdaDemo2
{
   //create a functional interface with a single abstract method
interfaceMyInter
   {
void add(int a, int b);
   }

public static void main(String args[])
   {

//create functional interface reference that refers to lambda expression
MyInter mi = (int a, int b) ->{ System.out.println("Sum= " +(a+b)); };

      //call the method using reference
mi.add(10, 22);

   }
}
```

Output:

```
C:\>javac LambdaDemo2.java
C:\>java LambdaDemo2
Sum= 32
```

Let us create a lambda expression that calculates and returns the square root value of a given number. For this purpose, imagine a method like this:

```
doublesquareroot(double x)
{
returnMath.sqrt(x);
}
```

This method can be rewritten as a lambda expression by eliminating the return type and name of the method as:

```
(double x) ->{ returnMath.sqrt(x); }
```

This expression should be assigned to a functional interface MyInter reference as:

```
MyInter mi = (double x) ->{ returnMath.sqrt(x); };
```

Then, we can execute the lambda expression using mi as:

```
double result = mi.squareroot(256);
```

Program 3: Write a Java program to create a lambda expression that calculates and returns the square root value of a given number.

```
//a lambda expression that returns square root value of a number
class LambdaDemo3
{
   //create a functional interface with a single abstract method
interfaceMyInter
   {
doublesquareroot(double num);
   }

public static void main(String args[])
   {
      //create functional interface reference that refers to lambda expression
MyInter mi = (double x) ->{ returnMath.sqrt(x); };

      //call the method using reference

System.out.println("Square root of 256= "+ mi.squareroot(256));

   }
}
```

Output:

```
C:\>javac LambdaDemo3.java
C:\> java LambdaDemo3
Square root of 256= 16.0
```

Important Interview Question

What is the target type of a lambda expression?

The target type of a lambda expression represents a type to which the expression can be converted. All lambda expressions are converted into a functional interface type. So, the target type of a lambda expression is functional interface.

What is the type of a lambda expression?

If we take separately, a lambda expression does not have any type. Its type is dependent on the context in which it appears and many times, its type is inferred by the Java compiler.

Accessing variables using Lambda expressions

We can access the variables declared inside a class or inside the same method where a lambda expression is defined. But, the variables of the method where the lambda expression is declared are treated implicitly as 'final'. We already know that the final type of variables cannot be modified. Hence, lambda expressions cannot modify them. But, they can modify the variables declared inside the class.

If a variable is declared with the same name in the class and also in the method, we should take the help of 'this' keyword to refer to the variable of the class. For example, 'x' is a variable in the class, and we can refer to it as 'this.x' using the lambda expression. Similarly, if 'x' is a variable in the method, we can refer to it simply as 'x' in the lambda expression. See Program 4.

In Program 4, we have taken a variable x in the class 'LambdaDemo4'. There is another variable in the method, void method(), with the same name 'x'. This method contains the following lambda expression:

```
() ->{ System.out.println("Var of the class= " + this.x);
System.out.println("Var of the method= "+ x); }
```

This lambda expression is using two println() statements to display the variable of the class and of the method as:

```
this.x  // represents variable of the class
x  //variable of the method
```

The next step is to run this lambda expression. For this purpose, we should use a functional interface reference. In this case, we are using Runnable interface reference to refer to the lambda expression as Runnable is a functional interface.

```
Runnable r = () ->{ System.out.println("Var of the class= " + this.x);
System.out.println("Var of the method= "+ x); }
```

This reference 'r' is passed to a Thread class object as:

```
Thread t = new Thread(r);
```

Now, if we run the thread by calling t.start(), the lambda expression that is referenced by 'r' will be executed.

Program 4: Write a program to access variables of a class and of a method using lambda expression.

```
//a lambda expression that accesses the variables with class scope and method
scope
```

```
class LambdaDemo4
{
   //variable in the class
int x = 10;

   //method in the class
void method()
  {
     //variable in the method
int x = 20;

     //create reference of functional interface to the lambda expression
     Runnable r = () ->{ System.out.println("Var of the class= " + this.x);
System.out.println("Var of the method=     "+ x); };

     //create a thread and run it
     Thread t = new Thread(r);
t.start();
  }

public static void main(String args[])
  {
     //create object to the class and call the method
     LambdaDemo4 obj = new LambdaDemo4();
obj.method();
  }

}
```

Output:

```
C:\>javac LambdaDemo4.java
C:\> java LambdaDemo4
Var of the class= 10
Var of the method= 20
```

The need of lambda expressions

Since Java is an object oriented language, everything is considered as an object. With this notation, performing operations on a huge number of objects will become difficult. For example, to handle several objects, Java offers the java.util package where several objects can be stored in a single object called 'collection object'. When performing operations on several objects, we may take the help of loops like for-each loop. Such looping operation will deal with object by object and not multiple objects at a time. We need to improve the functional aspect rather than the object orientation. Hence, JavaSoft people thought of lambda expressions which are nothing but simplified functions. Internally, lambda expressions are viewed as objects by the JVM. For example, in Program 4, JVM represents the lambda expression as an object of type Runnable interface.

When we are dealing with lambda expressions, we feel like an expression is being used to perform the tasks. This expression is referenced by a functional interface reference. This reference in turn can be passed to methods. It means we are passing lambda expressions as arguments to methods. This is another convenience of using lambda expressions. This concept is explained through Program 5.

We know that an anonymous inner class is a class that does not have a name. We generally use an anonymous inner class when we want to create an object to the class and pass it to a method as argument. In this case, using anonymous inner class is clumsy. Instead, passing a lambda expression like an object makes programming simple. This concept is explained in Programs 8 and 9.

Passing lambda expressions as arguments to a method

Lambda expressions perform the tasks like methods since they represent simplified methods. Lambda expressions are referenced by functional interface reference. Hence, by passing the functional interface reference as an argument to a method, we can pass the lambda expression as an argument to the method.

For example, let us take a lambda expression that calculates the area of a circle upon receiving the radius of the circle as:

```
(r) -> {System.out.println("Area= "+ Math.PI * r * r);}
```

This expression is in fact a representation of the method: void calculate (double radius) which is declared as an abstract method in the functional interface 'Circle'.

Hence, we can refer to this lambda expression using Circle reference 'ref' as:

```
Circle ref = (r) -> {System.out.println("Area= "+ Math.PI * r * r);};
```

Now, this reference 'ref' can be passed to any method as an argument where that method can utilize this reference to call the calculate() method as:

```
voidcircleArea(double radius, Circle ref)
 {
ref.calculate(radius);   //this will execute the lambda expression
 }
```

Here, Circle 'ref' is passed as second argument to circleArea() method. This method in turn calls calculate() method using this 'ref' as: ref.calculate();

Program 5: Let us write a program to calculate area of a circle by passing a lambda expression to a method.

```
//Passing lambda expression as an argument to a method
//we are going to calculate area of a circle
class LambdaDemo5
{
   //create a functional interface with one abstract method
interface Circle
   {
void calculate(double radius);
   }
   //a method with functional interface reference ref as its argument
voidcircleArea(double radius, Circle ref)
   {
      ref.calculate(radius); //this will execute the lambda expression
   }

public static void main(String args[])
   {
      //create object to the class
      LambdaDemo5 obj = new LambdaDemo5();

      //let the functional interface reference refer to the lambda expression
//this lambda expression implements calculate(radius) method
      Circle ref = (r) -> {System.out.println("Area= "+ Math.PI * r * r);};

      //call the method, pass radius and lambda expression as arguments
obj.circleArea(20, ref);
   }
}
```

Output:

```
C:\>javac LambdaDemo5.java
C:\> java LambdaDemo5
Area= 1256.6370614359173
```

Passing lambda expressions to objects

It is possible to pass lambda expressions to class objects like we pass a value. To understand this concept, first we will begin with a simple thread program. Let us write a Java program using a thread that displays a message. While dealing with threads, we know that the class where run() method is written should implement Runnable interface.

Program 6: Write a program to display a message using a thread.

```
//a simple thread program
class LambdaDemo6 implements Runnable
{
   //implement the run() of the Runnable interface
public void run()
  {
System.out.println("This is from thread");
  }
public static void main(String args[])
 {
    //create an object to the class
    LambdaDemo6 obj = new LambdaDemo6();

    //attach thread to the object
    Thread t = new Thread(obj);

     //run the thread on the object
t.start();
 }
}
```

Output:

```
C:\>javac LambdaDemo6.java
C:\> java LambdaDemo6
This is from thread
```

Suppose we do not want to implement Runnable interface with LambdaDemo6 class. Then, we have to create a separate implementation class (for example, Implclass) to the Runnable interface and then we should pass that class object to the thread as:

```
Thread t = new Thread(new Implclass());
```

This is denoted in Program 7 which does the same task as Program 6.

Program 7: Write a thread program to display a message, where a separate implementation class for Runnable interface is created.

```
//a simple thread program - version 2
//in this version, we are separately creating implementation
//class of Runnable interface
class LambdaDemo7
{
public static void main(String args[])
 {
    //create thread object and pass the object of
```

```
        //implementation class of Runnable interface
        Thread t = new Thread(new Implclass());

        //run the thread
    t.start();
    }
}

//this is the implementation class for Runnable interface
classImplclass implements Runnable
{
        //implement the run() of the Runnable interface
public void run()
    {
System.out.println("This is from implementation class");
    }
}
```

Output:

```
C:\>javac LambdaDemo7.java
C:\> java LambdaDemo7
This is from implementation class
```

In Program 7, we can make the Implclass as anonymous inner class. When it becomes anonymous inner class, we need not mention its name but we can create an object. The main advantage of anonymous inner class is that the code of the class can be passed to a method like an object. For example, to pass Implclass object to Thread, we have written as:

```
Thread t = new Thread(new Implclass());
```

If Implclass becomes anonymous class, then we need not mention its name. Now, what we got is this:

```
Thread t = new Thread(new .... );
```

In the above statement, after 'new' operator, we can copy-paste the code of the Implclass excluding its name. Now, we will have:

```
Thread t = new Thread(new Runnable())
{
public void run()
    {
System.out.println("This is from implementation class");
    }
}
```

From the preceding code, it appears as if we created an object to Runnable interface and passed it to the Thread. But, the fact is that we have created an object to the Implclass and passed it to the Thread. In this case, the name of the Implclass is not mentioned after the new operator. Hence, Implclass is classed as an anonymous class. Since Implclass code is copied into another class (for example, LambdaDemo8 class), it is called anonymous inner class. This code is implemented in Program 8.

Program 8: Write a thread program to display a message by using anonymous inner class.

```
//a simple thread program - version 3
//in this version, we are using anonymous inner class
class LambdaDemo8
{
public static void main(String args[])
 {
```

```
        //create thread object and pass the object of anonymous class
         Thread t = new Thread(new Runnable()
        {
           //implement the run() of the Runnable interface
    public void run()
        {
    System.out.println("This is from anonymous inner class");
        }
        });

        //run the thread
    t.start();
     }
    }
```

Output:

```
C:\>javac LambdaDemo8.java
C:\> java LambdaDemo8
This is from anonymous inner class
```

Lambda expressions can replace anonymous inner classes. Passing the code of an anonymous inner class, especially in the case of bigger applications is cumbersome. Hence, it is better to take the help of lambda expressions to simplify the code. This is shown in Program 9.

Program 9: Write a thread program to display a message using lambda expression.

```
//a simple thread program - version 4
//in this version, we are passing lambda expression to Thread class object
class LambdaDemo9
{
public static void main(String args[])
 {
    //create thread object and pass lambda expression
    Thread t = new Thread(
     //lambda expression to implement run() method
     () -> {System.out.println("This is from lambda expression");}
    );

    //run the thread
t.start();
 }
}
```

Output:

```
C:\>javac LambdaDemo9.java
C:\> java LambdaDemo9
This is from lambda expression
```

To understand the concept more clearly, we will write another program to display a push button using swing. When the button is clicked, we want to display a message at system prompt. To add action to the button, we should add action listener using the method addActionListener(). But, this method takes the object of implementation class of ActionListener interface. The implementation class of ActionListener interface should implement actionPerformed() method. So, the code looks like this:

```
but.addActionListener(new ActionListener()
 {
public void actionPerformed(ActionEventae)
    {
```

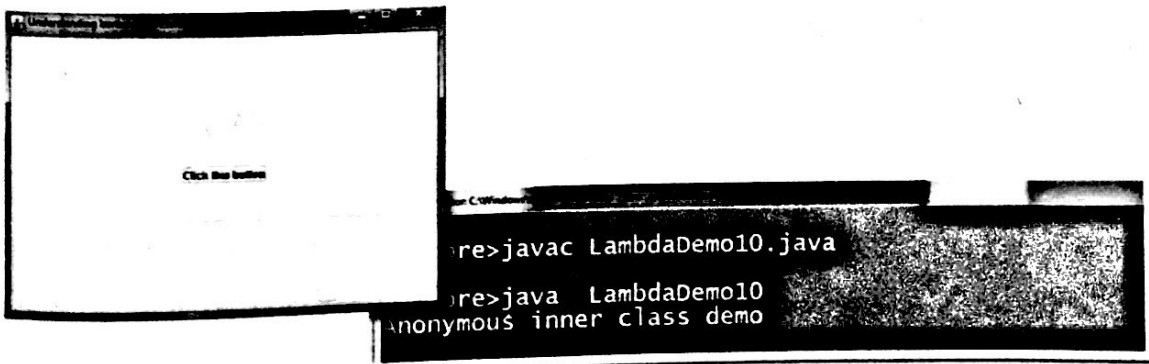
```
        System.out.println("Anonymous inner class demo");
        }
    });
```

Observe that we are passing anonymous inner class object to addActionListener() method in the above code. The complete code is given in Program 10.

Program 10: Write a Java program to display a message when a push button is clicked using anonymous inner class.

```
//In this program, push button works with Anonymous inner class
importjava.awt.*;
importjava.awt.event.*;
importjavax.swing.*;
class LambdaDemo10
{
public static void main(String args[])
  {
    //create a push button with label
JButton  but = new JButton("Click this button");

    //here, we are passing anonymous inner class object to
    //addActionListener() method
but.addActionListener(new ActionListener()
   {
public void actionPerformed(ActionEventae)
      {
      System.out.println("Anonymous inner class demo");
      }
    });

    //create frame
JFrame f = new JFrame("Understanding lambda expressions");

    //add button to the frame
f.add(but);

    //set the size of the frame
f.setSize(500, 350);

    //display the frame
f.setVisible(true);

    //close the frame when close button of frame is clicked
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

Output:



The same program can be rewritten using lambda expression in the place of anonymous inner class. We can understand that using lambda expression makes the coding easy. Observe the code passed to addActionListener method as:

```
but.addActionListener(e -> { System.out.println("Lambda expression demo");})
```

Here, 'e' indicates the object of the implementation class of ActionListener. This is internally understandable to the compiler. Hence we need not mention its type (or class name). Also, in the lambda expression, we need not mention the method name, i.e. public void actionPerformed(ActionEventae). Only the body of the method is mentioned. Hence, our code size is reduced.

So, shall we take anonymous inner classes and lambda expressions to be same? The answer is 'No'. We can have an anonymous inner class with constructors, variables and some methods. But, lambda expression indicates only a method. We can take a lambda expression as an anonymous method. Lambda expressions are anonymous methods which are designed to eliminate overhead (constructor and other bulky code) of anonymous class.

Program 11: Write a Java program to display a message when a push button is clicked using lambda expression.

```
//In this program, push button works with lambda expression - version 2
importjava.awt.*;
importjava.awt.event.*;
importjavax.swing.*;
class LambdaDemo11
{
public static void main(String args[])
  {
     //create a push button with label
JButton  but = new JButton("Click this button");

     //here, we are passing lambda expression to
     //addActionListener() method
but.addActionListener(e -> { System.out.println("Lambda expression
demo");});

     //create frame
JFrame f = new JFrame("Understanding lambda expressions");

     //add button to the frame
f.add(but);

     //set the size of the frame
f.setSize(500, 350);

     //display the frame
f.setVisible(true);

     //close the frame when close button of frame is clicked
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

Output:

```
Same as in case of Program 10.
```

Important Interview Question

What are the advantages of lambda expressions?

1. We can pass a lambda expression as an object to a method. This will reduce the overhead involved in passing anonymous class to a method. 2. Lambda expressions are useful to pass objects like data to methods. 3. Using lambda expressions, we can pass a method as an argument to another method.

How are anonymous classes different from lambda expressions?

Anonymous class	Lambda expression
Anonymous class is a class that can contain constructors, variables and methods.	Lambda expression is an anonymous method.
A programmer should use anonymous class when he wants to work with some variables and more than one method.	A programmer should opt for lambda expression when he wants to implement only one method of the functional interface.
In the anonymous class, 'this' keyword represents that class itself.	In case of lambda expression, 'this' keyword represents the enclosing class that contains the expression.
The compiler creates anonymous class as a separate class with the syntax outerclass$1.	During compilation, Lambda expression is converted into a private method of the class. During runtime, it is linked with actual code.

Default Methods

We know that an interface is a specification of method prototypes. All the methods of an interface are by default public and abstract. It means that no method can have a body. We cannot write concrete methods in an interface. Also, an interface contains only public static and final type variables.

It is necessary that all the methods of an interface should be implemented in its implementation classes. After that, it is possible to create objects to implementation classes.

In Java 8.0, a new type of method is allowed to be written inside an interface. They are called 'default methods' or 'defender methods'. A default method is declared using the keyword 'default' and it contains body also. For example,

```
defaultintmul(int x, int y)  //this is default method
  {
return (x*y);
  }
```

A default method is by default available to all the implementation classes of the interface. If the implementation class wants to use it, it can use it. Otherwise, it can ignore it and continue using its own methods.

Program 12: Write a program to understand how to use default methods of an interface.

```
//an interface with a default method
interface MyInter
{
int add(int x, int y); //this public and abstract

defaultintmul(int x, int y)  //this is default method
```

```
    {
return (x*y);
    }
}
class A implements MyInter
{
publicint add(int x, int y)
   {
return (x+y);
   }
}
class DefaultDemo1
{
public static void main(String args[])
  {
      //interface reference can refer to object of its implementation classes
MyInter mi = new A();
System.out.println("Sum= " + mi.add(10, 15));

      //default method is by default available in implementation class
System.out.println("Product= "+ mi.mul(10, 15));

  }
}
```

Output:

```
C:\>javac DefaultDemo1.java
C:\> java DefaultDemo1
Sum= 25
Product= 150
```

Lambda expressions require adding methods to functional interfaces. When we add a new method to an interface, the same method is to be implemented (writing body) in all of its implementation classes without fail. This is an inconvenience. For example, one of the main uses of lambda expressions is to work with collections. When we add a method to the List interface in collections framework, then we need to provide implementation for that method in several classes. In such cases, default methods will help us. Adding the default method to an interface makes this method available to all its implementation classes. The implementation classes can make use of it or ignore it.

We know that Java does not support multiple inheritance. However, multiple inheritance can be achieved by using interfaces something like this:

```
classImplclass implements One, Two
```

In this case, all the members of both the interfaces (i.e. One and Two) are available to the Implclass. Now, the question is will there be any conflict if we use default methods with the same name in both the interfaces. See the following code:

```
interface One
{
default void message()
    {
System.out.println("Hello from One");
    }
}
interface Two
{
default void message()
    {
```

```
    System.out.println("Hello from Two");
    }
}
```

In such a case, we should override the message() method in the implementation class and then call a specific method using the notation: interfacename.super.method(). For example, to refer to the message() method of interface One, we can use: One.super.message() and to refer to the message() method of interface Two, we can use: Two.super.message().

Program 13: Write a program to understand how to refer to a particular default method when the same method is found in two interfaces.

```
//two interfaces with same default method name
interface One
{
default void message()
    {
System.out.println("Hello from One");
    }
}
interface Two
{
default void message()
    {
System.out.println("Hello from Two");
    }
}
classImplclass implements One, Two
{
   //override the message() method to resolve the confusion
public void message()
    {
Two.super.message(); //this will call Two's message()
    }
}
class DefaultDemo2
{
public static void main(String args[])
  {
      //create implementation class object
Implclassic = new Implclass();
ic.message();

  }
}
```

Output:

```
C:\>javac DefaultDemo2.java
C:\> java DefaultDemo2
Hello from Two
```

When we add default methods to an interface, it looks like an abstract class. But, both are not same. An abstract class can have constructors, variables and concrete methods in addition to abstract methods. But, an interface with the default method does not contain constructors and variables. So, it cannot store data. It can contain only default methods and abstract methods. Already, we know that an interface with a single default method is called functional interface. Such a functional interface can become a reference to a lambda expression. But, an abstract class cannot be used like that.

Important Interview Question

What is the difference between an interface with default method and an abstract class?

Interface with default method	Abstract class
It does not contain constructors and variables.	It contains constructors and variables.
A default method can be added to an interface when the programmer wants to add a new functionality.	Abstract class contains concrete methods which provide different functionalities to subclasses.
Adding a new default method to an interface will not affect the implementation classes.	Adding a new method to the abstract class needs implementation in all of its subclasses.
An interface with a single default method can be used to refer to a lambda expression.	Abstract class cannot be used to refer to lambda expressions.

Predicates

A predicate is a function with a single argument that returns a boolean value. In Java 8.0, there is an interface by the name Predicate<T> available in java.util.function package. Here, T indicates the type of input taken by the Predicate. This Predicate interface is a functional interface that can refer to a lambda expression. This Predicate interface contains a method test() which takes one argument and returns boolean value.

Let us write a method to find all numbers which are greater than 10 as:

```
boolean method(int i)
{
if(i>10) return true;
else return false;
}
```

This function can be written as a lambda expression as:

```
(i) ->{ if(i>10) return true; else return false; }
```

Since, the Java compiler knows the type of the values returned, we can reduce this expression as:

```
(i) ->  (i) -> i>10;
```

Since Predicate is a functional interface, we can use its reference to refer to the lambda expression as:

```
Predicate<Integer>gt = (i) -> i>10;
```

Now, we can call the test() method to know whether n is greater than 10 or not as:

```
boolean result = gt.test(n);
```

If n is greater than 10, then the result will be true, otherwise it will be false.

Program 14: Write a program using predicate to know whether a given number is greater than 10 or not.

```
//Predicate to test if a number is greater than 10
importjava.util.function.*;
class PredicateDemo1
{
public static void main(String args[])
  {
     //Use Predicate reference to show the lambda expression
     Predicate<Integer>gt = (i) -> i>10;

     //call test() method of Predicate that returns true or false
boolean result = gt.test(15);
System.out.println("Greater than 10: "+ result);

  }
}
```

Output:

```
C:\>javac PredicateDemo1.java
C:\> java PredicateDemo1
Greater than 10: true
```

In Program 15, we are creating a predicate that returns true if a number is greater than 10. Then this predicate is passed to myMethod() along with the array of numbers.

Program 15: Write a program to create a predicate and pass it to a method that displays numbers greater than 10.

```
//Predicate to display numbers greater than 10
importjava.util.function.*;
class PredicateDemo2
{
public static void main(String args[])
  {
     //take a group of Integer objects in an array
     Integer []arr = {8,9,10,11,12,13,14,15};

     //create Predicate reference to lambda expression
     Predicate<Integer>gt = (i) -> i>10;

System.out.println("Numbers greater than 10:");
     //call the method and pass predicate reference
myMethod(gt, arr);
  }

static void myMethod(Predicate<Integer> p, Integer []arr)
  {
for(Integer i: arr)
if(p.test(i)) System.out.print(i+" ");
  }
}
```

Output:

```
C:\>javac PredicateDemo2.java
C:\> java PredicateDemo2
Numbers greater than 10:
11 12 13 14 15
```

Program 16: Write a Java program to create a predicate that displays all numbers and even numbers from a group of given numbers.

```
//Predicate to display all numbers and even numbers
importjava.util.function.*;
class PredicateDemo3
{
public static void main(String args[])
  {
      //take a group of Integer objects in an array
      Integer []arr = {8,9,10,11,12,13,14,15};

      //create Predicate references
      Predicate<Integer> all, evens;

      //lambda expression to return true upon taking a number
all = (n) -> true;

      //lambda expression to return true if it is even number
evens = (n) -> n%2==0;

System.out.println("All numbers:");
display(all, arr);

System.out.println("\nEven numbers: ");
display(evens, arr);
  }

static void display(Predicate<Integer> p, Integer []arr)
  {
for(Integer i: arr)
if(p.test(i)) System.out.print(i+" ");
  }
}
```

Output:

```
C:\>javac PredicateDemo3.java
C:\> java PredicateDemo3
All numbers:
8 9 10 11 12 13 14 15
Even numbers:
8 10 12 14
```

Joining the predicates

It is possible to join two predicates using and(), or() or negate() methods. When and() is used, it will return the value of the logical AND operation of two predicates. The or() method returns the value of the logical OR operation of two predicates. The negate() method negates the value. It means, true will become false and vice versa.

For example, let us take two lambda expressions as:

```
gt = (n) -> n>10;  // returns true if n is greater than 10
lt = (n) -> n<15;  //returns true if n is lesser than 15
```

In the preceding statements, 'gt' and 'lt' are obviously references of Predicate interface. We can use the and() method and join these two as:

```
gt.and(lt)  //returns true if n is greater than 10 and less than 15
gt.and(lt).negate()  //returns true if n is not greater than 10 and not less
than 15
```

Program 17: Write a Java program to understand how to join two predicates.

```
//Predicate to display required numbers from the list
importjava.util.function.*;
class PredicateDemo4
{
public static void main(String args[])
  {
      //take a group of Integer objects in an array
      Integer []arr = {8,9,10,11,12,13,14,15};

      //create Predicate references
      Predicate<Integer>gt, lt;

      //lambda expression to return true if the number > 10
gt = (n) -> n>10;

      //lambda expression to return true if the number < 15
lt = (n) -> n<15;

System.out.println("Numbers > 10 and < 15:");
display(gt.and(lt), arr);

System.out.println("\nNumbers<= 10 and >= 15:");
display(gt.and(lt).negate(), arr);
   }

static void display(Predicate<Integer> p, Integer []arr)
  {
for(Integer i: arr)
if(p.test(i)) System.out.print(i+" ");
   }
}
```

Output:

```
C:\>javac PredicateDemo4.java
C:\> java PredicateDemo4
Numbers > 10 and < 15:
11 12 13 14
Numbers <=10 and >=15:
8 9 10 15
```

Functions

Functions are similar to predicates, but functions return an object as result. We should remember that a function returns only one value. In Java 8.0, there is an interface by the name Function<T, R> available in the java.util.function package. Since this Function is a functional interface, it can refer to a lambda expression. This Functioninterface contains a method, apply(), which takes one argument and returns an object as result. However, T represents the type of input and R represents the type of result of the Function. It is possible to join two functions using the and Then() method.

Let us create a method to find the length of a string. This method accepts a string type object and returns an Integer type object.

```
Integer myMethod(String str)
{
returnstr.length();
 }
```

The preceding code can be converted into a lambda expression as:

```
(str) ->str.length();
```

This lambda expression can be referenced by the Function interface as:

```
Function<String, Integer>len = (str) ->str.length();
```

In the preceding code, we should understand that this Function takes a String as an argument and returns an Integer type object. Hence, we have written <String, Integer> after the Function interface name. The Function interface contains a method apply() that applies this function to the argument as: len.apply(str). Now, this function returns the length of the string str. This can be seen from Program 18.

Program 18: Write a Java program to create a Function that returns the length of a string.

```
//Function to find length of a given string.
importjava.util.function.*;
class FunctionDemo1
{
public static void main(String args[])
  {
     /* create a Function reference to lambda expression
to findlength of a string */
     Function<String, Integer>len = (str) ->str.length();

     //find the length of the string str
     String str = "Dreamtech publications";
System.out.println("Length= "+ len.apply(str));
  }
}
```

Output:

```
C:\>javacFunctionDemo1.java
C:\> java FunctionDemo1
Length= 22
```

Important Interview Question

What is the difference between a predicate and a function?

A predicate is a function that takes one argument and returns a boolean value.

But, a function represents a function that takes one argument and returns an object.

Both predicates and functions are useful to evaluate lambda expressions.

Double colon operator (::)

To refer to methods and constructors of a functional interface, we can use a double colon operator in the place of lambda expression while a method is referenced. The syntax is:

```
Classname :: methodName
```

We know that public void run() method is available in Runnable interface which is a functional interface. Hence, it can be represented by a lambda expression as:

```
Runnable r1 = () ->System.out.println("Hello from lambda");
```

Now, to use a double colon operator, we should create another method with the same signature like that of public void run(), but the method name can be different. For example, we can write the void display() method in the class DColonDemo1 as:

```
static void display()
{
System.out.println("Hello from display");
 }
```

This method can be referenced using a double colon operator as: DColonDemo1::display. This is shown in Program 19.

Program 19: Write a Java program to understand how to use double colon operator as a reference to a method.

```
//double colon operator to refer to a method
class DColonDemo1
{
static void display()
   {
System.out.println("Hello from display");
   }

public static void main(String args[])
   {
     //lambda expression represents the run() method of Runnable
     Runnable r1 = () ->System.out.println("Hello from lambda");
r1.run();

     //double colon refers the display() method of DColonDemo1 class
     Runnable r2 = DColonDemo1::display;
r2.run();
   }
}

C:\>javacDColonDemo1.java
C:\>java DColonDemo1
Hello from lambda
Hello from display
```

We can also use a double colon operator to refer to the constructor of a class. The syntax is:

```
Classname:: new;
```

For example, we want to refer to the following constructor of the Sample class:

```
//constructor
Sample(String x) {
this.x = x;
System.out.println("Constructor executed "+x);
   }
```

We can use a double colon operator in the place of a lambda expression as:

```
MyInter mi1 = Sample::new;  //refer to constructor of Sample class
```

Here, mi1 is the reference of functional interface, i.e. MyInter. Now, it is possible to call the method of the functional interface using mi1. Of course, the constructor and the method declared in the functional interface should have the same format. In Program 20, we are showing how to call constructor using lambda expression and also using a double colon operator.

Program 20: Write a Java program to understand how to use a double colon operator as a reference to the constructor of a class.

```
//double colon operator to refer to a constructor
class Sample
{
    //instance variable
private String x;

    //constructor
Sample(String x) {
this.x = x;
System.out.println("Constructor executed "+x);
    }
}

// functional interface with get() method that returns Sample class object
interfaceMyInter
{
    Sample get(String str);
}

class DColonDemo2
{
public static void main(String[] args)
   {
      //this lambda expression returns Sample class object
      MyInter mi = (String x)-> {return new Sample(x);};
Sample s = mi.get("from lambda expression");

      //double colon operator refers to Sample class constructor
      MyInter mi1 = Sample::new;
      Sample s1 = mi1.get("from double colon operator");
    }
}

C:\>javacDColonDemo2.java
C:\>java DColonDemo2
Constructor executed from lambda expression
Constructor executed from double colon operator
```

Conclusion

A lambda expression is a method that does not consist of name, access specifier and return data type. Lambda expressions look like simple expressions but they work like methods. To call lambdas, we need functional interfaces. A functional interface is an interface that contains only one abstract method. The reference of a functional interface can be used to refer to a method or a lambda expression. Predicates are similar to functional interfaces but they return a boolean value. Hence predicates can also be used to refer to lambda expressions. Lambda expressions focus mainly on functional aspects of programming where coding is given importance over data.

CHAPTER

35

STREAMS API

An API (Application Programming Interface) represents a group of classes and interfaces using which a programmer can develop applications or software. In Java 8.0, a new package by the name java.util.stream has been provided for creating and working with streams. We know that a stream represents flow of data from one place to another place. This definition is slightly changed in Java 8.0. Now, a stream represents flow of elements (objects) on which manipulations can be done.

The primary aim of a stream concept is to make the operations easy on collections. A collection represents an object that contains a group of other objects. In case of a collection, first the objects are planned properly and then stored in the collection. It means a collection contains objects. When we need to make some manipulations on the objects of a collection, we can use methods that are already available in the collection framework. But, we cannot use lambda expressions in this case. Lambda expressions are developed to make manipulations on the objects easy. To utilize this advantage, we should collect all the objects of a collection into a stream and then apply manipulations through lambda expressions. It means that a stream contains objects coming from a collection that can be easily manipulated with the help of lambda expressions. This is the reason streams are generally used with collections like lists or sets.

Important Interview Question

What is the difference between a collection and a stream?

A collection refers to a group of objects in memory. A stream contains objects that are taken from a collection for the purpose of doing some manipulations.

Creating streams

To create a stream from a list, we can use `stream()` method of Stream class of java.util.stream package. For example,

```
Stream<Integer>sm = lst.stream();
```

Here, lst refers to an object of ArrayList that contains several integer objects. To convert this ArrayList into a stream, we are using the `lst.stream()` method that returns Stream class object 'sm'. Once the stream is created like this, we can use manipulation functions on the stream to obtain the desired result. Please remember that the `stream()` method is not actually converting the ArrayList, it is only retrieving the elements from the ArrayList into the Stream object 'sm'. For example, to return a stream that contains numbers greater than 5, we can use a manipulation function `filter()` as:

```
sm.filter(i -> i >5)
```

Observe the lambda expression that is passed to the `filter()` function. This `filter()` function returns a stream that contains numbers greater than 5. We can collect all these elements into a List object using `collect()` function as:

```
List<Integer> lst1 = sm.filter(i -> i >5).collect(Collectors.toList());
```

The `collect()` function takes an argument `Collectors.toList()` that is responsible for converting the stream into a list. In Program 1, we can see how to convert a list into a stream and then how to collect the resultant elements into a list.

Program 1: Write a program to create a stream with the elements of an ArrayList and get the elements which are greater than 5 into a new list.

```
/*to create a stream from ArrayList and to collect the
elements of a stream into a list */
importjava.util.*;
importjava.util.stream.*;
class Convert1
{
public static void main(String args[])
  {
      //create a list to store Integer objects
      List<Integer>lst = new ArrayList<Integer>();

      //add elements to the list
for(int i = 1; i< 10; i++)
      {
lst.add(i);
      }

       //convert this list into stream using stream()
       Stream<Integer>sm = lst.stream();

       /*filter the elements which are greater than 5 and collect
them into a list using collect(Collectors.toList()) */
       List<Integer> lst1 = sm.filter(i -> i >5).collect(Collectors.toList());

       //display the new list
System.out.print(lst1);

  }
}
```

Output:

```
C:\>javac Convert1.java
C:\> java Convert1
[6, 7, 8, 9]
```

We can convert the resultant stream into any array using `toArray()` method of Stream class. This method takes an argument: Arraytype[] ::new. See the following code:

```
Integer[] arr = sm.filter(i -> i < 5).toArray(Integer[]::new);
```

Here, the stream is filtered by the `filter()` method to extract the numbers less than 5 and then the `toArray()` method converts the resultant stream into Integer type array.

Program 2: Write a program to retrieve the numbers from an ArrayList and convert the resultant stream into an array.

```
//to convert the elements of a stream into an array
importjava.util.*;
importjava.util.stream.*;
class Convert2
{
public static void main(String args[])
  {
     //create a list to store Integer objects
     List<Integer>lst = new ArrayList<Integer>();

     //add elements to the list
for(int i = 1; i< 10; i++)
     {
lst.add(i);
     }

      //convert this list into stream using stream()
      Stream<Integer>sm = lst.stream();

      /*filter the elements which are lesser than 5 and collect
them into an Integer type array using toArray(Integer[]::new) */
Integer[] arr = sm.filter(i -> i < 5).toArray(Integer[]::new);

      //display the array
for(Integer i: arr)
System.out.println(i);

  }
}
```

Output:

```
C:\>javac Convert2.java
C:\> java Convert2
1
2
3
4
```

Stream class's `stream()` method creates streams from lists or sets. There are other methods which create steams from other sources. For example, the `Stream.of()` method takes some values or an array and constructs a Stream object.

For example,

```
Stream<Integer> sm1 = Stream.of(10,11,12,13,14,15);
Stream<Integer> sm1 = Stream.of(arr);  //arr is Integer type array
```

In the first statement, the numbers from 10 to 15 actually represent Integer objects containing these numbers. These Integer objects are stored as elements in the Stream object sm1. Now, to display the elements of the stream, we can use `forEach()` method, as:

```
sm1.forEach(System.out::println);
```

Here, double colon operator is used to connect the `println()` method with System.out. The `forEach()` method is useful to perform a specified action on each element of the stream. Here, the action that it is used to perform is to display (i.e. println).

Program 3: Write a program to understand how to create streams of objects using the `Stream.of()` method.

```
//to create stream of objects using Stream.of()
importjava.util.stream.*;
class Create1
{
public static void main(String args[])
  {
      //create a stream of Integer objects using Stream.of() method
      Stream<Integer> sm1 = Stream.of(10,11,12,13,14,15);

      //display the elements of the stream
sm1.forEach(System.out::println);

      //create another stream of array of Float objects
      Float arr[] = {1.1f, 2.2f, 3.3f, 4.4f};
      Stream<Float> sm2 = Stream.of(arr);

      //display the elements of the stream
sm2.forEach(System.out::println);

  }
}
```

Output:

```
C:\>javac Create1.java
C:\> java Create1
10
11
12
13
14
15
1.1
2.2
3.3
4.4
```

Stream class's `generate()` method executes an expression repeatedly and stores the returned values in the Stream class object as:

```
Stream<Double>sm = Stream.generate(()-> {return Math.random();});
```

Here, the `Math.random()` method returns some double type random number. This method is repeatedly executed and resultant random numbers are stored as Double type objects in the stream 'sm'.

```
//to create a stream of objects using Stream.generate()
importjava.util.stream.*;
class Create2
{
public static void main(String args[])
  {
     //create the stream from random numbers
     Stream<Double>sm = Stream.generate(()-> {return Math.random();});

     //display the elements of the stream
sm.forEach(System.out::println);

  }
}
```

Output:

```
C:\>javac Create2.java
C:\> java Create2
0.3607151974733275
0.033580161759507066
0.0875276133908558
0.16947055502738795
0.23108985045528418
0.7615935729130013
0.8953651523297982
0.021107729627530936
0.5792976773074073
0.025146401155568876
0.4079176376016268
0.6330297883964154
:
:
```

Operations on streams

Stream class offers several methods to manipulate objects. There are two types of operations which can be done on a stream.

- **Intermediate operations:** These operations return a stream. Hence, we can use other methods on this stream to get another stream as result. In this way, it is possible to use a chain of methods. Example: `filter()`, `map()`, `sorted()`.
- **Terminal operations:** These operations return a value of a certain type. Example: `forEach()`, `count()`, `collect()`.

Important Interview Question

What is the difference between intermediate operations and terminal operations in case of streams?

Intermediate operations return a stream as a result of the operation. But, terminal operations return only a value as the result.

filter() method

This method accepts a predicate and filters the stream based on the condition given in the predicate. We can also use lambda expression in the place of predicate. Then it returns the resultant stream. Since the result is a stream, we can use another function to process it. For example,

```
long n = lst.stream().filter(x ->x.length()> 4).count();
```

Here, 'lst' represents List object which is converted into a stream by `stream()` method. This method returns a stream on which the `filter()` method is acting to find all the strings which are having more than 4 characters length. The result returned by `filter()` method is again a stream on which count() method is acting to count these strings and return a long integer.

map() method

This method converts each element into another object according to the given function. For example, the following `map()` method converts each string into upper case using String class `toUpperCase()` method as:

```
names.stream().filter((s) ->s.startsWith("A"))
 .map(x ->x.toUpperCase())
                    .forEach(System.out::println);
```

Here, 'names' represents an ArrayList that contains a group of strings. This is converted into a stream with the help of `stream()` method. On that stream, `filter()` method acts and filters the strings that start with 'A'. Then `map()` method acts on the stream returned by `filter()` method and converts all those strings into upper case. Finally, `forEach()` method is displaying those strings.

sorted() method

This method sorts the elements in ascending order and returns the sorted stream. If we want to sort the elements in descending order, then we should use Comparator. Observe the following example:

```
lst.stream().sorted().map(x ->x.toUpperCase()).collect(Collectors.toList());
```

Here, 'lst' is converted into stream by `stream()` method and then sorted by `sorted()` method. Then `map()` method converts the strings into upper case. Finally, `collect()` method collects these strings and returns them as a List object.

forEach() method

This method takes a lambda expression and applies that expression for each element of the stream. It does not return any result. For example, the following `forEach()` method displays all the names of the stream 'names'.

```
names.forEach(System.out::println);
```

count() method

This method returns long integer value. It counts the number of elements in the stream. For example,

```
longn = lst.stream().filter(x ->x.startsWith("U")).count();
```

'lst' represents a list of strings that is being converted into stream by `stream()` method. The `filter()` method acts on this stream and filters the strings starting with 'U'. Then `count()` method counts such strings and returns the result into long type 'n'.

collect() method

This method collects the elements from a stream and stores them in a collection as indicated by its argument. For example,

```
lst.stream().sorted().map(x ->x.toUpperCase()).collect(Collectors.toList());
```

In the preceding statement, after sorting the stream, `map()` method converts each element into upper case and then the `collect()` method collects those elements into a list as indicated by `Collectors.toList()`.

In Program 4, we take an array as a list using `asList()` method of Arrays class as:

```
List<String>lst = Arrays.asList("USA", "Japan", "India", "China", "", "Russia",
"UK");
```

Observe that this list 'lst' contains 7 strings including an empty string. We can convert this list into a stream by simply writing `lst.stream()`. Then, we can perform required operations using Stream class methods.

Program 4: Write a program to understand how to perform some important operations on streams using Stream class methods.

```
//different operations on streams
importjava.util.*;
importjava.util.stream.*;
classStreamOperations
{
public static void main(String args[])
   {
      //take a string type array and convert into a list
```

```
        List<String>lst = Arrays.asList("USA", "Japan", "India", "China", "",
"Russia", "UK");
        //count the no. of strings with length more than 4 characters
    long n = lst.stream().filter(x ->x.length()> 4).count();
    System.out.println("No. of strings with length more than 4:\n" +n);

        //count number of strings which startswith "U"
        n = lst.stream().filter(x ->x.startsWith("U")).count();
    System.out.println("No. of strings starting with U:\n" +n);

        //remove all empty strings from the list and
        //collect them into another list
        List<String> lst1 = lst.stream().filter(x -> !x.isEmpty())
    .collect(Collectors.toList());
    System.out.println("The list after removing the empty strings:\n" +lst1);

        //sort the stream and then convert into upper case and then collect into
another list
        List<String> lst2 = lst1.stream().sorted()
    .map(x ->x.toUpperCase()).collect(Collectors.toList());
    System.out.println("The list after sorting in uppercase:\n" +lst2);

        //convert all strings to capital letters and collect them into an array
    String[] arr = lst2.stream().map(x ->x.toUpperCase())
    .toArray(String[]::new);
    System.out.println("Array of sorted strings in uppercase:");
    for(String i: arr) System.out.print(i+",");
      }
    }
```

Output:

```
    C:\>javac StreamOperations.java
    C:\> java StreamOperations
    No. of strings with length more than 4:
    4
    No. of strings starting with U:
    2
    The list after removing the empty strings:
    [USA, Japan, India, China, Russia, UK]
    The list after sorting in uppercase:
    [CHINA, INDIA, JAPAN, RUSSIA, UK, USA]
    Array of sorted strings in uppercase:
    CHINA,INDIA,JAPAN,RUSSIA,UK,USA,
```

Conclusion

A stream represents the flow of objects on which manipulations or processing can be done. Since a group of objects can be stored into a collection like a list or set, streams are used in connection with lists and sets. Once a stream is formed, it can be manipulated using lambda expressions.

CHAPTER

36

JODA-TIME

Earlier versions of Java (till Java 7.0) have java.util package classes like Calendar, GregorianCalendar, Date, TimeZone, etc.; however, even these classes do not have efficient methods to handle date and time. Moreover, some of the methods of these classes are deprecated, making it impossible for programmers to use them. JavaSoft people felt the need to enrich date and time operations, and hence, introduced Joda-Time into Java 8.0.

Joda-Time

Joda-Time is an API created by *joda.org* which offers better classes to handle date and time. This API is included into Java 8.0 with the package, java.time. The following are some basic features of Joda-Time:

- Joda-Time uses easy field accessors like `getYear()`, `getDayOfWeek()`, `getDayOfYear()`.
- Joda-Time supports 7 calendar systems like Buddhist, Coptic, Ethiopic, Gregorian, GregorianJulian, Islamic, Julian. In addition, there is a provision to create our own calendar system.
- Joda-Time provides a rich set of methods required for date and time calculations.
- Joda-Time uses a database for time zones. This database is updated manually several times a year.
- Joda-Time methods will execute faster when compared to earlier methods of Java 7.0; thus, provides better performance.
- The objects in Joda-Time are immutable. So, they are thread-safe.

Prefixes like of, from, plus etc. are used before objects to perform calculations as:

```
LocalDate dob = LocalDate.of(2012, Month.MAY, 14);
LocalDatesecondBday = dob.plusYears(2);
```

Important classes in java.time package

The following are some important classes in java.time package:

- LocalDate class represents a date in the form of year-month-day and is useful for representing a date without time and time zone.
- LocalTime class represents time of the day without time zone.
- LocalDateTime class handles both date and time without considering the time zone.
- Year class represents a year.
- Month is an enumeration to handle month.
- YearMonth class represents the year along with months.
- MonthDay class represents the day of a specific month.
- DayofWeek is an enumeration that represents day of week.
- ZoneId class specifies a time zone identifier in the format: region/city (ex: Asia/Kolkata).
- ZoneOffset class specifies a time zone offset from Greenwich / UTC time. (ex: offset for Kolkata from Greenwich is +5.30).
- Period class represents a quantity or amount of time in terms of years, months and days.

In dealing with dates and times, ISO 8601 standards are used. According to these standards, we can represent the date and time in the following format:

```
yyyy-mm-ddTHH:MM:SS.SSS
```

The date fields are separated by dashes and time fields are separated by colons.

- Four digit year
- Two digit month, where 01 is January and 12 is December
- Two digit day of month, from 01 to 31
- Two digit hour, from 00 to 23
- Two digit minute, from 00 to 59
- Two digit second, from 00 to 59
- Three decimal places for milliseconds if required

Method naming conventions

Table 36.1 lists the commonly used prefixes for methods in the java.time package:

Table 36.1

Prefix	Method type	Use
of	static factory	Creates an instance without converting the input parameters
from	static factory	Converts the input parameters to target class object

Prefix	Method type	Use
parse	static factory	Parses the input string to produce an instance of the target class
format	instance	Formats the object to produce a string
get	instance	Returns a part or value from the target object
is	instance	Queries for a value from the target object
with	instance	Returns a copy of the target object
plus	instance	Returns a copy of the target object with an amount of time added
minus	instance	Returns a copy of the target object with an amount of time subtracted
to	instance	Converts this object to another type
at	instance	Combines this object with another

Now, let us write some programs to better understand the concept of how to use methods of different classes available in the java.time package.

Program 1: Write a program to display system date and time.

```
//Using LocalDate and LocalTime class
//To display system date and time
importjava.time.*;
class Prog1 {
public static void main(String args[]){

    //LocalDate.now() gives system date into LocalDateobj
LocalDate today = LocalDate.now();

    //LocalTime.now() gives system time into LocalTimeobj
LocalTime time = LocalTime.now();

System.out.println(today);
System.out.println(time);
}
}
```

Output:

```
C:\>javac Prog1.java
C:\> java Prog1
2015-02-07
10:19:59.739
```

Program 2: Write a program to retrieve the date and time in parts separately.

```
//LocalDate and LocalTime classes
//To retrieve date and time in parts
importjava.time.*;
class Prog2 {
public static void main(String args[]){

   //current date is returned by LocalDate.now().
LocalDate date = LocalDate.now();

   //get the date, month and year from date
intdd = date.getDayOfMonth();
```

```
int mm = date.getMonthValue();
intyy = date.getYear();

System.out.printf("\n%d-%d-%d", dd, mm, yy);

   //current time is given by LocalTime.now().
LocalTime time = LocalTime.now();

   //get the hour, minute, second and nano seconds from time
int h = time.getHour();
int m = time.getMinute();
int s = time.getSecond();
int n = time.getNano();

System.out.printf("\n%d:%d:%d:%d", h,m,s,n);
}
}
```

Output:

```
C:\>javac Prog2.java
C:\> java Prog2
7-2-2015
10:28:46:663000000
```

Program 3: Write a program to create LocalDateTime class object and extract some data from that object using methods.

```
//LocalDateTime class
//To display system date and time and get some data
importjava.time.*;
class Prog3 {
public static void main(String args[]){

    //get the current date and time
LocalDateTimedt = LocalDateTime.now();
System.out.printf("%nLocalDateTime object with current date and time: %n%s",
dt);

    //create LocalDateTime object with date: 1994-4-15 and time: 11.30 am
LocalDateTime dt1 = LocalDateTime.of(1994, Month.APRIL, 15, 11, 30);
System.out.printf("%nLocalDateTime object with some date and time: %n%s",
dt1);

    //find out the date and time from 6 months now
System.out.printf("%n6 months from now: %n%s", dt.plusMonths(6));

    //find the date and time 6 months before from now
System.out.printf("%n6 months ago: %n%s", dt.minusMonths(6));

    //get the day of week for the current date and time
DayOfWeekdw = dt.getDayOfWeek();

    //represent day of week with its name
    String s = dw.name();
System.out.printf("%nDay of week name: %n%s", s);

    //represent day of week with integer value
int n = dw.getValue();
System.out.printf("%nDay of week value: %n%s", n);
}
}
```

Output:

```
C:\>javac Prog3.java
C:\> java Prog3
LocalDateTime object with current date and time:
2015-02-07T11:07:32.675
LocalDateTime object with some date and time:
1994-04-15T11:30
6 months from now:
2015-08-07T11:07:32.675
6 months ago:
2014-08-07T11:07:32.675
Day of week name:
SATURDAY
Day of week value:
6
```

Program 4: Write a program to know the current time zone and find the date and time at the current location and at Los Angeles in USA.

```
//ZoneId and ZonedDateTime classes
//To know the time zone of our country
importjava.time.*;
class Prog4 {
public static void main(String args[]){

    /* get the default timezone. If this program is executed in India,
then the default timezone represents India */
ZoneId zone = ZoneId.systemDefault();
System.out.printf("%nCurrenttimezone= %s ", zone);

    //get the date and time in the default timezone
LocalDateTimedt = LocalDateTime.now();
System.out.printf("%nDate and Time in India: %s", dt);

    //get the zone identification for Los Angeles
ZoneId la = ZoneId.of("America/Los_Angeles");

    //get the date and time in Los Angeles
ZonedDateTimezdt = ZonedDateTime.now(la);
System.out.printf("%nDate and Time in Los Angeles: %s", zdt);
}
}
```

Output:

```
C:\>javac Prog4.java
C:\> java Prog4
Current timezone= Asia/Calcutta
Date and Time in India: 2015-02-07T11:35:35.704
Date and Time in Los Angeles: 2015-02-06T22:05:35.704-08:00[America/Los_Angeles]
```

Program 5: Write a program to know the number of days between your birthday and today.

```
//Period class
//To know number days between two dates
importjava.time.*;
class Prog5 {
public static void main(String args[]){

    //get today's date
LocalDate today = LocalDate.now();
```

```
    //create LocalDate object with your birth date
LocalDate birthday = LocalDate.of(1990, Month.JANUARY, 1);

    //find the number of days between using Period class between().
    Period p = Period.between(birthday, today);
System.out.printf("You are %d years %d months and %d days older.",
p.getYears(), p.getMonths(), p.getDays());

}
}
```

Output:

```
C:\>javac Prog5.java
C:\> java Prog5
You are 25 years 1 months and 6 days older.
```

Program 6: Write a program to test whether a given year is leap year or not.

```
//Year class
//To know whether a year is leap or not
importjava.time.*;
class Prog6 {
public static void main(String args[]){

    //take the year
int n = 2015;

    //create Year class object with that year
    Year y = Year.of(n);

    //test if y is leap or not using isLeap()
boolean flag = y.isLeap();  //flag is true if leap

if(flag) System.out.printf("%nYear %d is Leap.", n);
elseSystem.out.printf("%nYear %d is not leap.", n);
}
}
```

Output:

```
C:\>javac Prog6.java
C:\> java Prog6
Year 2015 is not leap.
```

Conclusion

To make date and time related calculations more convenient to perform, JavaSoft people have introduced Joda-Time API in Java 8 version. This API uses very easy to use methods to handle date and time related processing. Moreover, prefixes like of, from, is etc. are used with these methods to make the processing easy.